N° d'ordre : 140

Année 1988

THÈSE

présentée à

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

en vue de l'obtention du diplôme de

DOCTEUR INGENIEUR

SPECIALITE : AUTOMATIQUE

par

Pierre BACQUET

ENVIRONNEMENT DE PROGRAMMATION
POUR LA SYNCHRONISATION DE PROCESSUS
SUR CALCULATEUR TEMPS REEL : LIPS.

Soutenue le 21 décembre 1988 devant la Commission d'Examen :

MM.	**A. FOURNIER**	**Président**
	R. ALAMI	**Examinateurs**
	C. BARROUIL	
	P. CARTON	
	E. DOMBRE	
	J. GILLON	

A mon fils Olivier, dont la naissance n'a pas contribué à la rapidité de ma rédaction

A Steve JOBS qui a indirectement participé à l'ergonomie de ma rédaction

REMERCIEMENTS

Je tiens tout d'abord à exprimer ma reconnaissance à MM. Jean-Pierre FORESTIER et Marc LABARRERE, responsables successifs du Département d'Etudes et de Recherche en Automatique qui m'ont permis d'effectuer mon travail au sein de leur laboratoire.

Je tiens particulièrement à remercier M. Jacques GILLON, mon directeur de thèse qui a très rapidement cru aux possibilités de LIPS et a su mettre en œuvre tous les moyens nécessaires à la réalisation de cet environnement.

Je remercie les membres du Jury et son président, M. Alain FOURNIER pour avoir accepté de juger mon travail en cette période très chargée de fin d'année.

Je ne saurais non plus oublier M. Patrick CARTON, "co-géniteur" avec M. GILLON de CESAR, qui m'a fait découvrir le fonctionnement du système ainsi que de nombreuses astuces de programmation.

Qu'il me soit aussi permis de remercier MM. Hervé POILVE et Alain ROBERT sans qui LIPS ne serait pas ce qu'il est actuellement.

De même, je tiens à remercier MM. Christophe GRAULLE et Jean-Louis FERRARI qui ont travaillé sur la partie "hors ligne" et hors temps-réel (développement du logiciel PC-LIPS) et qui ont grandement contribué à l'ergonomie du produit final.

MM. Vincent FUERTES et Michel GRAU n'ont pas hésité à sacrifier une partie de leur temps pour me former au système de développement que j'ai utilisé; qu'ils soient ici remerciés de leur patience.

Je remercie aussi tous les utilisateurs de LIPS qui m'ont fait part de leurs réflexions et de leurs besoins; en particulier, MM. Claude BARROUIL et Jean-Pierre CHRETIEN.

Enfin, un grand merci à tout l'équipe logistique du DERA: Lilliane ARENDO et Pierre GUILLEMOT pour leur aide constante durant mon séjour au DERA.

TABLE DES MATIERES

INTRODUCTION 1

I. Les systèmes temps réel 5

I.1. Généralités sur les systèmes temps réel 5
I.1.1. Introduction 5
I.1.2. Caractéristiques des systèmes temps réel 5
I.1.2.1. Adaptation à l'environnement 5
I.1.2.1.1 Dispositifs d'interfaces entrées/sorties nombreux et variés 5
I.1.2.1.2 Prise en compte d'événements externes asynchrones 6
I.1.2.1.3 Temps de réponse et délai de cohérence 6
I.1.2.2. Non déterminisme des systèmes temps réel 7
I.1.2.3. Agents actifs: Tâches ou Processus 8
I.1.2.3.1 Définitions 8
I.1.2.3.2 Parallélisme 9
I.1.2.3.3 Relations entre tâches 9
I.1.2.4. Ressources d'un système temps réel 10
I.1.2.4.1 Ressources de synchronisation 10
I.1.2.4.2 Ressources de communication 11
I.1.2.5. Coopération et concurrence des tâches 12
I.2. Application au système CESAR 12
I.2.1. Matériel 12
I.2.1.1. Configuration mono-processeurs 13
I.2.1.2. Configuration multi-processeurs 14
I.2.1.3 Application à CESAR 16
I.2.2. Etats d'une tâche 16
I.2.3. Notion de requête 18
I.2.4. Les différentes ressources de CESAR 22
I.2.4.1. Les ressources de synchronisation 23
I.2.4.1.1 Ressources de cadencement 23
I.2.4.1.2 Ressources IT Soft 25
I.2.4.1.3 File d'ordre 26
I.2.4.2. Les ressources de données 29
I.2.4.2.1 Les ressources d'espace mémoire 29
I.2.4.2.2 Les files d'échange 29
I.2.4.2.3 Les ressources d'Entrée/Sortie 31

II. Synchronisation de requêtes et extension de la notion de tâche 35

II.1 Introduction 35
II.2 Le Compilateur d'expressions logiques 36
II.2.1 Exemple pratique 36
II.2.2 Intégration du compilateur d'expressions logiques à la chaîne de développement 38
II.2.3 Structures et fonctionnement du "compilateur d'expressions logiques" 39
II.2.3.1 Préliminaire 39
II.2.3.2 Généralisation 39
II.2.3.3 Structures utilisées 40
II.2.3.3.1 Structures associées à l'ordre $IF 40

II.2.3.4 Implantation pratique sous CESAR 43
II.2.3.4.1 Présentation de la chaîne de développement 43
II.2.3.4.2 Structures des nœuds sous CESAR 44
II.2.3.5 Fonctionnement du "compilateur d'expressions logiques" 46
II.2.3.5.1 Structure générale du compilateur d'expressions logiques 46
II.2.3.5.2 Génération des structures associées à l'ordre $WAIT 48
II.2.3.5.3 Analyse syntaxique de l'expression logique associée à un ordre $IF 50
II.3 Tache généralisée 54
II.3.1 Introduction 54
II.3.2 Concepts utilisés 55
II.3.2.1 Synchronisation hiérarchisée - Notion de bloc 55
II.3.2.2 Ressource logicielle 57
II.3.3 Synchronisation par les données 58
II.3.3.1 Réalisation sous CESAR 59
II.3.3.2 Description d'une tâche généralisée 60
II.3.3.2.1 Les ordres 61
II.3.3.2.2 Les événements 61
II.3.3.2.3 Conditions internes à la tâche
II.3.3.2.4 Propagation de certains ordres 66
II.3.3.3 Structures utilisées 67
II.3.3.3.1 Dialogue par file d'ordre 69
II.3.3.3.2 Table d'état d'une tâche généralisée
II.3.4 Fonctionnement du noyau de gestion des tâches généralisées 76
II.3.4.1 Préliminaires 76
II.3.4.2 Principes généraux 76
II.3.4.3 Utilisation des logiciels développés 76
II.3.4.3.1 Ecriture d'une tâche généralisée 76
II.3.4.3.2 Configuration de l'application 77
II.4 Intéret respectif des deux logiciels et exemples 80
II.4.1 Notations utilisées 80
II.4.2 Tâche gardée 81
II.4.3 Autres synchronisations possibles 81
II.5 Conclusion 83

III. L'environnement logiciel LIPS 87

III.1 Organisation logicielle de LIPS 89
III.1.1 Initialisation de l'environnement LIPS 90
III.1.2 Connexion d'une primitive à l'environnement LIPS 91
III.1.3 Généralités sur le traitement des commandes LIPS 93
III.1.3.1 Organisation des modules en mémoire 93
III.1.3.1.1 Chargement d'un module: commande load 95
III.1.3.1.2 Déchargement d'un module: commande download 95
III.1.3.1.3 Exécution d'un module: commandes call et step 96
III.1.3.1.4 Arrêt d'un programme: commande stopg 97
III.1.3.1.5 Reprise de l'exécution d'un programme: commande resume 97
III.2 Fonctionnement du synchroniseur 97
III.3 Structures et fonctionnement de l'interpréteur CLIPS 100
III.3.1 Généralités 101
III.3.2 Structure des fichiers sources - Communication entre

III.3.3 Avant-Propos....108
III.3.4 Fonctionnement de l'analyseur lexicographique....108
III.3.5 Fonctionnement de l'analyseur syntaxique....114
III.3.5.1 Interface avec le superviseur....114
III.3.5.2 Initialisation de l'interpréteur....116
III.3.6 Structures et fonctionnement de l'analyseur syntaxique....117
III.3.6.1 Pile des drapeaux....118
III.3.6.2 Traitement d'un bloc case....121
III.3.6.3 Traitement des boucles while....122
III.3.6.4 Manipulations de pile....124
III.3.6.4.1 Traitement des ordres break et continue à l'intérieur d'une boucle while....125
III.3.6.5 Traitement de l'appel à un sous-programme....130
III.3.6.6 Traitement de l'instruction exit....133
III.3.6.7 Traitement de l'instruction return....133
III.3.6.8 Traitement de l'appel à une primitive....133
III.3.7 Gestion des variables LIPS....134
III.3.7.1 gestionnaire des variables et traitement des expressions....135
III.3.7.2 Principes généraux du fonctionnement du gestionnaire des variables....135
III.3.7.2.1 Cas particulier des variables de type repère....137
III.3.7.2.2 Stratégies d'exploration de l'arborescence
III.3.7.2.3 Expressions non entières....142
III.3.7.2.4 Evaluation des paramètres d'appel d'un module....143

IV CONCLUSION....147

LISTING DU COMPILATEUR....A.1

LISTING DE LA TACHE GENERALISEE....B.1

LISTING DE L'ANALYSEUR LEXICOGRAPHIQUE....C.1

LISTING DE L'ANALYSEUR SYNTAXIQUE....D.1

DIAGRAMMES SYNTAXIQUE DE CLIPS....E.1

REFERENCES BIBLIOGRAPHIQUES....F.1

BIBLIOGRAPHIE....F.2

INTRODUCTION

Nous assistons, à l'heure actuelle, à l'automatisation d'un nombre croissant de processus. Cette évolution a été rendue nécessaire dans de nombreux cas:

- Chaque fois que le processus à commander était trop complexe pour un opérateur humain.
- Chaque fois que le temps de réponse d'un opérateur humain était trop pénalisant.
- Chaque fois que l'opérateur humain aurait du agir en environnement hostile.
- Pour décharger un opérateur humain de travaux rébarbatifs ou pénibles.

...

Le dernier demi-siècle a vu se succéder de nombreux types de machines de commande aux technologies toujours plus complexes: automates à relais, automates en logique cablée (éventuellement associés à des opérateurs electroniques analogiques), automates programmables, calculateurs de commande temps réel.

Quelle que soit la complexité du processus à commander, et quelle que soit l'architecture du système de commande il est toujours possible de décomposer un système de commande en deux parties: une partie synchronisation et une partie opérative.

Il existe à l'heure actuelle de nombreuses méthodes de spécification et de validation des logiciels composant la partie opérative d'un système de commande.

Par contre, dans les systèmes temps réel, la notion de synchronisation de processus (aussi bien temporelle que par les données) est fondamentale. Le non respect de certaines règles pratiques pouvant conduire à un fonctionnement erratique du système ou même à une absence de fonctionnement (blocage). Dans un système temps réel, de surcroît multi-processeurs comme CESAR, en l'absence de tout outil, les problèmes de synchronisation peuvent devenir beaucoup plus préoccupants.

En général, la synchronisation de processus dans une application temps réel peut se faire suivant deux modes: synchronisation temporelle et synchronisation par les données; ces deux modes étant souvent employés simultanément dans une même application. Il arrive fréquemment, qu'en phase de validation ou d'intégration d'un logiciel temps réel, l'utilisateur soit amené à revoir la synchronisation des différents processus; ce qui peut souvent amener à une complète refonte de modules logiciels déjà validés. Afin de minimiser les opérations nécessaires à la reconfiguration de la synchronisation d'une application, nous proposerons une méthode et des logiciels permettant, de séparer le plus possible la partie "calculatoire" pure de la partie synchronisation et ainsi de retrouver au niveau méthodologique la division fonctionnelle en deux parties

une extension de la notion de tâche telle qu'elle est définie sous la grande majorité des moniteurs temps-réel. Nous en étudierons par la suite, une implantation pratique sur le système temps réel CESAR.

Par la suite, certaines des notions définies dans la méthode annoncée ci-dessus ont été reprises et appliquées au cas particulier de la robotique dans le souci de réaliser un environnement de développement le plus ouvert et le plus flexible possible et doté de fonctions de synchronisation évoluées (parallélisme explicite, tâches gardées, ...). Cet environnement a pour nom LIPS.

Enfin, il semble inconcevable de définir un environnement de développement pour la robotique sans le doter d'un langage de programmation afin de pouvoir essayer les différents scénarii possibles d'une même application ou de réaliser plusieurs manipulations sur un même site. Dans ce but, nous avons doté l'environnement LIPS d'un interpréteur de langage structuré conservant toutes les possibilités de synchronisation déjà offertes par LIPS.

CHAPITRE I
LES SYSTEMES TEMPS RÉEL

I. LES SYSTEMES TEMPS RÉEL

I.1. Généralités sur les systèmes temps réel

I.1.1. Introduction

Un système informatique fonctionnant en temps réel est un système de traitement de l'information ayant pour mission de contrôler un processus, que nous appellerons application, en respectant les contraintes de temps et de débit de l'information caractéristiques de cette application . Cette mission de contrôle est réalisée par un ensemble d'actions fonctionnellement et temporellement distinctes et ayant, éventuellement , des relations entre elles.

Un système informatique temps réel peut être considéré comme un ensemble de dispositifs informatiques matériels, associés à du logiciel, en relation étroite avec son environnement, l'application contrôlée. Système informatique et application inter-agissent pour assurer, avec un maximum d'efficacité, la mission confiée à l'ensemble de la réalisation.

Il est difficile de donner une définition synthétique et exhaustive des systèmes informatiques temps réels qui mettent en œuvre des concepts complexes et variés. Il est, sans doute, plus efficace de les caractériser suivant un ensemble de critères reconnus de façon générale.

I.1.2. Caractéristiques des systèmes temps réel

I.1.2.1. Adaptation à l'environnement

I.1.2.1.1 Dispositifs d'interfaces entrées/sorties nombreux et variés

Un système informatique temps réel comporte un nombre relativement important de dispositifs d'interfaces.

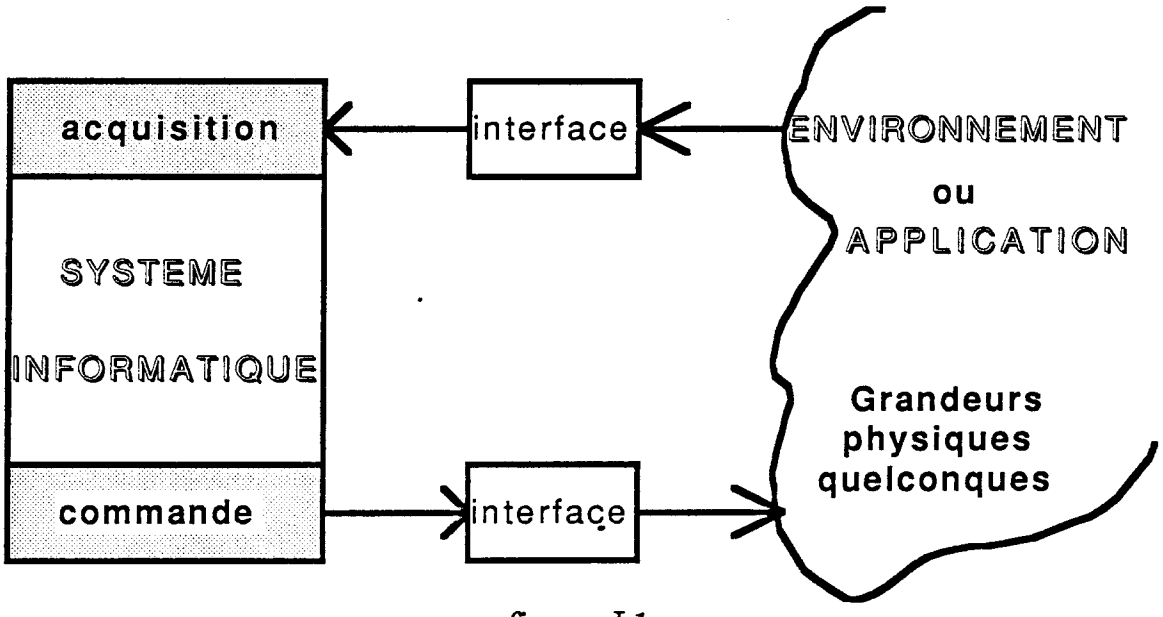


figure I.1

Les interfaces d'acquisition assurent la transformation et l'adaptation d'une information de nature, d'amplitude, de caractéristique temporelle quelconques provenant de l'environnement en une information binaire pouvant être traitée par le sytème informatique.

Inversement, les interfaces de commande transforment l'information binaire en une information adaptée au contrôle de l'application.

I.1.2.1.2 Prise en compte d'événements externes asynchrones

Un système informatique temps réel met en oeuvre les dispositifs matériels et les logiciels associés nécessaires à la prise en compte de sollicitations ou événements provenant de l'environnement. Ces événements sont "asynchrones" c'est-à-dire que leur occurence n'est pas liée au déroulement du logiciel.

Un système informatique temps réel est généralement équipé d'un système de gestion d'interruptions performant. Un tel dispositif assure la prise en compte d'événements externes asynchrones matérialisés par des signaux électriques, niveaux ou impulsions: les interruptions. Il génère, au niveau de la machine informatique un déroutement (saut) dans l'exécution du programme en cours réalisant ainsi une fonction associée (attachée) à l'événement externe en lieu et place de la fonction courante exécutée.

I.1.2.1.3 Temps de réponse et délai de cohérence

Le temps de réponse peut être défini comme le temps que met le système informatique pour répondre à une sollicitation externe asynchrone et exécuter, de façon complète, les algorithmes relatifs au traitement de cette sollicitation.

Ce temps de réponse doit, en tout état de cause, rester inférieur au temps au-delà duquel le fonctionnement correct du système de commande n'est plus garanti (délai de cohérence).

Le temps de réaction sépare l'occurence de la sollicitation et le début effectif du traitement relatif à cette sollicitation.

Temps de réponse = temps de traitement + temps de réaction

Temps de réponse <= délai de cohérence de fonctionnement du système

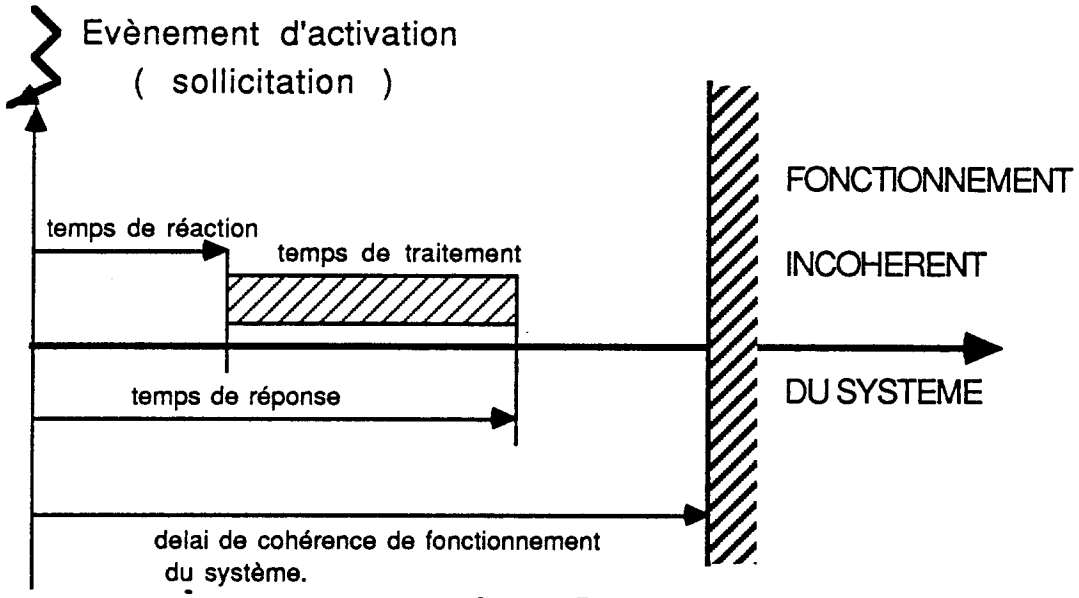


figure I.2

I.1.2.2. Non déterminisme des systèmes temps réel

Les séquences d'actions réalisant la mission d'un système temps réel peuvent être interrompues par des événements externes asynchrones, ces événements étant eux mêmes imprévisibles dans le temps et quant à l'ordre de leur arrivée. Ces événements conditionnent généralement la réalisation de séquences d'actions particulières. L'ordre de traitement de ces séquences devient donc imprévisible.

Cette imprévision nous permet de définir un système temps réel comme un système évoluant de manière non-déterministe.

Exemple :

Soit une mission constituée de 4 actions distinctes notées A1, A2, A3 et A4.

Lorsque A1 est terminée, A2 est déclenchée. Lorsque A2 est terminée, A1 est déclenchée : fonctionnement bouclé de A1 et A2.

A3 est déclenchée par un événement externe EV3. De même, A4 est déclenchée par un événement externe EV4.

A4 ne peut être interrompue par A3.

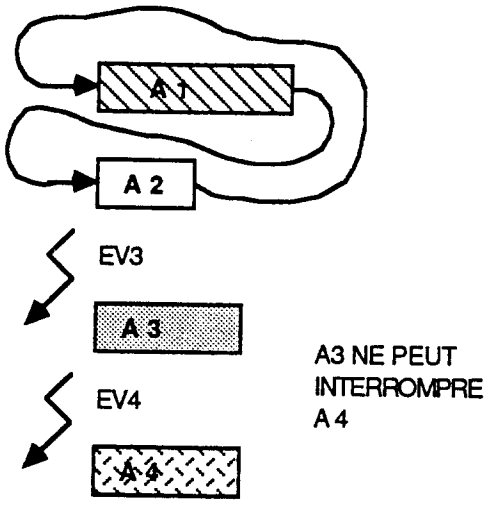


figure I.3

Le diagramme suivant met bien en évidence, sur trois cas possibles d'enchaînements des actions, le non déterminisme de l'état du système à un instant donné.

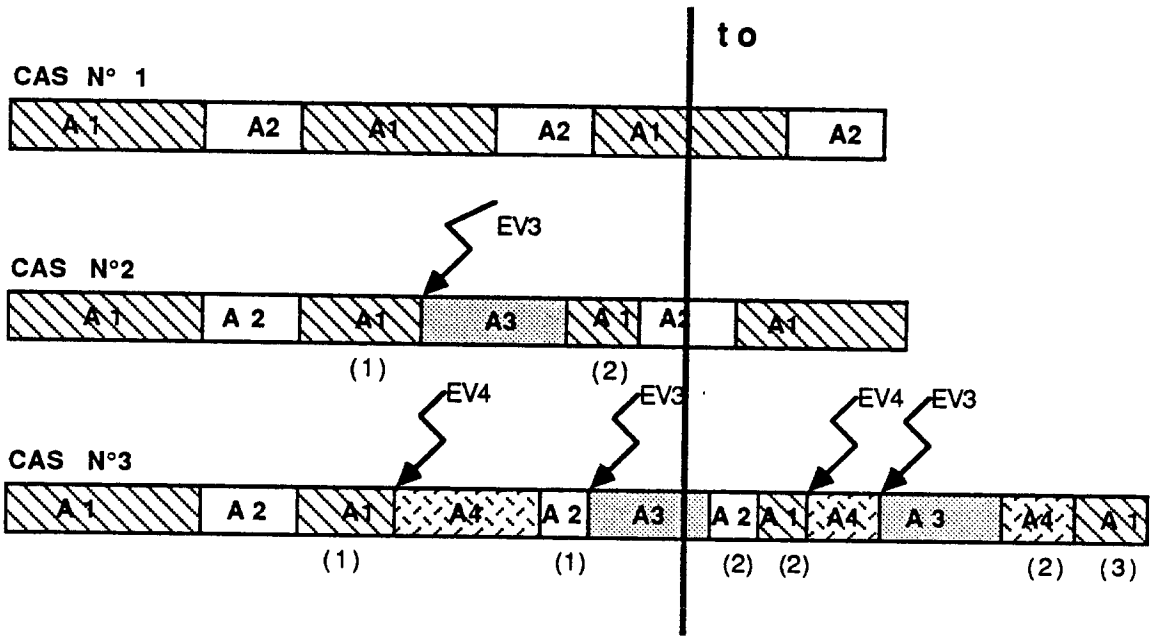


figure I.4

Au temps to, dans le premier cas, c'est A1 qui est active, dans le deuxième, c'est A2 et dans le troisième A3.

I.1.2.3. Agents actifs: Tâches ou Processus

I.1.2.3.1 Définitions

La mission du système informatique est réalisée par un ensemble d'agents actifs réalisant une activité élémentaire et cohérente correspondant à l'exécution d'un algorithme sur un processeur matériel et agissant sur un ensemble de données associées.

Les différents agents actifs ont des contraintes d'activation, délai, activation périodique, occurrence d'un événement externe asynchrone, synchronisation... et des contraintes de durée, temps de réponse comparé au délai de cohérence de fonctionnement du système, qui leur sont propres.

Ces agents actifs sont appelés tâches ou processus.

Une tâche peut donc être définie comme du logiciel, c'est-à-dire, un ensemble d'algorithmes et de structures de données associées qui prend sa valeur fonctionnelle lorsqu'il est exécuté, activé, sur un processeur. A chaque tâche correspond des conditions d'activations qui lui sont propres ainsi que des contraintes de durée particulières.

I.1.2.3.2 Parallélisme

Certains agents actifs doivent pouvoir évoluer EN PARALLELE alors qu'une machine informatique conventionnelle (à achitecture de Von Neumann [MEYN 71] appelée aussi SISD [SYRE 83]) gère, par définition, sur un seul processeur, ses activités de façon séquentielle.

Pour régler ce problème essentiel du parallélisme, un système informatique peut être :

- Monoprocesseur: un seul processeur exécute les différentes tâches assurant la mission du système. Il s'agit donc, en fait, de pseudo parallélisme.

  Le temps nécessaire au traitement effectif des algorithmes réalisant l'activité de la tâche doit être sensiblement plus petit que le délai de cohérence de fonctionnement du système.

  Le processeur peut alors exécuter en séquence les algorithmes correspondant aux diverses tâches tout en respectant la fin des délais de cohérence et assurer ainsi un parallélisme global.
- Multiprocesseurs: systèmes fortement couplés, en général, par partage d'une mémoire commune. Plusieurs processeurs, unités centrales, mettent en oeuvre les différents logiciels de l'application.
- Distribué: systèmes faiblement couplés. Les différentes machines informatiques (processeurs associés à des mémoires privées) sont liées par des voies de communication, sous forme de réseau ou par l'intermédiaire de bus partagé.

I.1.2.3.3 Relations entre tâches

En général, aucun agent actif, processus ou tâche, n'est totalement isolé:

Il peut communiquer avec d'autres agents actifs. Il échange des signaux de synchronisation ou pouvant réaliser d'autres fonctionnalités. Il peut également être amené à échanger des informations données.

Il peut détruire d'autres agents actifs, suspendre provisoirement leur activité, faire reprendre cette activité.

I.1.2.4. Ressources d'un système temps réel

Les disponibilités, LES RESSOURCES, d'un système informatique sont limitées et doivent être partagées entre les différents agents actifs qui en ont besoin pour réaliser leur fonction.

On peut distinguer les ressources physiques et les ressources logiques:

Ressources physiques:

- Unité centrale
- Mémoire centrale
- Organes Entrées/Sorties
- Organes de dialogue (imprimante, console)
- Mémoire de masse
- etc, etc . . .

Ressources logiques:

- Mécanismes logiciels
- Tâches, sous programmes utilitaires
- Structures de données, dictionnaires, fichiers, données partagées
- Délais, événements logiques échangés entre tâches.
- etc, etc . . .

Cependant il est possible de définir un ensemble de services que tout système temps réel digne de ce nom doit offrir:

I.1.2.4.1 Ressources de synchronisation

I.1.2.4.1.1 Horloges

Un tel service permet en général la "mise en sommeil" d'un processus pendant un temps déterminé ou son activation périodique Notons que certains exécutifs introduisent la notion de requête avec délai maximum de traitement: toute requête peut être annulée si elle n'a pas été traitée dans un certain laps de temps; le processus émetteur sera alors réveillé lors de la complétion de la requête ou lorsque le délai sera expiré.

I.1.2.4.1.2 Signaux

Un signal est un mécanisme logiciel permettant la synchronisation temporelle d'au moins deux processus. Il est en général levé par un processus qui désire avertir un ou plusieurs autres processus de la fin d'un traitement. Les processus destinataires du signal restent bloqués tant que le signal n'est pas levé.

I.1.2.4.2 Ressources de communication

I.1.2.4.2.1 Boîtes aux lettres

D'un côté d'une boîte aux lettres une tâche produit des données:

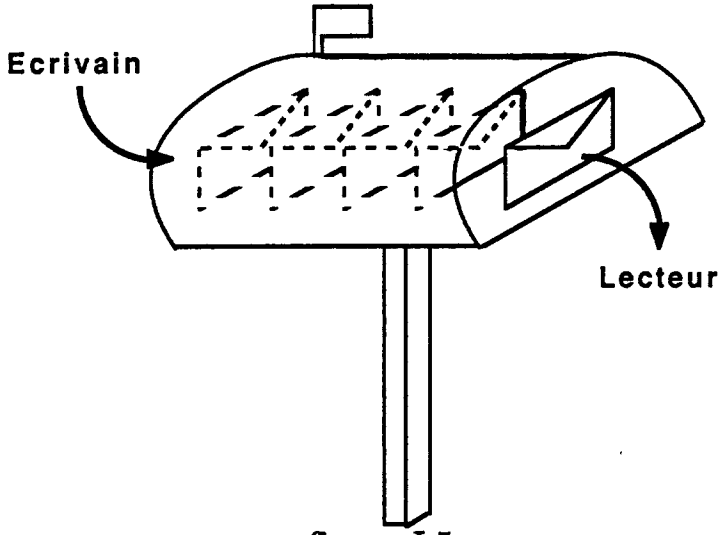


figure I.5

de l'autre côté, une tâche lectrice va "consommer" les données par ordre d'ancienneté: le message "posté" le premier sera lu le premier. En général, une requête d'écriture dans une boîte aux lettres ne sera considérée comme terminée que lorsque le message associé sera lu. La réciproque est aussi vraie: une requête de lecture en attente n'est satisfaite que lorsqu'une entité écrira un message à l'autre bout de la boîte aux lettres. En fonction des modes de synchronisation de l'exécutif temps réel (requêtes bloquantes ou non bloquantes), plusieurs modes de fonctionnement peuvent être envisagés:

- Scrutation sans attente de la boîte aux lettres.
- Ecriture de message sans attente.
- Synchronisation producteur-consommateur.

Ce dernier mode est décrit par le réseau de Petri suivant:

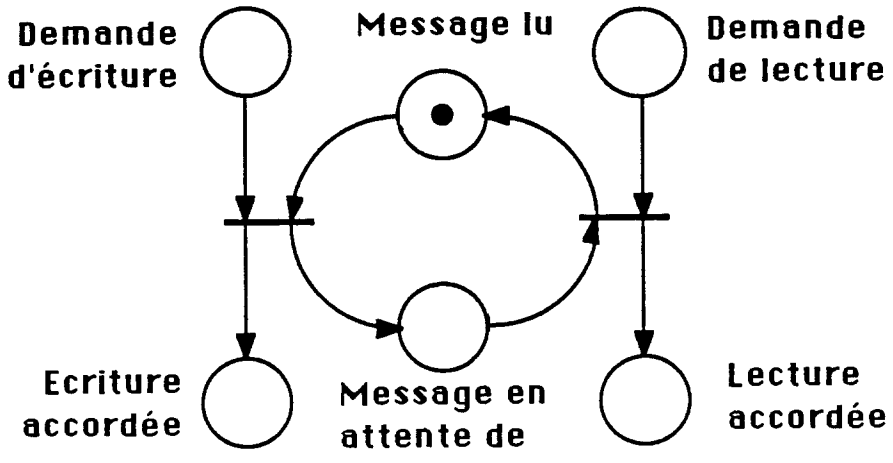


figure I.6

Suivant les exécutifs, les boîtes aux lettres peuvent être à capacité fixe (dans ce cas, les deux places centrales du réseau de Petri précédent sont bornées) ou à capacité infinie.

I.1.2.4.2.2 Ressources d'entrée-sortie

Elle permettent la communication entre le système de commande et le monde extérieur (opérateur humain, certains types de capteurs et d'actionneurs, ...).

I.1.2.5. Coopération et concurrence des tâches

Les différentes tâches doivent COOPERER entre elles pour assurer la mission du système : communiquer ou partager des informations en assurant leur intégrité, se synchroniser, c'est-à-dire respecter un ordre cohérent d'exécution.

Elles peuvent, également, entrer en CONCURRENCE pour accéder à des ressources limitées du système informatique. L'intégrité de ces accès doit être assurée.

I.2. Application au système CESAR

Les limitations des machines à architecture classique de Von Neumann apparaîssent de façon cruciale chaque fois qu'il s'agit de commander un processus rapide: dès que le temps de réponse imposé par le processeur devient supérieur au délai de cohérence, l'architecture de Von Neumann ne convient plus. Deux solutions sont alors possibles:

- Conserver la même architecture SISD mais avec un processeur plus rapide. Cette solution n'est pas viable en pratique, car elle s'accompagne généralement d'un changement de technologie du processeur et impose une modification des logiciels de commande pour les adapter au nouveau processeur.
- Abandonner l'architecture SISD au profit d'une architecture parallèlisable.

C'est cette dernière solution qui a été retenu lors de la définition du système temps réel CESAR, qui dès le départ, a été conçu pour fonctionner en environnement multi-processeurs. Une autre clause importante de son cahier des charges était que la reconfiguration d'une application (passage de n processeurs à $n+1$, en cas de sous-estimation de la puissance de calcul par exmple) soit la plus facile possible. Enfin, de manière à minimiser le temps de latence de certaines tâches critiques, l'ordonnanceur de CESAR ne sera activé qu'à chaque terminaison de requête: il n'y a pas d'ordonnancement "à priori" toutes les x milli-secondes comme dans certains exécutifs temps réel (OS9 par exemple); un tel type de fonctionnement augmente l'activité du système, souvent pour rien, et conduit, dans le pire des cas à des temps de latence de x milli-secondes.

I.2.1. Matériel

Le système temps réel CESAR est actuellement implanté sur plusieurs

Bien qu'il soit multiprocesseur, dans le cas d'applications ne nécessitant pas un volume de calcul très important, CESAR peut être utilisé en configuration mono-processeur. Nous nous intéresserons donc aux structures de base dans le cas de la configuration mono-processeur puis nous envisagerons les modifications à y apporter pour passer en configuration multi-processeurs.

I.2.1.1. Configuration mono-processeur

La configuration minimale du système CESAR est organisée autour d'un processeur 16 bits. Par son bus local, il accède à une zone de mémoire vive (16 k-octets minimun) et aux mémoires mortes contenant le système, ses drivers d'entrées-sorties et éventuellement des bibliothèques spécifiques à l'utilisateur. En plus de la mémoire, il est nécessaire de disposer d'un composant capable de gérer une liaison RS232 et d'un circuit de temporisation qui génère une horloge utilisée par les applications nécessitant un cadencement temporel.

A la mise sous tension, ou lors d'une réinitialisation du processeur, le système se trouve dans un état minimal[1]: aucune structure n'est créée, aucun processus n'existe.

Dans cet état, le processeur scrute sa liaison RS232 et attend des messages en provenance du monde extérieur. En effet, comme à l'initialisation, le système est "nu", il faut qu'un calculateur hôte, servant de machine de développement, configure le système en y affectant l'espace mémoire nécessaire à la création des tables d'état du système. Une fois cette opération effectuée, l'utilisateur crée les ressources nécessaires à son application et en télécharge le code depuis le calculateur hôte.

La figure suivante résume la configuration minimale d'une carte processeur portant CESAR.

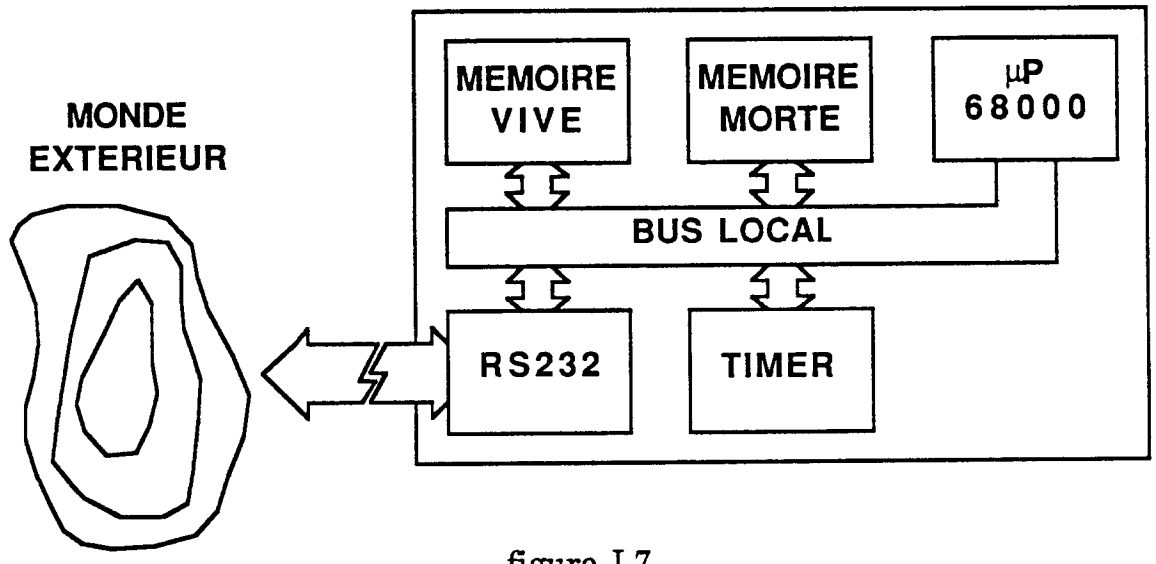


figure I.7

I.2.1.2. Configuration multi-processeurs

En fonctionnement multi-processeurs, deux nouvelles contraintes surgissent:

- Les tables système doivent être accessibles à n'importe quel processeur; il est donc indispensable de disposer d'une zone mémoire commune qui contiendra ces tables, et donc d'un bus d'échange permettant aux différents processeurs d'accéder aux ressources matérielles communes: le bus actuellement retenu dans la grande majorité des applications est le bus VME.
- Il doit être possible à n'importe quel processeur P_i d'interrompre tout autre processeur P_j dans le cas où un processus porté par P_i émet une requête vers une entité connue de P_j seulement.

De ces deux contraintes découle la configuration matérielle retenue:

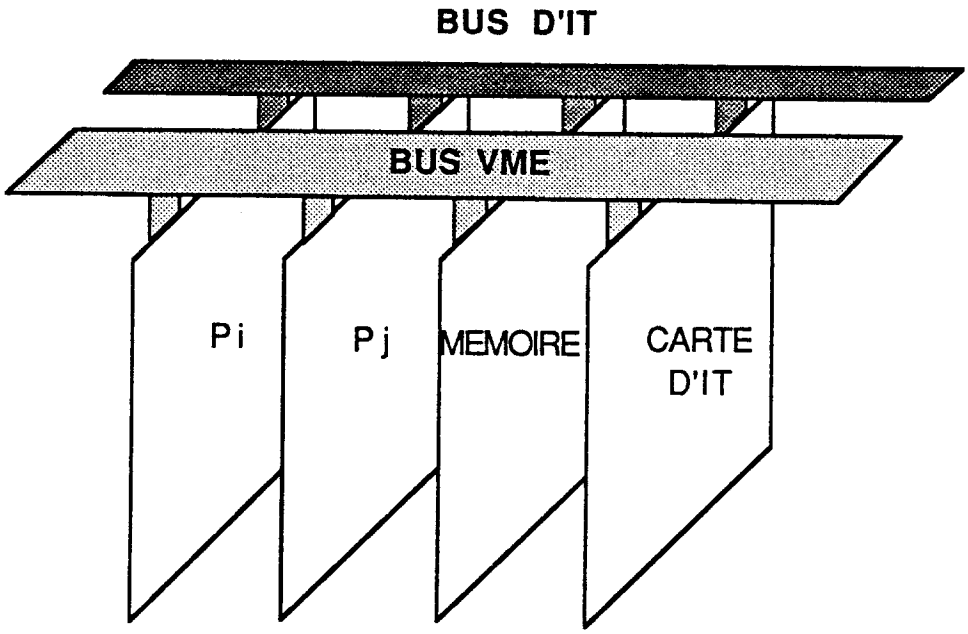


figure I.8

Avec pour chaque carte processeur:

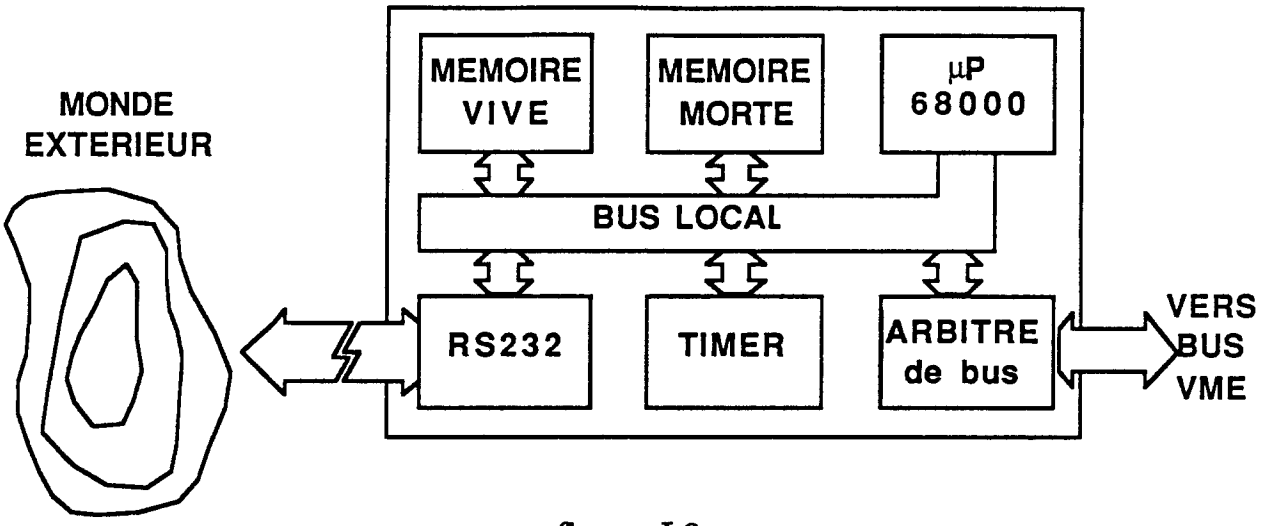


figure I.9

Le bus VME assure ici les communications entre les cartes "maîtres" (processeurs) et les cartes esclaves (mémoire globale, carte d'InTerruption, périphériques, ...).

Comme le bus VME ne permet pas à un processeur d'interrompre sélectivement un autre processeur[1] (contrairement à Multibus II), il a été nécessaire de rajouter l'ensemble carte d'interruption + bus d'interruption, dont le fonctionnement peut se résumer comme suit:

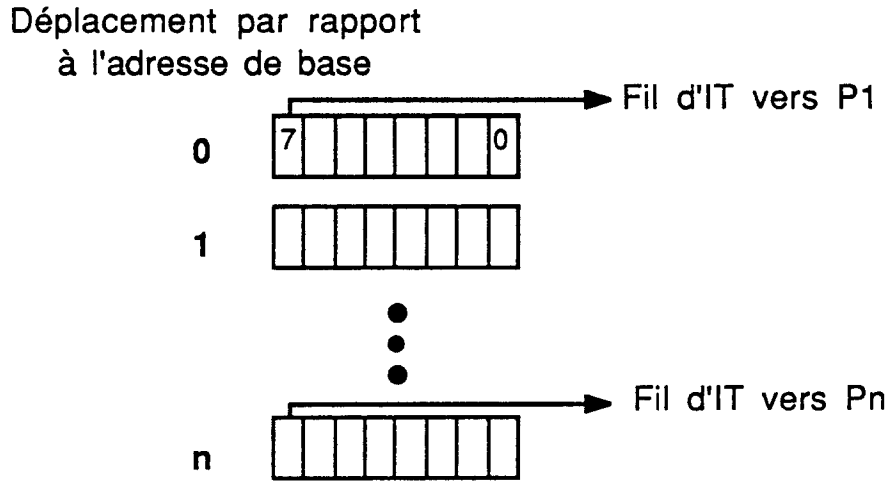


figure I.10

La carte d'IT est constituée d'un ensemble de 16 cellules mémoire (octets); à chaque cellule est associée un processeur. Pour générer une IT vers le ième processeur, il suffit de positionner au niveau logique 1 le bit de poids fort du ième octet de la carte. L'acquittement de l'IT est effectué par le processeur interrompu, en remettant ce bit au niveau logique 0. Par la suite, le signal d'interruption inter-processeurs sera dénommé IT système.

---

1 A l'heure actuelle, de nombreux fabricants de cartes processeurs, conscients de cette lacune, proposent des solutions d'interruptions sélectives par boîte aux lettres: chaque processeur possède une petite zone mémoire, accessible du bus, dans laquelle il suffit d'écrire ou de lire pour générer une IT au processeur. Une seule condition doit être remplie: chaque processeur

Le microprocesseur MC68000 possède 7 niveaux distincts d'interruption notés symboliquement IT1 à IT7. L'IT1 est la moins prioritaire et ainsi de suite, jusqu'à l'IT7 la plus prioritaire, qui est aussi non masquable. Afin de ne pas pénaliser un utilisateur qui désirerait implanter des processus rapides déclenchés par IT, il a été décidé d'attribuer le niveau 1 à l'IT système. Le système peut donc être considéré comme le processus le moins prioritaire sous IT.

Il est à noter qu'en fonctionnement multi-processeurs, le système est entièrement réparti; c'est à dire qu'aucun processeur n'est maître par rapport à un autre; tous les processeurs portent en mémoire morte le même noyau de l'exécutif temps réel.

I.2.1.3. Application à CESAR

Suivant les auteurs, les "agents actifs" d'une application temps réel sont appelés tâches ou processus. Par la suite, nous emploierons le mot "tâche" pour les désigner. Nous préférons, en effet, réserver le terme "processus" pour des sections de code participant à la mission du système, mais la plupart du temps invisibles à l'utilisateur (niveau système, "drivers" d'E/S, gestionnaire d'ITs, ...).

Sous CESAR qui utilise des microprocesseurs de la famille MC68000, il est possible de formaliser cette distinction grâce à la notion de mode de fonctionnement du processeur:

- Les tâches s'exécutent dans le mode "utilisateur" du microprocesseur.
- Les processus, eux, s'exécutent au niveau "superviseur" du microprocesseur, et ont donc accès au jeu complet d'instructions, incluant les opération du type masquage aux ITs, par exemple, interdites au niveau utilisateur.

I.2.2. Etats d'une tâche

L'ordonnanceur de CESAR est une machine à états. Chaque tâche CESAR au cours de son évolution peut passer par cinq états différents:

- L'état initial d'une tâche est l'état inexistant: dans cet état, le code exécutable de la tâche est en mémoire, mais la tâche n'est pas connue du système, aucun descripteur de tâche n'a été créé pour elle. L'opération de création a pour but de faire connaître la tâche au système qui lui alloue un descripteur; la tâche passe alors dans l'état activable. Toute tâche peut retourner à l'état inexistant par l'opération de destruction; cette opération peut être effectué à la demande de la tâche ou de toute autre tâche. C'est une opération dangereuse, qui est très généralement réservée au cas d'une tâche d'initialisation qui, une fois son rôle achevé, "se fait hara-kiri".
- L'état activable est un état où la tâche dispose de toutes les ressources nécessaires à son exécution à l'exception du processeur.
- Sur un processeur donné, la tâche active est la tâche activable de plus forte priorité. Dans l'état actif, la tâche est en cours d'exécution par le processeur. Une tâche quitte l'état actif soit parce qu'elle a émis au système une requête avec attente qui ne peut être immédiatement

autre tâche plus prioritaire est passée de l'état en attente à l'état activable.

- Une tâche passée à l'état "en attente" le reste tant qu'elle ne reçoit pas un événement.
- Dans l'état "pausé" une tâche ne participe pas à la mission du système; elle ne pourra passer à l'état activable ou même actif (même si elle n'est pas en attente et de la plus forte priorité) que grâce à une requête de "dépause" (qui ne peut donc provenir que d'une autre tâche). L'action de pause peut être effectuée par la tâche elle-même ou par une autre tâche.

La figure suivante représente les différents états par lesquels peut passer une tâche et les différentes transitions possibles.

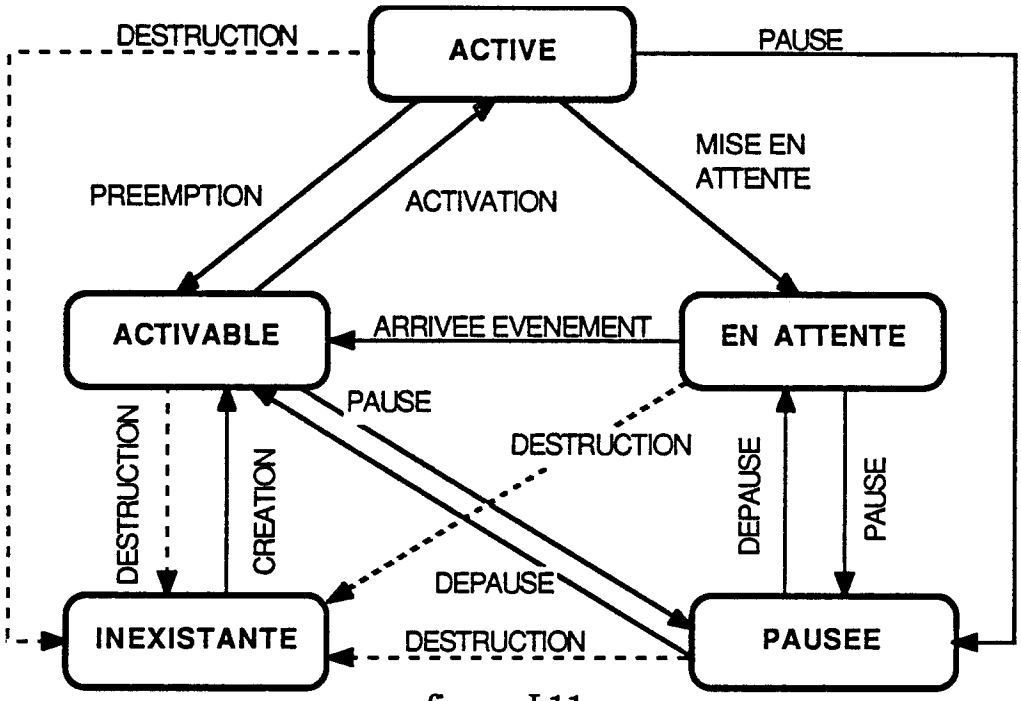


figure I.11

Ces différents états sont représentés de manière interne par un mot d'état appartenant au descripteur de tâche. Ce mot contient les champs suivants assimilables à des variables booléennes:

- Un octet indiquant suivant sa valeur si la tâche est pausée ou non.
- Un octet indiquant si la tâche est en attente d'événement.
- Un octet indiquant si la tâche a reçu un événement.

Au vu de ces structures nous pouvons énoncer des règles simples concernant le fonctionnement de l'ordonnanceur:

1) Chaque fois qu'il y a lieu de déterminer la nouvelle tâche activable, l'ordonnanceur scrute la liste des tâches à la recherche de la tâche de plus forte priorité, non en attente et non pausée.
2) Une tâche ayant déjà reçu un événement (octet *événement* VRAI) effectuant une requête avec attente verra ses deux octets *événement* et *attente* passer à la valeur logique FAUX; cette tâche reste donc

3) Lorsqu'une tâche en attente, reçoit un événement, l'octet en attente devient FAUX et la tâche passe à l'état activable.

I.2.3. Notion de requête

Dans le paragraphe I.2.2.3 nous avons vu qu'une tâche est rarement isolée: d'une part, elle a besoin de ressources pour mener à bien sa mission; d'autre part, elle peut être amenée à échanger des données ou des signaux de synchronisation avec d'autres tâches.

Toutes ces opérations se font par le biais de requêtes à l'exécutif temps réel CESAR. Dans la version actuelle du système (V1.xx) une requête est constituée d'un ensemble de 8 mots longs (soit 32 octets) contigus. Cet ensemble est dénommé **bloc requête** (abrégé B.R. par la suite).

Avant d'examiner la constitution d'un B.R., il nous paraît essentiel d'expliquer certains choix qui ont été effectués lors de la définition du noyau CESAR et qui déterminent certaines particularités du système:

- Sa rapidité est en partie due au fait que, lors de la transmission d'une requête au système, il n'y a pas recopie de cette dernière (dans un espace mémoire inaccessible à l'utilisateur, par ex.).
- Il est peu protégé (l'utilisateur peut par mégarde, ou intentionnellement modifier les tables système). L'absence de protection permet un gain de temps lors de la traversée du système.

D'où les observations suivantes:

- Une tâche peut très bien avoir accès aux différents champs d'une de ses requêtes et éventuellement les modifier alors que la requête en question est en cours de traitement par le système. Une telle action, peut évidemment avoir des conséquences inattendues voire même désastreuses.

Ce qui nous permet d'énoncer notre première règle de programmation sous CESAR:

> Lorsqu'une requête est en cours de traitement par le système (ou toute autre entité), elle n'appartient plus à sa tâche émettrice, qui ne doit en aucun cas en modifier le contenu (l'accès en lecture reste autorisé).

Comme toutes les règles dignes de ce nom, la règle précédente souffre une exception que nous verrons par la suite.

- Certaines requêtes sont exécutées uniquement par le processeur local; dans ces cas le B.R. peut être réservé en mémoire locale. Mais lorsqu'une tâche d'un processeur P_i s'adresse à une entité portée par un processeur P_j (avec $i \neq j$), il est nécessaire que le B.R. soit réservé dans une zone mémoire accessible aux deux processeurs, donc en mémoire globale. Dans ce cas, la requête sera d'abord adressée au système local qui la retransmettra au système du processeur destinataire.En résumé, quelle que soit sa destination, une requête est toujours adressée en premier lieu au système local; le déroutement éventuel vers un autre processeur est invisible de l'utilisateur. D'où la seconde règle:

L'utilisateur doit s'efforcer de conserver cette transparence, et ne doit pas se préoccuper de connaître l'implantation (le processeur propriétaire) d'une entité à laquelle il s'adresse.

De manière générale, il utilisera des B.R. implantés en mémoire globale lors de requêtes vers toute entité (tâche, ressource, ...).; cette entité pouvant à priori ne pas être implantée sur le processeur courant.

A la longue, la gestion, par l'utilisateur, des B.R. peut se révéler fastidieuse. Pour pallier cet inconvénient, l'exécutif CESAR comporte un gestionnaire de B.R. dénommé banque. Celle-ci est un ensemble de B.R. réservés en mémoire globale et fonctionnant comme une file: une tâche peut à tout moment demander un B.R.; la banque lui allouera le bloc en tête de la file. Lorsqu'une tâche n'a plus l'utilisation d'un B.R., elle peut le rendre à la banque, ce dernier sera alors placé en queue de la banque.

La banque est dimensionnée lors de l'initialisation du système. Il y a lieu de prévoir un "fond de roulement" suffisant de B.R., car s'il n'y a plus de B.R. disponible à un moment donné, la banque "saute": il y a crash[1] du système.

L'obtention d'un B.R. se fait par une "trappe" du MC68000: TRAP 8. Après exécution de cette instruction, le registre d'adresse A2 pointe sur l'adresse de base du bloc requête alloué. La restitution d'un B.R. à la banque se fait aussi en une seule instruction: TRAP 9; A2 doit pointer sur ce B.R. .

Après avoir décrit le fonctionnement de la banque, nous pouvons aborder le détail des différents champs d'un B.R..

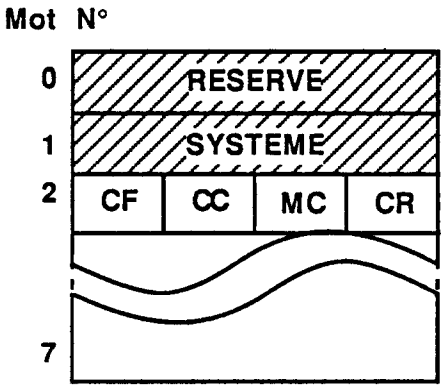


figure I.12

- Les deux premiers mots longs sont réservés au système, et nous ne nous en préoccuperons plus par la suite.
- Le mot numéro 2 est en fait décomposé en quatre octets:
- Le code fonction (CF) qui décrit la classe de service demandé au système (Ex: initialisation système, E/S, etc...).

---

[1] Lors d'un crash le processeur retourne en mode console (plus aucune tâche ne tourne, les timers sont gelés); en configuration multi-processeurs les autres processeurs passent aussi en

- Le code fonction complémentaire (CC); il précise le service demandé à l'intérieur d'une classe définie (Ex: pour la classe E/S: lecture, écriture, initialisation "driver").
- Le mot de commande (MC) qui indique au système ce qu'il convient de faire de la requête (Ex: mettre la tâche en attente de la complétion de cette requête, ou au contraire, ne pas attendre, ...). Nous détaillerons cet octet plus loin.
- Le code retour (CR): maintenu par le système, il indique l'état de la requête. Il contient une information de type ternaire:
  - Négatif: la requête est en cours de traitement par le système.
  - Positif: la requête s'est terminée suite à une erreur.
  - Nul: la requête s'est terminée sans erreur.

- Les mots 3 à 7 sont les paramètres d'entrée et de sortie de la fonction. La signification de ces mots varie suivant le service demandé (cf. [CES 84]).

Description du mot de commande (1 Octet):

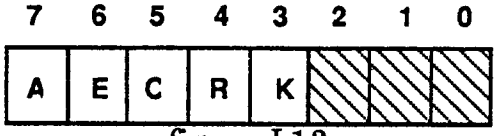


figure I.13

- Les bits 0 à 2 sont inutilisés et réservés pour des extensions futures.
- Le bit 3 (K) comme Krash:
  - à 1: En cas d'erreur lors du traitement de la requête (code retour > 0), le système entrera en crash. Cette option est très utile dans les phases de mise au point.
  - à 0: Pas de crash généré si CR > 0. Il appartiendra à l'utilisateur de tester la valeur du CR afin de diagnostiquer des erreurs éventuelles et de prendre les mesures appropriées.
- Le bit 4 (R) comme Restitution:
  - à 1: A la terminaison de la requête, le bloc sera automatiquement rendu à la banque. Permet d'économiser un TRAP 9.
  - à 0: La tâche reste propriétaire du bloc après exécution de la requête. Il importe de positionner ce bit à 0 si l'on attend des paramètres de retour.
- Le bit 5 (C) comme Chaînage:
  Utilisé uniquement pour les requêtes adressées à des ressources.
  - à 1: Si la ressource n'est pas disponible, la requête est chaînée dans la file d'attente de la ressource, et sera traitée dès que la ressource redeviendra disponible.
  - à 0: La requête ne sera traitée que si la ressource est immédiatement disponible; sinon, la requête est renvoyée avec un CR positif indiquant que la ressource n'est pas disponible.
- Le bit 6 (E) comme Evènement:
  - à 1: Lors de la complétion de la requête un événement sera généré à la

- Le bit 7 (A) comme Attente:

  à 1: La tâche est mise en attente de l'arrivée d'un événement.

  à 0: La tâche n'est pas mise en attente, elle continue en séquence.

La grande souplesse du système vient de la possibilité de positionner le mot de commande au gré de l'utilisateur. Ainsi, grâce aux bits 7 et 6 (les plus utilisés en pratique) il est possible d'envoyer au système des requêtes non bloquantes.

L'envoi d'une requête est effectué par l'instruction TRAP 4; le registre A2 doit alors pointer sur le B.R. à transmettre au système.

N.B.: Par la suite, nous adopterons la convention suivante pour décrire de façon concise le mot de commande utilisé dans les requêtes au système:

- Une lettre en minuscule (a, e, c, r ou k) indique que le bit associé est au niveau logique 0.
- Une lettre en majuscule (A, E, C, R ou K) indique que le bit associé est au niveau logique 1.
- Si une lettre est absente, cela signifie que la valeur logique du bit associé est indifférente.

Remarque:

Nous avons vu au paragraphe I.2.2 concernant les différents états d'une tâche, qu'à chaque tâche sont associés deux drapeaux: l'un indiquant si la tâche est en attente d'événement; l'autre, si elle en a reçu un. Considérons maintenant la séquence d'instructions suivante:

```
    Actions 1
    ...
(1) Requête 1 au système sans attente, avec génération d'événement (aE)
    ...
    Actions 2
        ...
(2) Requête 2 au système avec attente et génération d'événement (AE)
    ...
    Actions 3
```

figure I.14

Une telle suite de code ne garantira pas que l'on attendra la fin de la requête 2 avant d'enchaîner sur l'exécution des actions 3; ce qui est en général le comportement désiré.

En effet, si lors de l'exécution de la requête N°2, la requête N°1 est terminée, la tâche recevra un événement et l'attente de la fin de la requête N°2 n'aura pas lieu.

Deux conclusions s'imposent:

- Si l'on est sûr qu'aucune requête non bloquante n'est en cours, il est possible d'envoyer une nouvelle requête bloquante; auquel cas

l'exécution de la tâche reprendra après la terminaison de la dernière requête.

- Dans le cas contraire, il faut s'assurer, par un examen du code retour de la requête, qu'elle est bien terminée et que l'événement ne provient pas d'une autre requête en cours.

Modifions l'exemple précédent comme suit:

```
     Actions 1
     ...
(1)  Requête 1 (aE)
     ...
     Actions 2
     ...
(2)  Requête 2 (aE)

     TANTQUE (Requête 2 en cours) FAIRE
          Fonction NOP (Ae)
     FIN TANTQUE

     Actions 3
```

figure I.15

La fonction **NOP** est une fonction système qui n'effectue strictement aucune action. Sa seule utilité est qu'elle permet de positionner le mot de commande, en particulier pour effectuer une simple mise en attente d'événement. Dans l'exemple ainsi corrigé, il est maintenant garanti que les actions 3 ne seront exécutées qu'**après** terminaison de la requête N°2.

Dans des cas de figure plus compliqués, de telles manipulations deviennent rapidement fastidieuses. Les logiciels que nous avons développés et qui sont décrits dans le Chapitre II apportent une grande amélioration dans la souplesse d'utilisation de toutes les fonctionnalités de CESAR.

I.2.4. Les différentes ressources de CESAR

Il est possible de séparer les ressources de CESAR en deux grandes catégories:

- Les ressources de synchronisation, qui permettent aux tâches de spécifier leurs conditions d'activation.
- Les ressources de données qui permettent les communications (ou échanges de données) entre les tâches et le monde extérieur ou les communications inter-tâches. Notons que ce type de ressource permet d'implanter quelques mécanismes simples de synchronisation du type "producteur-consommateur".

I.2.4.1. Les ressources de synchronisation

I.2.4.1.1 Ressources de cadencement

- Timers:

Nombreuses sont les applications pour lesquelles il est nécesaire de disposer d'une base de temps précise: scrutation périodique de capteurs, asservissement en vitesse de dispositifs mécaniques, etc ... . A cet effet, le système CESAR dispose de ressources de type **timer** pouvant fonctionner suivant deux modes différents:

- Monocadencement:

  La tâche spécifie dans sa requête au timer une heure (ou un délai). Lorsque l'heure est atteinte (ou le délai expiré) un événement sera généré à la tâche.

- Multicadencement:

  Dans ce mode, le timer génère des événements à la tâche avec une cadence fixe. Il est en outre possible, lors de la transmission de la requête au timer, de préciser une heure (ou un délai) correspondant à l'arrivée du premier événement.

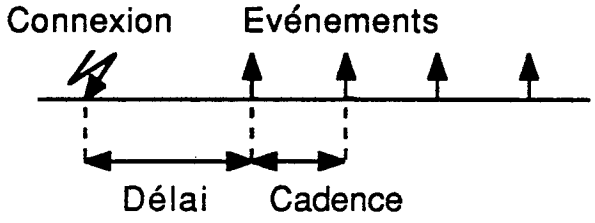


figure I.16

L'utilisation du mode multi-cadencement se fait classiquement de la manière suivante:

```
Req1: Connexion Timer Multicadencement (aCE)
BOUCLE CADENCEE
    Req2: Fonction NOP (Ae)
    ...
    Actions
FIN BOUCLE
```

figure I.17

La requête Req2 sert de point de synchronisation pour attendre les différents événements timer. Il reste cependant un petit problème à régler; considérons, en effet, le cas suivant:

La tâche T_1 est de priorité P_1

La tâche T_2 est cadencée par un timer, et de priorité $P_2 < P_1$

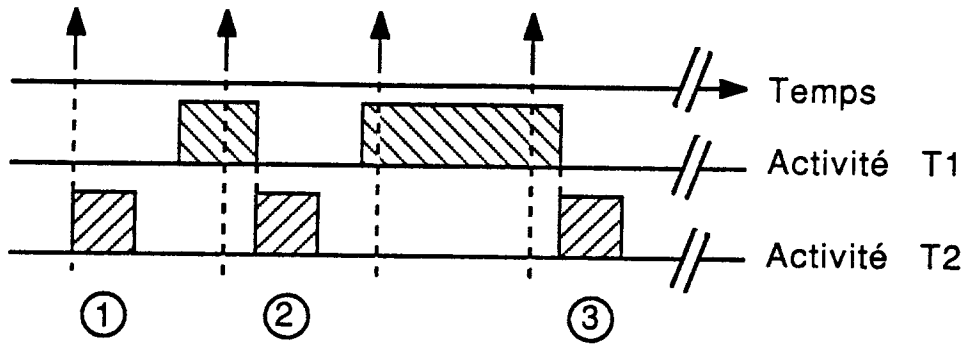


figure I.18

Si l'on veut que la tâche T_2 soit activée à chaque événement timer, il est clair que dans le cas numéro 3, on manquera un événement si l'on ne prend pas de précautions particulières.

Une solution à ce petit problème consiste à utiliser deux compteurs: l'un géré par le système, l'autre par T_2. D'une part le système réserve un champ du bloc requête de connexion timer comme compteur d'événements système: à chaque fois qu'un événement est généré par le timer ce compteur est incrémenté. D'autre part, la tâche doit maintenir à jour un "compteur personnel" contenant le nombre d'événements timer effectivement traités.

L'exemple décrit précédemment doit donc être modifié de la façon suivante:

```
Req1: Connexion Timer Multicadencement (aCE)
...
BOUCLE CADENCEE
    SI (CompteurPersonnel = Compteur Système) ALORS
        Req2: Fonction NOP (Ae)
    FINSI
    incrémenter CompteurPersonnel
    ...
    Actions
    ...
FIN BOUCLE
```

figure I.19

Cette manière de faire est la seule qui garantisse le traitement de tous les événements reçus.

Cependant, en pratique, on désire la plupart du temps garantir la constante periodicité des activations d'une tâche cadencée: c'est-à dire-qu'il faut rendre constant et si possible minimiser le temps de latence entre l'arrivée de l'événement et l'activation effective de la tâche. Le temps de traversée du système étant en première approximation constant, la seule chose qui puisse rendre non constant cette période de latence est la présence de tâches actives de priorité supérieure à la tâche cadencée (cas numéro 2 fig I.18); donc, il est nécessaire d'attribuer aux tâches cadencées les priorités les plus élevées. Ainsi, il est impossible de manquer un événement et la construction de la fig I.17 suffit largement.

Masquage des événements et déconnexion du timer

Rappelons qu'après son envoi au système, une requête de connexion en mode mutli-cadencement reste chaînée à la ressource timer en permanence; à l'opposé, une requête de connexion en mode mono-cadencement reste chaînée à la ressource timer jusqu'à la complétion de la requête (délai expiré ou heure atteine).

Si le besoin s'en fait sentir, il est possible de "geler" le timer pendant un laps de temps: dans ce cas, le timer continuera de tourner mais ne générera plus d'événements vers la tâche. Pour cela, il suffit de mettre à zéro le bit "Génération d'événements" du mot de commande de la requête de connexion au timer.

De même, s'il faut se déconnecter d'un timer la manipulation est identique dans son principe:

1 Mettre à zéro le bit de génération d'événement de la requête de connexion de façon à ne plus recevoir d'événements.

2 Transformer la requête de multi-cadencement en requête de mono-cadencement; ainsi, une fois satisfaite, la requête de mono-cadencement sera déchaînée de la file d'attente du timer.

Ces deux exemples sont les exceptions à la règle énoncée au paragraphe I.2.3 qui conseille de ne pas modifier une requête dès qu'elle est sous contrôle du système.

I.2.4.1.2 Ressources IT Soft

Ce type de ressources est aussi appelée signal dans la terminologie des systèmes temps réel. Sous CESAR, l'utilisation des ressources IT Soft est analogue à celle des ressources timer; la seule différence réside dans le fait que les événements ne sont plus générés à une cadence fixe par un timer, mais par une ou plusieurs autres tâches.

Exemple: Soit la synchronisation suivante:

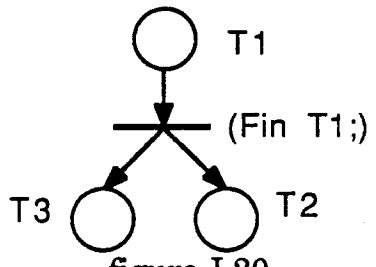


figure I.20

Lorsque la tâche T_1 est terminée les tâches T_2 et T_3 sont exécutées en paralléle. Une telle synchronisation peut être implantée grâce au code suivant:

Tâche T_1	Tâche T_2 (ou T_3)
	Connexion ressource ITSoft R_{IT} (ACE)
Actions1	Actions2 (ou Actions3)
Génération ITSoft sur R_{IT}	
Fin T_1	Fin T_2 (ou T_3)

figure I.21

Lorsque T_1 générera l'IT Soft à la ressource R, les tâches connectées à cette ressource (T_2 et T_3 ici) recevront un événement.

La connexion à une ressource IT Soft se fait exactement de la même manière que pour une ressource timer (mode mono-cadencement et multi-cadencement, existence d'un compteur d'événements, ...). Seules les notions de délai (T) et de cadence (cad.) n'ont évidemment plus aucune signification.

I.2.4.1.3 File d'ordre

Il arrive fréquemment que l'on doive assurer des synchronisations du type appel-réponse, compte-rendu: une tâche T_1 sous-traite une partie de son travail à une tâche T_2, pendant l'exécution de cette dernière, T_1 enchaîne une série d'actions avant d'attendre la fin du travail sous-traité à T_2.

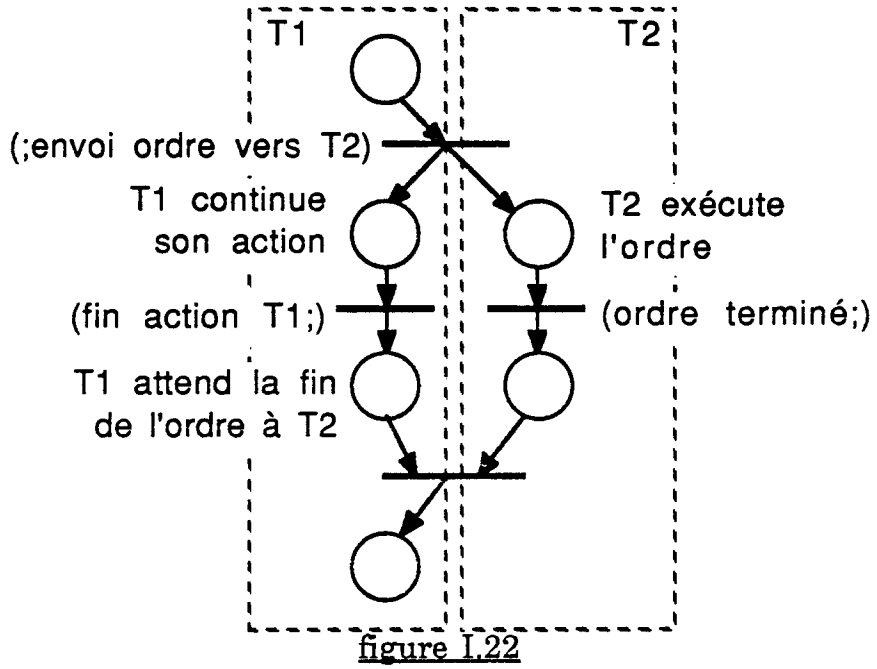


figure I.22

Il est possible d'implanter une telle synchronisation en utilisant des ressources IT Soft.

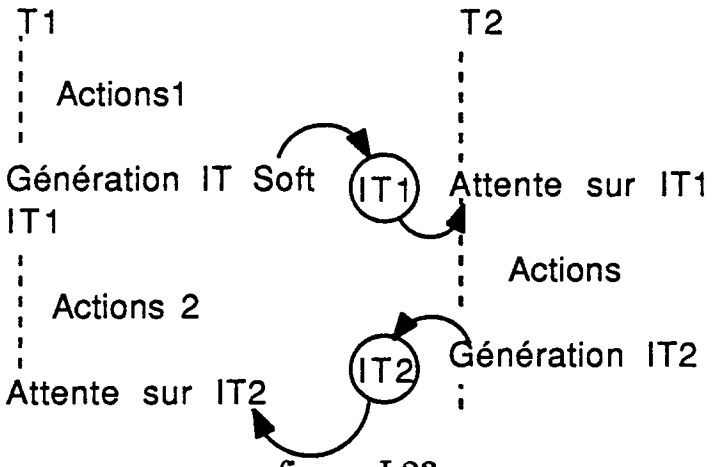


figure I.23

S'il y a en plus, échange de données entre les deux tâches (Ex: paramètres d'entrée indiquant le travail à effectuer, compte rendu de fin d'exécution, ...) il est possible de remplacer les ressources IT Soft par des files d'échange.

Il est donc clair qu'à chaque fois qu'il faudra implanter une synchronisation du type appel-réponse, compte-rendu, cette solution nécessite l'utilisation de deux ressources. Cependant, CESAR permet d'effectuer ce type de synchronisation grâce à une seule ressource: une file d'ordre. Ce type de service ne se trouve que sur quelques exécutifs temps réel, en général assez complexes, sous le nom d'ASQ (pour **A**synchronous **S**ervice **Q**ueue); la tâche gérant l'ASQ (T2 ici) étant dénommée ASR (pour **A**synchronous **S**ervice **R**outine).

Reprenons l'exemple précédent en utilisant une file d'ordre, puis explicitons-en le fonctionnement.

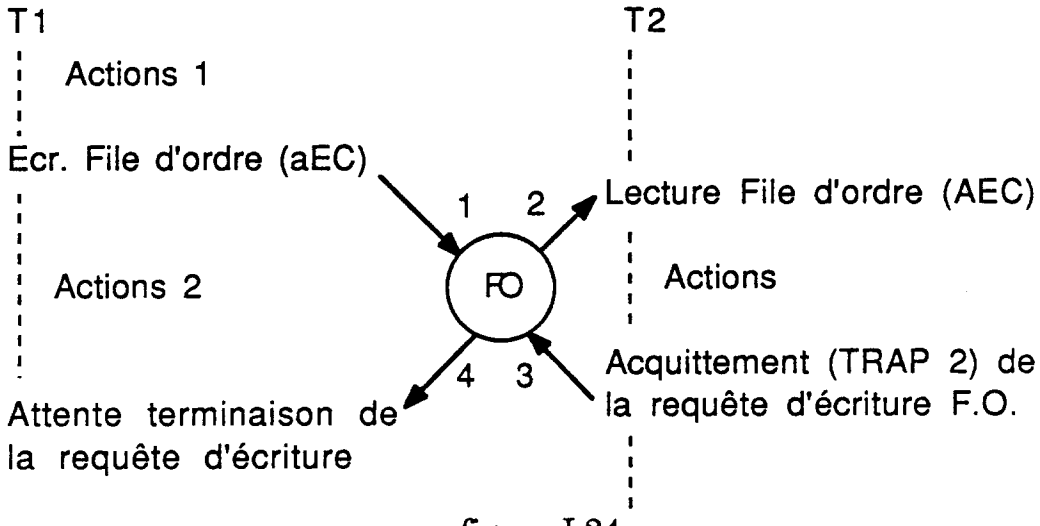


figure I.24

Le format d'une requête d'écriture sur une file d'ordre est le suivant:

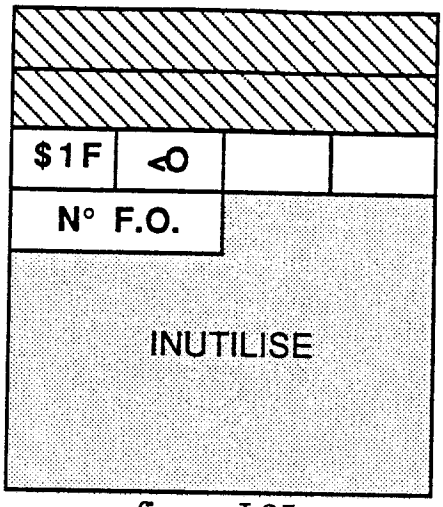


figure I.25

Celui d'une lecture par contre est:

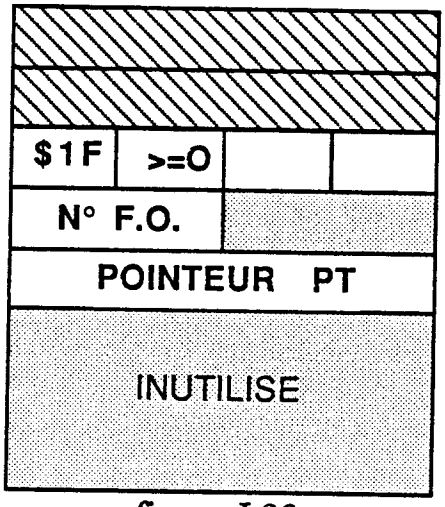


figure I.26

Lorsqu'une requête d'écriture est envoyée sur une file d'ordre, elle est chaînée jusqu'à l'arrivée d'une requête de lecture (comme pour les files d'échange), à ce moment la requête d'écriture est déchaînée et le champ PT de la requête de lecture pointe sur la requête d'écriture correspondante

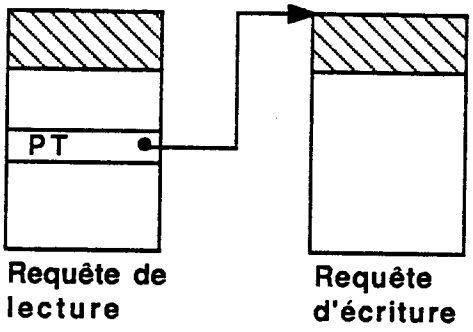


figure I.27

Par le biais de ce pointeur, la tâche lectrice a accès au bloc requête d'ecriture et en particulier aux informations que peut contenir la zone

Aprés traitement de l'ordre ainsi reçu, la tâche destinatrice doit acquitter le bloc requête d'écriture; cette opération se fait grâce à une "trappe" du système: TRAP 2.

Interface d'entrée:

- A2 pointe sur le bloc requête écriture à acquitter.
- D0.B contient un code retour:
  - Nul: traitement normal.
  - Positif: Une erreur a été détectée.

Lors de l'acquittement, la requête chemine au travers des exécuteurs de sortie du système: si la tâche émettrice avait demandé une génération d'événement, un événement lui sera envoyé, et dans le cas d'un code retour (D0.B) positif il y aura mise en crash si le bit K du mot de commande était positionné.

N.B.: Il est possible de considérer que dans ce cas, la tâche lectrice joue le rôle du système. Il est ainsi tout à fait possible de bâtir des "sur-systèmes" où la file d'ordre joue le rôle de canal d'Entrée/Sortie du sur-ensemble ainsi construit.

I.2.4.2. Les ressources de données

I.2.4.2.1 Les ressources d'espace mémoire

Une telle ressource est, lors de sa création, constituée d'un ensemble de N blocs contigus de taille m octets. Ces ressources sont manipulées par deux fonctions systèmes:

- La demande d'un bloc mémoire, qui alloue le premier bloc en tête de la ressource.
- La restitution d'un bloc; aprés restitution, le bloc est chaîné en queue de la ressource (qui fonctionne donc à la manière d'une file).

Ce type de ressource peut être utilisé de deux manières:

- La ressource est seule. Elle sert alors à la création et à la destruction de variables dynamiques de taille maximale m octets.
- La ressource est associée à une ressource d'Entrée/Sortie ou à une file d'échange; elle sert alors de réserve de "buffers" pour la transmission des informations.

I.2.4.2.2 Les files d'échange

On désigne ainsi sous CESAR ce qui est généralement appelé "boîtes aux lettres" sous d'autres exécutifs temps réel ou encore "pipe" sous Unix.

Une boîte aux lettres peut fonctionner suivant deux modes définis par CESAR:

-Mode standard:

Dans ce cas, le proccessus écrivain doit fournir, l'adresse de la zone mémoire dans laquelle il a écrit son message. Les limitations de ce mode

- Comme l'implantation du destinataire du message n'est, à priori, pas connue (le processus écrivain ne connait que le numéro de la ressource dans laquelle il va écrire), le bloc mémoire servant de support au message doit être réservé en mémoire globale.
- Dans le cas où le processus écrivain continue à produire un flot de données sans attendre que le processus lecteur ne "consomme" les messages à l'autre bout de la boîte aux lettres, il est nécessaire qu'il gére un stock de "buffers". Le (ou les) processus destinataire(s) devra(ont) libérer ces buffers après usage et en avertir de ce fait le processus producteur. La gestion d'un tel stock, dans un contexte multiprocesseur, impose des manipulations classiques de sémaphores (instruction Test And Set, masquage aux interruptions, ...) dont certaines sont privilégiées donc inaccessibles au niveau tâche.

-Mode flot:

Ce mode permet de pallier les inconvénient cités ci-dessus. A cet effet, il faut, avant la création de la ressource file d'échange, créer une ressource espace mémoire contenant n blocs et qui servira de "réserve de buffers". Lors de toutes les opérations de gestion mémoire, l'utilisateur n'a à connaitre que le numéro de la boîte aux lettres et pas celui de la ressource espace mémoire associée:

Avant toute écriture, il est nécessaire de demander un bloc mémoire à la file d'échange; le système renvoie alors l'adresse et la taille du premier bloc libre de la ressource espace mémoire; le bloc ainsi attribué est disponible en écriture pour le processus producteur. Le processus lecteur, une fois qu'il a pris connaissance du contenu du message, doit rendre le bloc contenant le message à la file d'échange (le système se chargera alors de le mettre en queue de la ressource espace mémoire associée) afin d'éviter de bloquer le système, à la longue, par pénurie de buffers.

Le réseau de Pétri de la figure suivante permet de modéliser le fonctionnement d'un ensemble de processus producteur et consommateur dialoguant par le biais d'une file d'échange à laquelle est associée une ressource espace mémoire de N blocs.

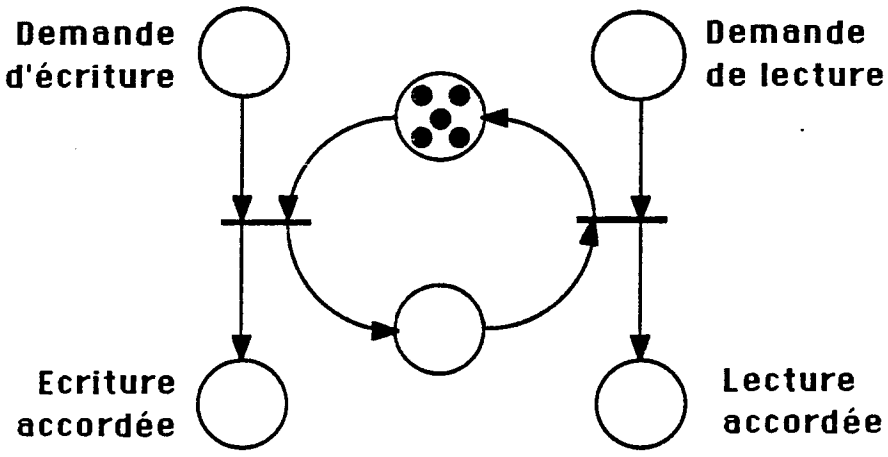


figure I.28

Ici N=5

I.2.4.2.3 Les ressources d'Entrée/Sortie

De la même façon que les ressources de type file d'échange permettent la communication de données entre deux tâches, les ressources d'entrée/sortie permettent la communication de données entre une tâche et le monde extérieur (entrée de valeurs au clavier, affichage de messages, d'alertes, ...).

Du point de vue de l'utilisateur, le fonctionnement d'une ressource d'entrée/sortie est strictement équivalent à celui d'une file d'échange (les fonctions systèmes sont exactement les mêmes): en particulier les deux modes standard et flot existent.

La seule différence dans le cas du mode flot, est qu'après écriture, le buffer est automatiquement rendu à la ressource espace mémoire associée par le systéme; rappelons que dans le cas d'un échange par boîte aux lettres, ce rôle est dévolu au processus destinataire. De même, avant lecture sur une ressource d'entrée, le système alloue automatiquement un buffer.

Cette similitude de fonctionnement entre les ressources file d'échange et entrée/sortie est très intéressante pour la mise au point des applications; en effet, considérons le cas suivant: soient deux processus P_1 et P_2 s'échangeant des données par le biais d'une boîte aux lettres, avec P_1 producteur et P_2 consommateur.

Dans un premier temps, il est possible de valider le fonctionnement du processus P_1 en remplaçant, lors de la configuration de l'application, la file d'échange par une ressource d'E/S reliée à une console (ou à une imprimante) et en analysant visuellement les données ainsi produites; de même le fonctionnement de P_2 peut être validé en fournissant, grâce à un clavier, un jeu de données d'entrée cohérent, et en observant les réactions du processus. Grâce à cette facilité, l'utilisateur, lors de la phase de mise au point, n'a pas à modifier et à recompiler le code de ses processus P_1 et P_2; seul le fichier de configuration est à modifier.

CHAPITRE II

SYNCHRONISATION DE REQUETES ET EXTENSION DE LA NOTION DE TACHE

II. SYNCHRONISATION DE REQUETES ET EXTENSION DE LA NOTION DE TACHE

II.1 Introduction

Nous avons vu au chapitre précedent que la mise en œuvre d'une application sur un système temps réel soulevait de nombreux problèmes inconnus sur une machine informatique conventionnelle. Ce phénoméne se manifeste à plus forte raison dans toute machine à structure parallèle.

Il existe de nombreuses méthodes de spécification et de validation des logiciels. Elles sont en général basées sur une analyse descendante du cahier des charges aboutissant en la décomposition de l'application en modules fonctionnels, eux-mêmes subdivisables, etc... . La validation de l'application se faisant en sens inverse: test unitaire des composants de base des modules fonctionnels, puis test des modules fonctionnels et enfin intégration et test de l'ensemble de l'application. Ces méthodes sont cependant surtout destinées aux logiciels conventionnels, et fort peu proposent des solutions réellement adaptés aux problèmes spécifiques des machines temps-réel.

En effet, il arrive assez fréquemment dans le cycle de vie d'une application temps réel, qu'un utilisateur disposant de divers modules logiciels veuille en organiser ou en modifier la synchronisation. La première approche de ce problème consiste à incorporer les appels à l'exécutif temps réel à l'intérieur du code de ces différents modules. Dans ce cas toute modification de l'enchaînement temporel des modules nécessitera leur recompilation, d'où perte de temps.

De plus, du point de vue méthodologique, il est impossible de valider complétement un module séparément des autres, en raison de la présence de ces ordres de synchronisation.

En s'appuyant sur la théorie des réseaux de Petri et sur la décomposition classique en deux parties d'un système de commande, il nous a donc semblé nécessaire d'ajouter à un noyau temps réel un "sur-système" permettant de séparer les modules en deux catégories:

- D'une part, des modules purement opératoires (représentant en quelque sorte les procédures associées aux places ou aux franchissements de transitions dans un Réseau de Petri étiqueté [COUR 80]).
- D'autre part, des modules assurant la synchronisation de la partie opérative (correspondant au graphe de contrôle d'un Réseau de Petri).

Cette approche permet de n'avoir à modifier que les seuls modules de synchronisation dans le cas d'une redéfinition de l'enchaînement temporel des différents modules opératoires de l'application.

En outre, il est nécessaire que les méthodes d'analyse descendante et de validation ascendante d'un logiciel puissent aussi être appliquées aux modules de synchronisation.

Quel que soit l'exécutif temps réel choisi, il est possible de définir un squelette de module de synchronisation: un tel module comprend systèmatiquement une suite généralement non linéaire d'activations de modules de plus bas niveau et d'attente de compte-rendus d'exécution. Toutes ces actions sont réalisées par le biais de requêtes à l'exécutif temps réel.

On trouvera ainsi fréquemment l'activation de plusieurs processus en parallèle effectuée au moyen de requêtes non bloquantes, puis plus loin dans le code la mise en attente de la fin de m requêtes parmi les n lancées. La programmation d'une telle suite d'appels à l'exécutif est en général fastidieuse et très souvent particulière au type de synchronisation désiré, donc difficilement réutilisable dans le cas d'autres synchronisations.

Afin de pallier les inconvénients cités précédemment, nous avons défini les bases d'un utilitaire permettant la mise en attente sur une combinaison logiques de fin de requêtes: le Compilateur d'expressions logiques. Cet outil a été voulu indépendant du concept de tâche généralisée de manière à être utilisable en dehors de tout environnement particulier.

II.2 Le Compilateur d'expressions logiques

II.2.1 Exemple pratique

Considérons l'exemple suivant, classique en robotique d'assemblage.

Soit à réaliser l'insertion d'une pige dans un trou:

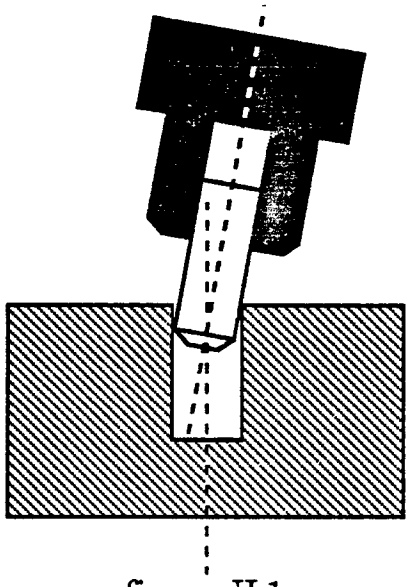


figure II.1

La primitive d'insertion est capable (grâce à des dispositifs mécaniques compliants, ou grâce à une commande hybride) de corriger des écarts de positionnement relativement peu importants.

On désire abandonner l'insertion soit sur détection d'un effort d'insertion trop grand (notion de garde en effort), soit lors de l'expiration d'un délai donné.

Nous supposerons que, pour réaliser cette application, l'implantation a été

- La primitive d'insertion est une tâche à laquelle est associée une file d'ordre FO_1. L'activation de la primitive se fait par écriture sur ce canal. Lorsque l'insertion est terminée, la requête d'écriture est alors acquittée. A chaque passage dans la boucle de calcul principale, une variable logique StopInsertion, initialement fausse, est testée afin de savoir s'il y a lieu d'arrêter l'insertion; dans ce cas la requête d'écriture sur FO_1 est aussi acquittée.
- La garde en effort est réalisée par une tâche cadencée, activée de la même manière que ci-dessus par une écriture dans une file d'ordre C_2. Si l'effort mesuré devient supérieur à l'effort maximum spécifié, la requête d'écriture sur FO_2 est immédiatement acquittée. De même que ci-dessus une variable logique StopGarde est testée à chaque réveil timer.

N.B.: Nous avons ici choisi le mode de synchronisation par file d'ordre pour simplifier l'exemple (une seule ressource est nécessaire pour assurer la synchronisation "activation - compte rendu"). Sous un autre exécutif, il aurait été nécessaire d'associer à chaque tâche un signal d'entrée à lever pour déclencher l'exécution de la tâche et un signal de sortie que la tâche lève lorsqu'elle a rencontré ses conditions de terminaison (cf. § I.2.4.1.3).

Le code nécessaire pour le déclenchement de ces actions est le suivant:

```
StopInsertion = FAUX
StopGarde = FAUX

Req1: Ecriture sur FO1 sans attente (aE)
Req2: Ecriture sur FO2 sans attente (aE)
Req3: Timer: Génération d'un événement (E) au bout de délai
Mise en Attente d'un événement quelconque
Scrutation des requêtes afin de déterminer laquelle (ou lesquelles) a (ont)
provoqué l'événement.

Si Requête1 Alors
     /* Fin normale, Insertion réussie */
FinSi
Si Requête2 ou Requête3 Alors
     StopInsertion = VRAI
     /* Echec à l'insertion */
FinSi
StopGarde = VRAI     /* N'a aucun effet si la garde est déjà terminée */
```

figure II.2

Même si la description de cet algorithme en pseudo-code peut sembler assez simple, son implantation constitue un bon exercice de programmation.

Dans cet exemple, seulement trois requêtes sont en cours simultanément, et les conditions logiques conditionnant les différents traitements sont triviales. Il suffirait que la complexité des équations logiques à surveiller augmente pour que le travail de programmation devienne rapidement pénible et difficile à modifier.

Afin de décharger le programmeur de ce travail, nous avons intégré une étape supplémentaire à la chaîne classique de production de logiciel. Cette étape consiste en la "compilation" des expressions logiques de requêtes.

Bien que le compilateur d'expressions logique ait été intégré à la chaîne de développement CESAR, ses principes de fonctionnement peuvent être étendus à tout exécutif temps réel à requêtes non bloquantes.

II.2.2 Intégration du compilateur d'expressions logiques à la chaîne de développement

Lors de l'intégration du compilateur d'expressions logiques deux critères ont prévalu:

- L'intégration de ce nouveau maillon à la chaîne de développement devait être la plus simple possible.
- Minimum de perturbations pour l'utilisateur.

Aussi nous avons placé de compilateur d'expressions logiques comme **pré-processeur** de la chaîne de développement: le code source du compilateur d'expressions logiques est un sur-ensemble du code source du premier maillon de la chaîne; seules les extensions spécifiques au compilateur d'expressions logiques seront reconnues par lui et expansées en un code assimilable par le premier maillon (en général assembleur ou compilateur). Deux solutions sont alors possibles: générer des appels à des fonctions contenues dans une bibliothèque spécifique: il faudra alors en faire l'édition de lien au moment du chargement de l'application; ou, générer en ligne un code assurant la synchronisation désirée.

Reprenons l'exemple précédent et écrivons-le en "pseudo-assembleur" en utilisant les possibilités que nous offre le "compilateur d'expressions logiques".

* Partie données

Réservation de trois zones pour mémoriser l'adresse des descripteurs de requête REQ1, REQ2 et REQ3.

* Partie code

Mémorisation de l'adresse du descripteur de la Requête REQ1
Envoi au système de REQ1.

.

Mémorisation de l'adresse du descripteur de la Requête REQ2
Envoi au système de REQ2

.

Mémorisation de l'adresse du descripteur de la Requête REQ3
Envoi au système de REQ3.

.

```
*$WAIT REQ1, REQ2, REQ3
*$IF REQ1 THEN OK
*$IF REQ2+REQ3 THEN ECHEC
```

.

.

ECHEC: Code de traitement en cas d'échec

Dans cet exemple, les ordres destinés au compilateur apparaissent sous forme de lignes commentaires pour l'assembleur (ligne comprenant un caractère * en première colonne) suivie d'un signe reconnu par le "compilateur d'expressions logiques" (le caractère $).

Pour pouvoir utiliser les fonctions du "compilateur d'expressions logiques", il est nécessaire de réserver, pour chaque requête traitable par le compilateur d'expressions logiques une zone mémoire qui contiendra l'adresse du descripteur de requête (partie donnée de l'exemple ci-dessus). L'ordre **$WAIT** permet d'effectuer une mise en attente jusqu'à ce qu'au moins une des requêtes passée en argument de **$WAIT** soit terminée (dans l'exemple précédent, tout événement provenant d'une requête autre que REQ1 à REQ3 sera ignoré). Les différents ordres **$IF** permettent de spécifier les expressions logiques conditionnant l'exécution de certaines parties du programme; tant qu'aucune des ces expressions n'est vraie, la tâche reste en attente.

N.B.: Afin de faciliter l'analyse lexicographique des programmes, le ET et le OU logique sont représentés dans les expressions logiques par leurs équivalents mathématiques . et + .

II.2.3 Structures et fonctionnement du "compilateur d'expressions logiques"

II.2.3.1 Préliminaire

Afin de définir des structures permettant une évaluation plus performante des expressions logiques nous avons dégagé une propriété fondamentale: à toute requête à l'exécutif temps réel utilisé, il est possible d'associer une variable logique, qui est fausse tant que la requête n'a pas été émise ou est en cours de traitement par le système, et vraie une fois que la requête a été honorée. Ces variables logiques associées à des requêtes constituent une classe bien particulière de l'ensemble des variables logiques: ce sont des variables qui sont initialement fausses, puis qui, à un moment donné, deviennent vraies et le restent.

II.2.3.2 Généralisation

L' exemple précédent correspond à une simple utilisation des ordres du "compilateur d'expressions logiques". De façon plus générale, un programme utilisant les fonctions du "compilateur d'expressions logiques" peut comprendre plusieurs ordres **$WAIT** se comportant comme autant de points de synchronisation; chaque ordre **$WAIT** étant suivi d'un ou plusieurs ordres **$IF**. Les expressions logiques associées aux ordres **$IF** sont constitués d'une suite d'identificateurs de variables séparés par des opérateurs logiques + et . éventuellement groupés dans des blocs parenthésés.

Exemple: $IF (A+B).C THEN ...

Afin de pouvoir analyser et compiler le programme en une seule passe,

- Tous les ordres **$IF** relatifs à un même ordre **$WAIT** doivent être contigus et se trouver immédiatement après cet ordre **$WAIT**. En conséquence, un ordre **$WAIT** ou toute autre ligne autre qu'un ordre **$IF** termine le bloc **$WAIT-$IF** précédent.
- Si deux expressions appartenant à un même bloc **$WAIT-$IF** sont vraies simultanément, la plus proche de l'ordre **$WAIT** est la plus prioritaire.

Exemple: Soit le bloc $WAIT-$IF suivant:

```
$WAIT a, b, c, d
$IF a.b+c THEN Etiq1
$IF a.d THEN Etiq2
$IF c.d THEN Etiq3
```

Si *c* devient vraie alors que *d* l'est déjà et qu'*a* et *b* sont fausses, le programme se déroutera vers l'étiquette Etiq1: en effet, *a.b+c* et *c.d* sont simultanément vraies, mais la première de ces deux expressions est plus prioritaire d'après la règle ci-dessus.

II.2.3.3 Structures utilisées

II.2.3.3.1 Structures associées à l'ordre $IF

De manière à ne pas réévaluer systèmatiquement toutes les expressions logiques chaque fois qu'un événement survient suite à une terminaison de requête, nous avons défini des structures permettant d'effectuer le minimun de calculs à chaque arrivée d'événement.

Tout d'abord, chaque expression logique est traduite sous forme d'un arbre binaire en respectant les règles de précédence classiques (. est plus prioritaire que +):

Exemple: IF (a+b).(c.d) THEN ...
donne:

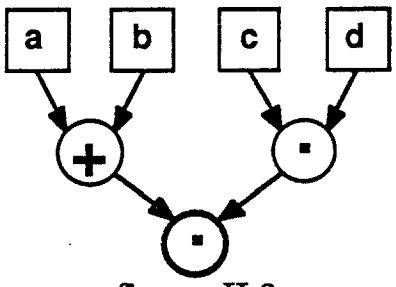


figure II.3

où les feuilles de l'arbre sont les requêtes que nous nommerons aussi "variables-requêtes" ou tout simplement variables. Chaque nœud est composé d'un opérateur logique liant entre eux deux autres nœuds, ou une variable et un nœud ou deux variables. Un nœud prend la valeur logique VRAI ou FAUX suivant la valeur des entités auxquelles il est lié. La véracité de l'expression

De plus, une même variable peut apparaître dans plusieurs expressions logiques comme dans:

```
$WAIT  a, b, c, d
$IF a.b THEN Etiq1
$IF b+c+d THEN Etiq2
$IF a.d THEN Etiq3
```

Ce qui peut se traduire en représentation arborescente par "une même feuille peut appartenir à plusieurs arbres !!". L'exemple précédent peut donc se représenter comme suit:

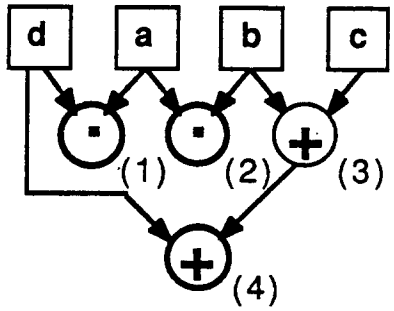


figure II.4

Les structures précédentes peuvent être implantées de manière classique en associant à chaque variable une liste de nœuds auxquels elle est liée; de même un nœud peut être considéré comme une variable structurée (**record** en **Pascal**, ou **struct** en **C**) composée de deux champs:

- une variable logique indiquant la valeur logique du nœud.
- une zone "pointant" sur le nœud suivant dans l'arbre, ou pour un nœeud terminal vers l'adresse ou se branchera le programme.

L'évaluation d'un nœud (et par là même de l'expression logique complète) se faisant grâce à un algorithme d'exploration et de marquage de l'arbre.

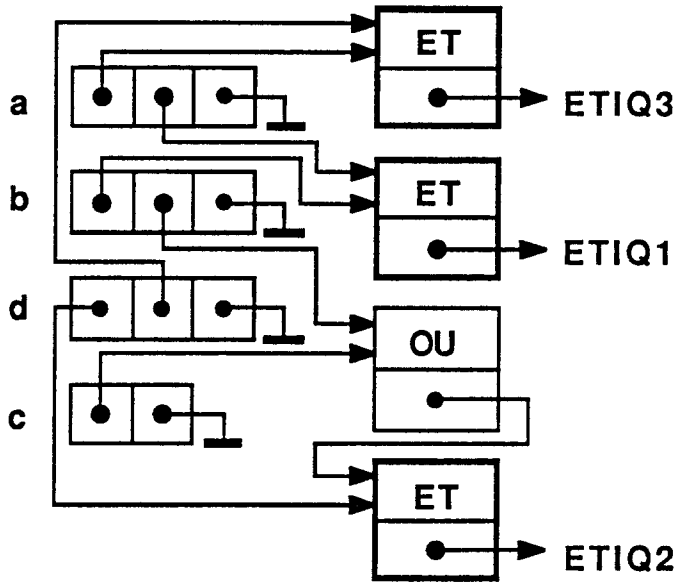


figure II.5

La figure II.5 représente la structure compléte composée pour chaque variable de la liste des nœuds auxquels elle est liée et du chaînage des nœuds

II.2.3.3.2 Principe de fonctionnement des nœuds

Considérons un nœud quelconque n_i d'un arbre; ce nœud relie deux sous-arbres:

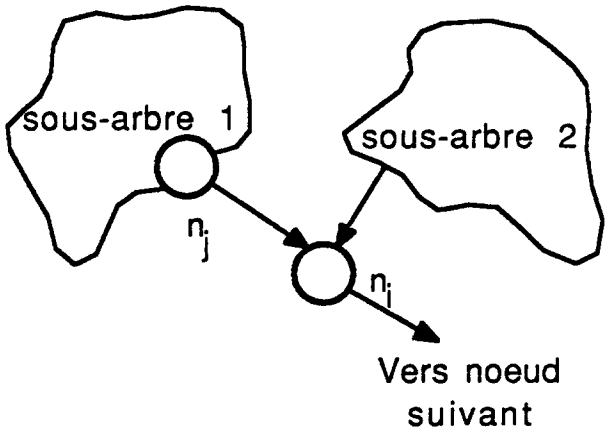


figure II.6

soit n_j le nœud du sous-arbre 1 lié à n_i. La structure ci dessus représente la phrase dont le diagramme syntaxique est donné ci-dessous:

<phrase> ::= <$expr_1$> <op_i> <$expr_2$> ....

Où $expr_1$ (resp. $expr_2$) est l'expression décrite par le sous-arbre 1 (resp. 2).

Si n_i est un nœud de type ET, il faut donc qu'*$expr_1$* et *$expr_2$* soient toutes les deux vraies; supposons que lors de l'exploration de l'arbre on arrive de n_j vers n_i, n_i est alors marqué pour indiquer qu'il a été atteint au moins une fois; s'il était précédemment non marqué, c'est que *$expr_2$* était fausse, il est alors inutile de passer au nœud suivant, *expr* étant alors toujours fausse; au contraire, si le nœud était déjà marqué (il n'a alors pu l'être que lors de l'exploration du sous-arbre 2), *expr* est donc vraie et l'exploration de l'arbre complet peut progresser vers le nœud suivant.

Au contraire, dans le cas d'un nœud de type OU, il suffit que l'un au moins des deux opérandes soit vrai; dans ce cas, si, en provenant de n_j, on trouve le nœud n_i non marqué, on peut poursuivre l'exploration de l'arbre avec le nœud auquel n_i est lié, en ayant au préalable marqué n_i. A l'opposé, si en cours d'exploration, on trouve le nœud n_i déjà marqué, il est inutile de la poursuivre plus loin, puisqu'elle a fatalement "buté" plus loin sur un nœud ET non encore marqué (cf. règle précédente) ou sur un nœud OU déjà marqué (cf. règle courante).

Le traitement des nœuds terminaux est globalement identique, à ceci près qu'un nœud terminal contient l'adresse vers laquelle il faut se dérouter si l'expression totale est vraie. Le seul fait d'atteindre un nœud terminal OU entraîne la véracité de l'expression totale; il est donc inutile de marquer ce nœud comme dans les autres cas.

II.2.3.4 Implantation pratique sous CESAR

Après une brève description de la chaîne de développement pour CESAR, nous étudierons dans ce paragraphe la façon dont le compilateur d'expressions logiques a été intégré à un environnement déjà existant ainsi qu'une implantation pratique des structures définies plus haut.

II.2.3.4.1 Présentation de la chaîne de développement

La chaîne de développement utilisée pour la production et le chargement de code sur CESAR est représentée par le schéma ci-dessous:

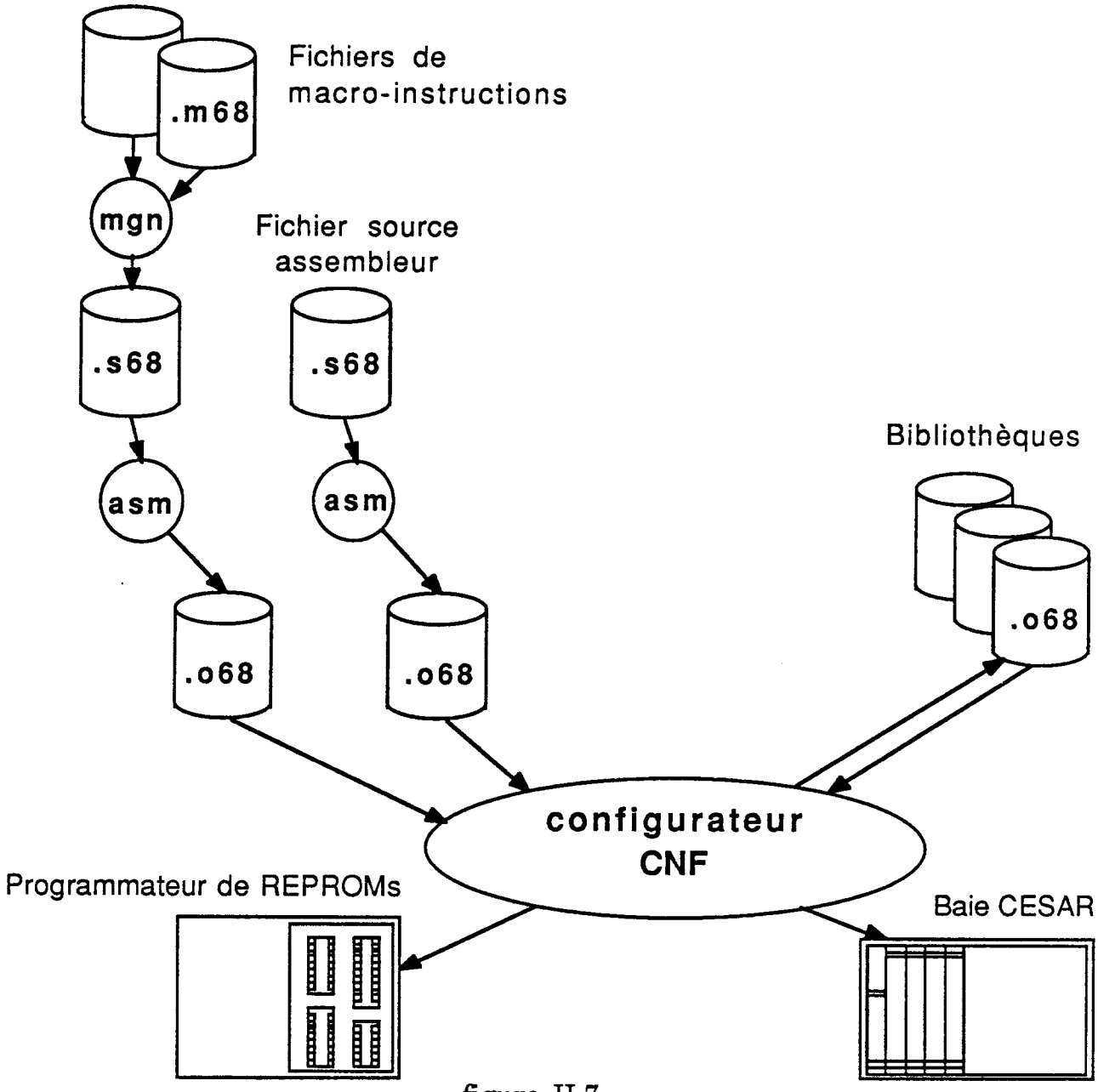


figure II.7

Le cycle de production d'un logiciel passe tout d'abord par l'assemblage des différents modules sources de l'application (fichiers de suffixe .S68) puis par l'édition de lien des fichiers objets (suffixe .O68) et leur chargement sur la baie CESAR au moyen du configurateur CNF. L'assemblage des modules peut

séparer des instructions spécifiques à des composants physiques déterminés du code indépendant du matériel d'une application; les fichiers à macro-générer ont un suffixe .M68. Enfin, il est possible d'automatiser le fonctionnement de "l'éditeur de liens/chargeur" multi-processeur CNF, en écrivant des fichiers dits "de configuration" (fichiers de suffixe .CNF, non représentés sur la figure ci-dessus) contenant les directives que CNF exécutera en séquence.

La compilation des expressions logiques de requêtes a été insérée avant la macro-génération. L'étape de "compilation" consiste à explorer le fichier d'entrée et à le recopier tel quel en sortie, sauf si des directives spécifiques au compilateur sont rencontrées. Après compilation, le fichier résultant sera macro-généré avec les bibliothèques de macros de l'utilisateur plus une bibliothèque comprenant les macros nécessaires à l'évaluation en temps réel des expressions logiques. Nous avons choisi de passer par une phase de macro-génération dans un souci de simplification du compilateur d'expressions logiques; en effet, il est beaucoup plus simple au niveau du compilateur d'expressions logiques de générer des appels à des macro-instructions qui seront expansées ultérieurement, que de générer directement du code source alors qu'un macro-générateur est présent dans la chaîne de développement existante. Il est à noter, enfin, que la phase de "compilation" a été insérée dans la chaîne de façon à ne pas avoir à modifier les fichiers sources (M68 ou S68) qui n'utiliseraient pas le "compilateur".

Le reste de la chaîne de développement ne subit pas de modifications.

II.2.3.4.2 Structures des nœuds sous CESAR

Dans le cadre du "compilateur d'expressions logiques" nous avons voulu privilégier la rapidité d'évaluation par rapport à la longueur du code généré (bien que dans le cas d'arbres ne comportant pas plus d'une demi-douzaine de feuilles, il soit difficile de mettre au point un code plus court); aussi, pour la solution que nous avons retenue, chaque nœud est plus qu'une variable structurée au sens des langages de haut niveau: tout nœud comprend, en effet, les deux champs cités plus haut (en ce sens, un nœud est une variable structurée), mais en plus, il inclut une portion du code destiné à l'évaluation itérative de l'arbre. Il existe quatre types de nœuds: les nœuds **OU** et **ET intermédiaires** et les nœuds **OU** et **ET terminaux** dont les structures sont données ci-dessous:

N_ud_OU:

```
        TAS     FLAG_OU Marque le n ud, s'il n'était pas déjà...
        BEQ     SUIVANT ... marqué, on passe au suivant.
        RTS             Arrête l'exploration
FLAG_OU DS.W    2
```

N_ud_ET:

```
        TAS     FLAG_ET Marque le n ud, s'il était déjà ...
        BNE     SUIVANT ... marqué, on passe au suivant.
        RTS             Arrête l'exploration
FLAG_OU DS.W    2
```

Un nœud terminal est composé d'une structure similaire:

N_ud_OU_Terminal:

```
        ADD.W   #8,SP    Aligne la pile
        JMP     @retour  Saute à l'étiquette associée à
                         l'expression
```

N_ud_ET_Terminal:

```
        TAS     FLAG_ET  Marque le n_ud
        JMP     @retour  Saute à l'étiquette ...
EXIT    RTS              Arrêt de l'exploration
FLAG_OU DS.W    2
```

Chaque nœud intermédiaire et chaque nœud ET terminal est composé de deux champs:

- Un marqueur indiquant si le nœud a été exploré ou non.
- Pour les nœuds intermédiaires, SUIVANT pointe sur le nœud suivant dans l'arbre (par exemple, dans le cas de l'arbre décrit en figure II.4, le nœud 4 est le suivant du nœud 3). Pour un nœud terminal le champ @retour contient la valeur de l'étiquette associée à l'ordre $IF courant.

N.B.:

1) Les nœuds OU terminaux, ne comportent pas de marqueurs, puisqu'il suffit d'arriver sur un tel nœud pour prouver la véracité de l'expression.

2) Dans les cas des nœuds intermédiaires ET et OU et des nœuds ET terminaux, nous utilisons l'instruction TAS (Test And Set). Ce choix est dû non au fait que cette instruction est indivisible (propriété fondamentale pour assurer l'exclusion mutuelle en fonctionnement multi-processeurs; ici la structure arborescente n'est manipulée que par un seul processeur), mais parce que c'est l'instruction la plus rapide car elle permet **à la fois** d'effectuer le test **et** de marquer le nœud.

II.2.3.5 Fonctionnement du "compilateur d'expressions logiques"

Pour faciliter l'étude du fonctionnement du compilateur d'expressions logiques, nous nous appuyerons sur son implantation adaptée à la chaîne de développement de code pour CESAR. Cependant l'algorithme de son fonctionnement est suffisamment général pour pouvoir être aisément modifiable quel que soit le choix de l'exécutif temps réel[1] et de la représentation des nœuds (seules les fonctions de plus bas niveau sont en fait dépendantes des choix effectués). Le "compilateur d'expressions logiques", dont le listing est donné en Annexe A, a été implanté sur un calculateur Perkin Elmer 3270 et écrit en Fortran 77 en essayant d'utiliser un maximum d'instructions standards (au sens de la norme Ansi 77) de façon à faciliter sa portabilité.

II.2.3.5.1 Structure générale du compilateur d'expressions logiques

La structure générale du programme est décrite ci-dessous:

```
Compilateur:
Debut
    /* Initialisations */
    Compteur de ligne <- 0
    Booléens OKWAIT <- OKIF <- Faux
    Ouverture des différents fichiers
    Repéter
        Lire une ligne
        Incrémenter le compteur de ligne
        Si les 2 premiers caractères de la ligne sont $* Alors
        Analyse_ligne
        Sinon
            /* Première ligne après un bloc $WAIT-$IF */
            Si OKWAIT Alors
                Si !OKIF Alors
                    Erreur (Pas de IF après WAIT)
                Sinon
                    /* Génération du code d'un bloc $WAIT-$IF */
                    Genere_code
                    OKWAIT <- OKIF <- Faux
                FinSi
            FinSi
            Ecriture de la ligne sur le fichier de sortie
        FinSi
    Jusque Fin de fichier d'entrée
    Fermer tous les fichiers
    Afficher le nombre d'erreurs éventuelles
Fin
```

figure II.8

La boucle principale du "compilateur d'expressions logiques" consiste en une exploration ligne à ligne du fichier d'entrée; toutes les lignes non reconnues par le "compilateur d'expressions logiques" (ne commençant pas par *$) sont recopiées telles quelles en sortie; toutefois, si une ligne non reconnue par le "compilateur d'expressions logiques" termine un bloc $WAIT-$IF, le code correspondant à ce bloc est généré avant que la ligne ne soit recopiée sur le fichier de sortie. Les variables booléennes *OKWAIT* et *OKIF* permettent de mémoriser l'arrivée des ordres $WAIT et $IF respectivement; le "compilateur d'expressions logiques" les utilise pour vérifier que la succession des ordres $WAIT et $IF est bien respectée et signale des erreurs à l'utilisateur dans les cas contraires (ordre $WAIT non suivi d'au moins un ordre $IF, ordre $IF non précédé d'un ordre $WAIT, ...).

```
Analyse_ligne:
Debut
    /* Décomposition de la ligne en différents champs */
    Decomposer_ligne
    Cas ordre De
        '$WAIT':
            Si OKWAIT Alors
                /* Ordre $WAIT fermant un bloc $WAIT-$IF */
                Si OKIF Alors
                    GenereCode
                Sinon
                    Erreur (Pas de IF après WAIT)
                FinSi
                OKIF <- Faux
                /* Recherche des arguments de $WAIT puis tri par
                 ordre alphabétique */
                RechercheArguments
                TriArguments
                ng <- ng + 1
                Ecrire en sortie: "<TAB>WAIT&<TAB>nb,ng"
                Répéter Pour chaque argument
                    Ecrire en sortie: "<TAB>TAB&<TAB>arg,ng"
                FinRépéter
            FinSi
            OKWAIT <- Vrai
        '$IF':
            Si!OKWAIT Alors
                Erreur (Pas de WAIT avant le IF)
            Sinon
                OKIF <- Vrai
            FinSi
            /* Analyse syntaxique de l'expression suivant l'ordre $IF */
            AnalyseSyntaxique
        Autre:
            Erreur (ordre inconnu)
    FinCas
Fin
```

figure II.9

N.B.:
Si, lors du traitement d'un ordre $WAIT, les variables booléennes *OKWAIT* et *OKIF* sont simultanément vraies, c'est que l'ordre $WAIT en cours de traitement est la première instruction après un bloc $WAIT-$IF; il y a donc lieu de générer le code correspondant à ce bloc avant de poursuivre l'analyse du nouveau bloc commencé par $WAIT.

La variable *ng* est un compteur de blocs $WAIT-$IF; lors de la génération des macro-instructions de synchronisation par le "compilateur d'expressions logiques", la valeur de cette variable est systématiquement concaténée aux étiquettes d'entrée des macro-instructions de manière à éviter que le code correspondant à plusieurs blocs $WAIT-$IF ne comporte plusieurs fois la même étiquette, ce qui serait très génant lors de l'assemblage qui suit normalement la compilation des expressions logiques.

Le tri des arguments de l'ordre $WAIT par ordre alphabétique est destiné à faciliter la vérification (qui pourra se faire par recherche dichotomique) de la déclaration de toutes les variables qui apparaissent dans les expressions logiques des ordres $IF du bloc $WAIT-$IF courant.

II.2.3.5.2 Génération des structures associées à l'ordre $WAIT

A chaque apparition d'un ordre $WAIT le "compilateur d'expressions logiques" génère tout d'abord une portion de code effectuant la mise en attente des requêtes spécifiées dans la zone argument de l'ordre $WAIT (écriture d'un appel à la macro-instruction WAIT& dans le cas de l'implantation spécifique à CESAR[1]) puis une table (employée par la macro précédente) dont la structure est donnée ci-dessous, chaque entrée dans la table est associée à une requête; cette table est générée par autant d'appels à la macro-instruction TAB& qu'il y a d'arguments (ou des requêtes) dans la zone argument de l'ordre $WAIT.

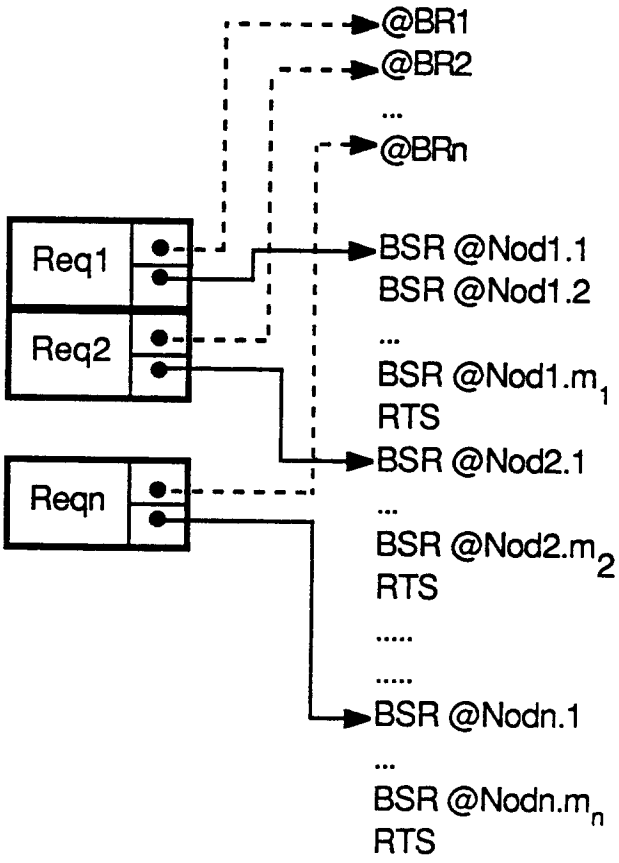


figure II.10

Chaque entrée dans la table est composée de deux champs:

- Le premier est l'adresse d'un pointeur (dans l'exemple II.10 les variables REQi sont de tels pointeurs) sur le bloc requête. Cette double indirection est rendue nécessaire par le fait que l'adresse d'un bloc requête est fournie par le système, et est donc susceptible de varier lors d'envois différents d'une même requête; l'utilisateur doit donc créer,

[1] Pour une implantation destinée à un autre couple système temps réel/chaîne de développement,

pour chaque requête, une variable de type pointeur (mot long en assembleur 68000) dans laquelle il rangera, avant chaque envoi d'une requête, l'adresse du bloc qui lui aura été alloué par le système; de ce fait le nom symbolique du pointeur (ou son adresse, ce qui est équivalent en assembleur) représente l'objet requête au fil du temps.

- Le deuxième champ est un pointeur vers une zone du segment de code de l'application; cette zone dont la structure suit constitue la liste des nœuds qui sont liées à une même "variable requête".

```
BSR     @NODi_1
BSR     @NODi_2
...
BSR     @NODi_mi
RTS
```

Où: m_i est le nombre de nœuds liés à la $i^{ème}$ requête
@NODi_j est l'adresse du $j^{ième}$ nœud lié à la $i^{ème}$ requête.

Chaque fois qu'un ordre WAIT est rencontré, il faut générer l'appel à une seéquence de code effectuant la mise en attente de la terminaison des requêtes passées comme arguments de l'ordre et l'aiguillage vers les nœuds associés chaque fois qu'une requête se termine. Du fait de la structure choisie pour l'implantations des tables, l'algorithme associé à l'ordre $WAIT (macro-instruction WAIT&) est donné ci-dessous:

```
Répéter Toujours
    Scrutation:
    Répéter Pour chaque requête
        Si la requête est terminée Alors
            marquer la requête
            branchement vers la table de nœuds associée à la requête
            AllerA scrutation:
        FinSi
    FinRépéter
    Mise en attente
FinRépéter
```

figure II.11

Notes:

- Le marquage des requêtes terminées a pour but d'éviter d'explorer de nouveau les tables associées à ces requêtes chaque fois qu'un événement arrivera.
- Cet algorithme peut sembler boucler indéfiniment; en fait, la sortie de la boucle principale se fait lors de l'exploration de l'arbre représentant les différentes expressions logiques, chaque fois qu'un nœud terminal est atteint.
- Si, après l'arrivée d'une requête et exploration de l'arbre l'on reste dans la boucle de scrutation, c'est qu'aucune des conditions terminales n'a été atteinte (cf. remarque précédente); il y a lieu de recommencer la boucle de scrutation depuis le début, afin d'éviter de manquer des événements dans le cas où plusieurs requêtes se termineraient

- Si, après exploration complète de la boucle de scrutation, toutes les requêtes ont été trouvées "en cours" ou déjà traitées (marquées), il faut génèrer une mise en attente; seule, en effet, la complétion d'une requête encore en cours est désormais susceptible de faire évoluer la situation.

II.2.3.5.3 Analyse syntaxique de l'expression logique associée à un ordre $IF

L'analyse de l'expression logique qui précède un ordre $IF se fait suivant une méthode classique que nous exposerons ci-dessous:

A chaque opérateur est affectée une priorité:
0 pour (et)
1 pour +
et 2 pour .

l'analyseur syntaxique utilise deux piles: l'une qui contiendra le nom des opérandes, l'autre qui contiendra les opérateurs.

- Chaque fois qu'un opérande est reconnu, il est mis au sommet de la pile des opérandes.
- Quand un opérateur est analysé quatre cas peuvent se produire:
  - Règle1: L'opérateur actuel est plus prioritaire que l'opérateur précédent (si ce dernier existe); il est alors empilé dans la pile des opérateurs. Il est en de même s'il s'agit du premier opérateur rencontré. La priorité actuelle est celle de l'opérateur actuel.
  - Règle 2: L'opérateur courant (o1) est de même priorité que l'opérateur précédent (o2) ou de priorité inférieure; l'opérateur au sommet de la pile des opérateurs (o2) est alors appliqué aux deux opérandes au sommet de la pile des opérandes, ces deux derniers sont dépilés et sont remplacés par le résultat de l'opération (qui est un nœud), l'ancien opérateur o2 est remplacé au sommet de la pile par l'opérateur o1, la priorité actuelle est alors celle de o1.
  - Règle 3: L'opérateur courant est une parenthèse ouvrante, elle est empilée et la priorité actuelle devient nulle.
  - Règle 4: L'opérateur courant est une parenthèse fermante; les deux piles sont alors explorées (dépilage de l'opérateur et des deux opérandes du sommet de leurs piles respectives puis empilage du résultat) jusqu'au dépilage d'une parenthèse ouvrante.

Illustrons ces règles par un exemple:

Soit à analyser l'expression a + b * (c * d + e)

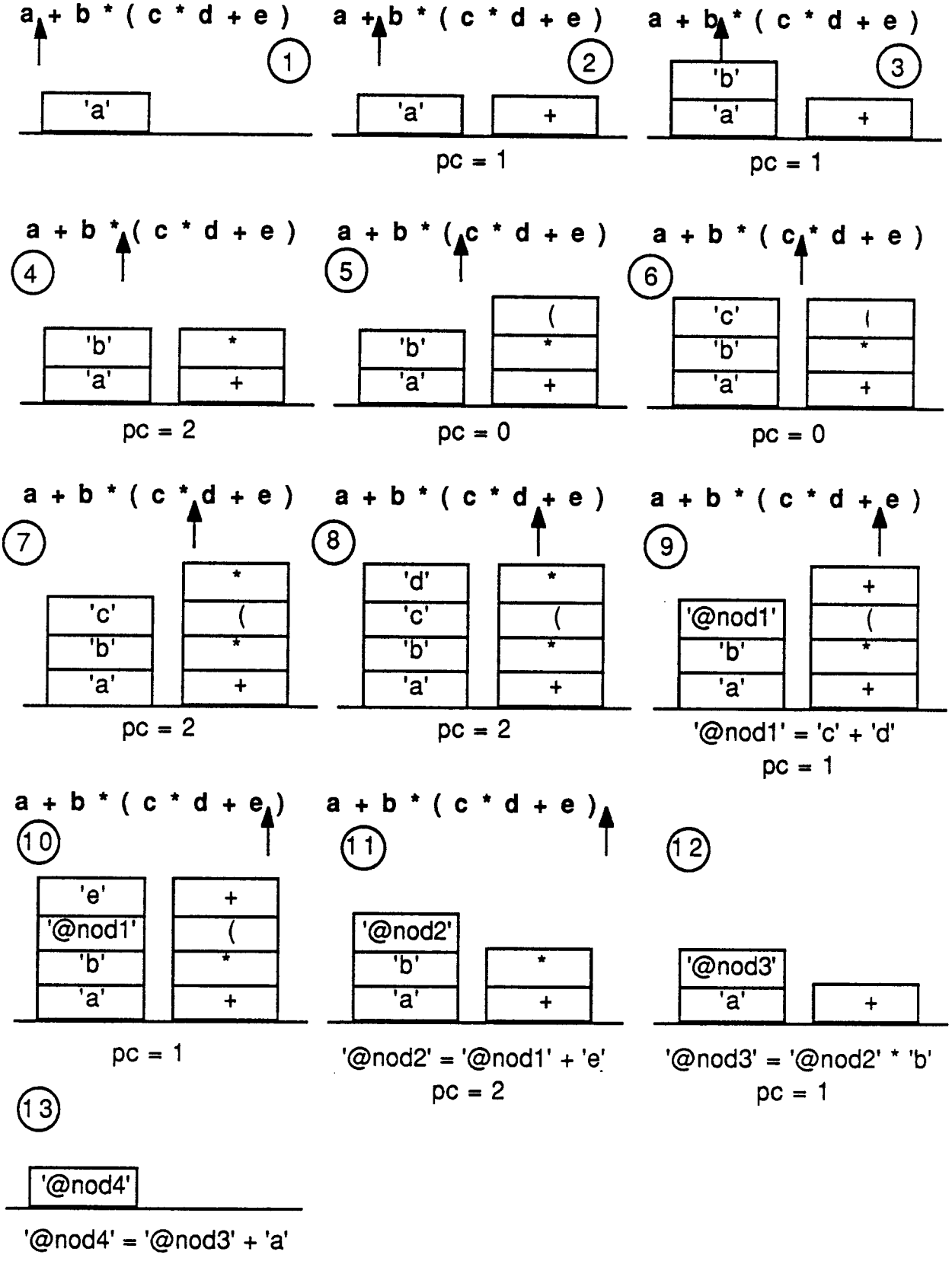


figure II.12

N.B.: Le symbole pc représente la priorité courante (celle du dernier opérateur empilé). Les parenthèses, bien qu'étant toujours considérées comme opérateurs de plus forte priorité (elles permettent de changer l'ordre normal d'évaluation) se voient ici affecter la priorité 0 de façon à permettre l'empilage de n'importe

Lors de la génération du code assembleur constituant l'arbre binaire représentant une expression, l'analyseur syntaxique utilise deux fonctions:

- **evaltt()** qui vide les 2 piles selon la règle 4 jusqu'à rencontrer une parenthèse ouvrante dans la pile opérateur; à chaque opération de dépilage, cette fonction fait appel à la fonction eval().
- **eval(opérande1, opérande2, opérateur)** qui est chargée de la construction itérative de l'arbre binaire; à chaque appel cette fonction construit une "fourche" du type:

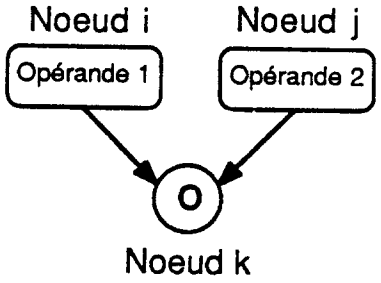


figure II.13

au fil des appels à eval(), chaque nœud se voit affecter un numéro attribué par ordre croissant. L'arbre est représenté de manière interne au "compilateur d'expressions logiques" par une table à une entrée: le fait que le $n^{ième}$ nœud de l'arbre est relié au $m^{ième}$ avec le sens n->m élément de la table est traduit par: la valeur de la $n^{ième}$ entrée de la table est m. L'exemple précédent serait donc représenté par la table suivante;

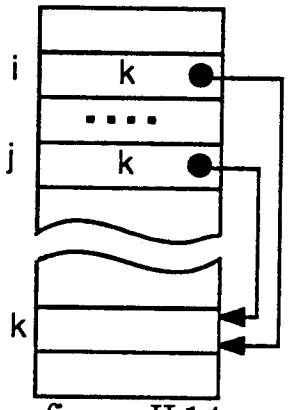


figure II.14

Enfin, chaque fois qu'un bloc $WAIT-$IF est terminé, le "compilateur d'expressions logiques" appelle la procédure GENSYN qui va explorer les tables produites par la fonction eval() et générer deux portions de code distinctes et consécutives:

- La première, qui sera appelée avant chaque exécution de la macro WAIT&, a pour but de remettre à zéro tous les nœuds des arbres concernés par l'ordre $WAIT courant. Cette précaution est indispensable chaque fois qu'un même bloc $WAIT-$IF est utilisé plusieurs fois dans une même application (ce qui constitue le cas général pour les applications temps réel qui, le plus souvent, exécutent perpétuellement la même boucle principale); après exploration d'un

résiduel avant d'effectuer la mise en attente des différentes requêtes passées en paramètres à l'ordre $WAIT.

- La seconde est la traduction en assembleur des arbres binaires sous forme de suites d'appels aux macros ET et OU intermédiaires fermées par des appels aux macros ET ou OU terminales.

II.3 Tache généralisée

II.3.1 Introduction

Lors de la phase de définition de son projet, le concepteur d'une application temps réel se trouve confronté à la spécification de deux entités inter-dépendantes qui constituent son application:

- La partie commande

- La partie opérative ou partie données.

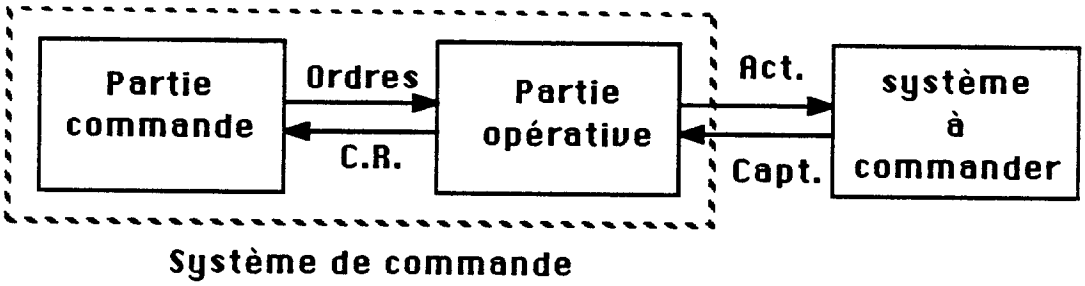


figure II.15

- La partie opérative est un ensemble de variables (ou objets) et un ensemble d'opérateurs. Les variables représentent l'état de la partie opérative qui peut être modifié par l'activation d'un opérateur.
- La partie commande gère le séquencement des opérateurs de la partie opérative. Elle peut être représentée par un automate dont les sorties (associées aux états ou aux transitions) correspondent à des ordres d'activation des opérateurs. Les entrées de la partie commande sont des comptes-rendus sur l'état de la partie opérative.

La coopération de deux parties peut se résumer de la façon suivante: lorsque la partie commande émet un ordre d'exécution d'un opérateur, cet ordre est maintenu jusqu'à la réception d'un compte-rendu qui lui permet d'évoluer. Un compte-rendu est le signal de fin d'exécution d'une opération et éventuellement un renseignement concernant le choix d'un nouvel état de commande.

De manière générale, la partie opérative d'un système de commande peut se concevoir de façon classique par analyse descendante (en général, il est possible de distinguer quatres phases: spécification des besoins, analyse générale, analyse détaillée et codage) ; sa validation se faisant, au contraire, en sens inverse: le modules opératifs de plus bas niveau sont validés (test unitaires) avant de passer à ceux de niveau supérieur (intégration puis test de sous-ensembles).

Par contre, en ce qui concerne la partie commande, les choses se

système temps réel pouvant conduire à des aléas de fonctionnement ou même dans certains cas graves (erreurs de conception) à des blocages. Il faut là aussi disposer d'outils d'analyse et de spécification. A l'heure actuelle deux méthodes se dégagent nettement: les Réseaux de Petri et le GRAFCET. Ces deux méthodes permettent une analyse descendante grâce à la notion de bloc.

Cependant, l'application simultanée de deux méthodes (une pour la partie opérative et une pour la partie commande) se heurte à de nombreuses difficultés: très souvent les deux parties sont intimement liées. En effet, il arrive fréquemment, qu'après avoir validé le fonctionnement hors-temps réel du système (c'est-à-dire la partie purement algorithmique), l'utilisateur insère des appels au moniteur temps réel dans le code constituant les agents actifs de l'application afin d'en assurer l'enchaînement temporel.

Cette façon de faire appelle plusieurs remarques:

- Chaque fois qu'il est nécessaire de changer le séquencement des tâches composant une application, il est nécessaire de modifier des modules qui ont déjà été validés (ceux qui contiennent des appels au système temps réel), ce qui n'est pas très bon du point de vue méthodologique.
- Ces modifications imposent une recompilation des modules touchés, alors qu'hormis les appels au système temps réel, ces modules n'ont pas été modifiés.

La définition d'une interface unique aux "tâches opératoires" et aux "tâches de synchronisation", permet de généraliser la notion de tâche[1] aux modules de synchronisation. L'approche que nous proposons permet la séparation effective des parties commandes et opératives de manière à pouvoir appliquer les méthodes de conception et de spécification de logiciels aux deux parties d'un système de commande:

Il doit être possible à l'utilisateur de produire d'une part des modules "opératifs" (ou de calcul pur) et d'autre part des modules de synchronisation assurant l'enchaînement temporel adéquat des modules opératifs.

II.3.2 Concepts utilisés

Pour le développement de la tâche généralisée et des logiciels de synchronisation associés, nous nous sommes basés sur les concepts que nous définirons dans la suite de ce chapitre. De ces notions vont découler les spécifications minimales que devra remplir le noyau de synchronisation que nous présenterons par la suite.

II.3.2.1 Synchronisation hiérarchisée - Notion de bloc

De la même façon que dans n'importe quel langage de programmation (depuis les langages de haut niveau jusqu'à l'assembleur) est définie la notion de sous-programme, il est possible de définir un concept équivalent en ce qui concerne la synchronisation.

Afin d'illustrer notre propos, nous utiliserons le formalisme des Réseaux de Petri [COUR 80, BRAM 83].

Exemple:

Considérons le réseau de Petri décrivant l'enchaînement temporel des tâches réalisant les asservissements de la commande hybride (position-force) du poignet actif développé au CERT-DERA ([ROBE 87]).

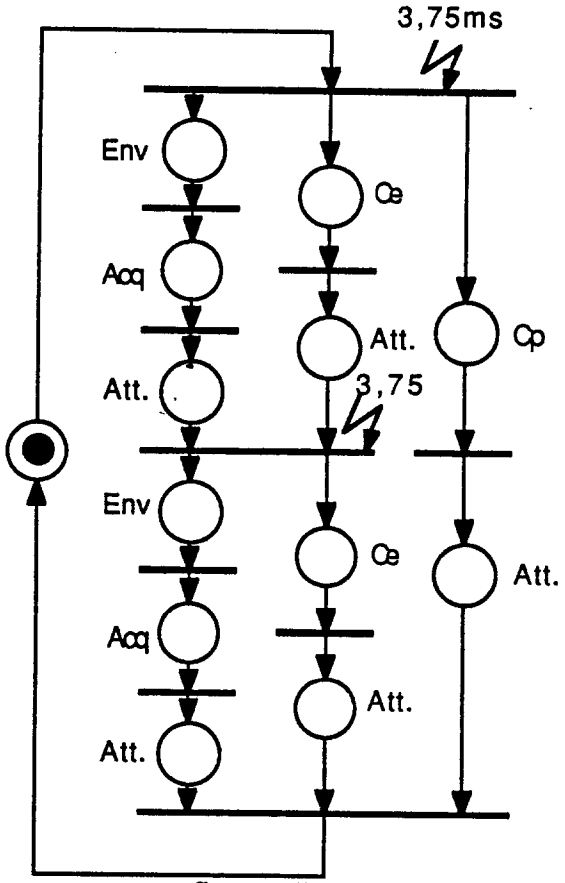


figure II.16

Où: Env est l'envoi de la consigne mixte (comprenant la contribution de l'asservissement en effort et celle de l'asservissement de position).

Acq est l'acquisition des allongements des brins et des capteurs d'effort.

Ce est le calcul de la consigne d'effort.

Cp est le calcul de la consigne de position.

Att. est une place d'attente.

Un sous-ensemble d'un réseau de Petri avec une seule transition d'entrée dénommée transition initiale et une seule transition de sortie, dite transition finale est dénommé **bloc** [COUR 80].

Considérons un bloc possédant les "bonnes propriétés" (réseau vivant, sauf, ...) inclus dans un schéma à Réseau de Petri plus général. Il est possible de remplacer tout le bloc par une seule transition; le réseau simplifié ainsi obtenu aura les mêmes propriétés que le réseau initial. En pratique, cette propriété se révèle très intéressante pour simplifier et valider le réseau de Petri initial.

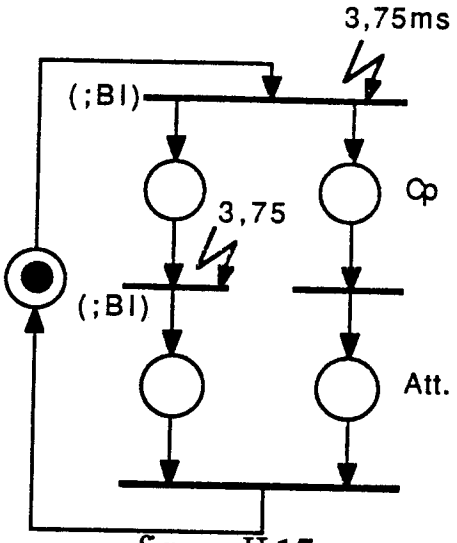


figure II.17

Chaque fois que les deux transitions temporisées à 3,75ms sont tirées le bloc Bl (composé de Env, Acq et Ce) est exécuté[1].

Il est clair que la notion de bloc est nécessaire dès que l'on veut avoir une approche structurée de la synchronisation: dans un premier temps, l'utilisateur décrit la synchronisation globale de son application puis remplace certaines transitions par des blocs représentant une synchronisation plus locale.

Il est donc tout à fait possible de considérer que le bloc est à la partie commande ce que le sous-programme est à la partie opérative.

Afin de pouvoir appliquer les méthodes de l'analyse descendante lors de la définition et de la programmation de la partie commande, l'utilisateur doit avoir la possibilité d'employer des modules de synchronisation comme "sous-programmes" d'autres modules.

II.3.2.2 Ressource logicielle

Définition: Soit un module *mp* utilisant un module *mf* comme "sous-programme" de synchronisation, nous dirons que *mf* est "le fils" de *mp*, ou réciproquement que *mp* est "le père" de *mf*.

Afin de garantir la cohérence de l'information de synchronisation, un même module ne pourra être **simultanément** le fils de deux modules différents (qui pourraient lui envoyer des ordres contradictoires, si cette possibilité était permise).

Considérons, en effet l'exemple suivant:

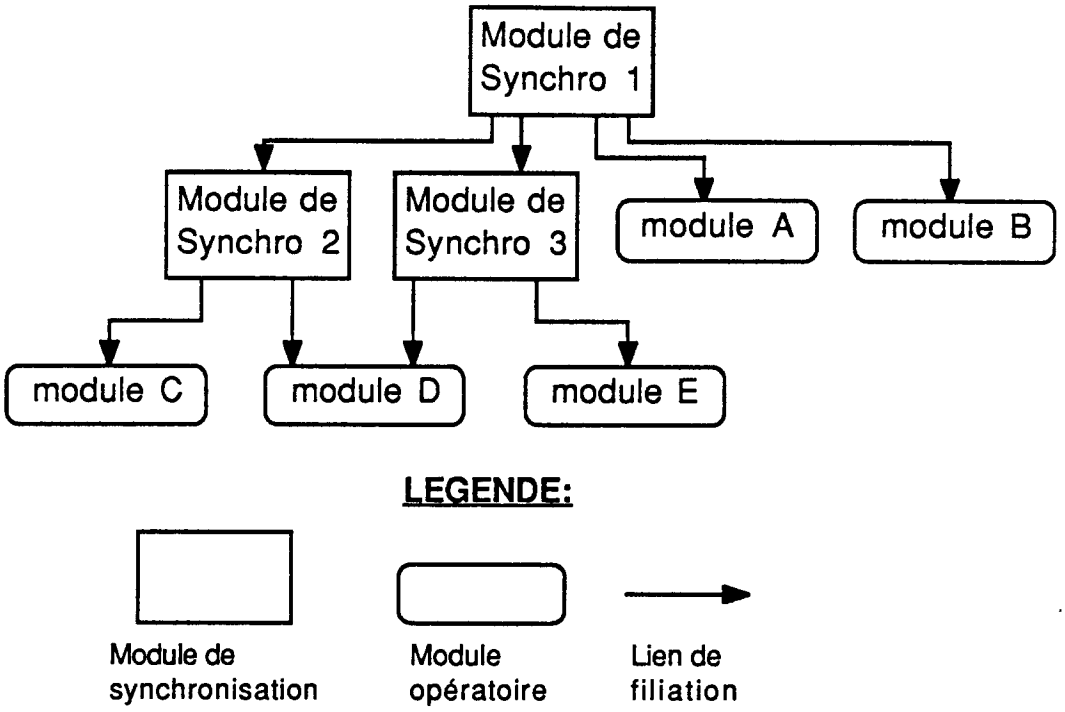


figure II.18

D'après le schéma, il est clair que MS_2 et MS_3 peuvent être les pères de MOD_D (qui est ici un module opératif, mais qui pourrait aussi bien être un module de synchronisation).

Ce cas se rencontre chaque fois qu'une ressource physique unique et non partageable (Exemple: chariot dans un atelier flexible) est gérée par un module (qui, dans notre exemple, pourrait assurer le pilotage du dit chariot); dans ce cas, le module en question peut être considéré comme une ressource logicielle non partageable.

Afin de garantir le bon fonctionnement du système de synchronisation, il faudra implanter un mécanisme d'**exclusion mutuelle**: tout module père, avant d'envoyer des ordres vers un de ses fils, devra dans un premier temps s'assurer que ce dernier est libre; dans le cas où le module fils n'est pas libre plusieurs stratégies sont possibles: soit attente de la libération du fils, soit exécution d'une séquence de traitement d'échec.

II.3.3 Synchronisation par les données

Tout module opératoire peut être représenté par le schéma général suivant:

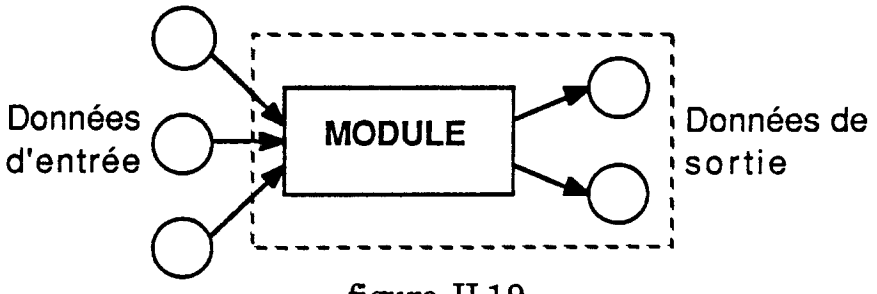


figure II.19

où les flots de données d'entrée ou de sortie peuvent éventuellement être absents.

A un instant t, il peut exister un flot de données entre deux modules:

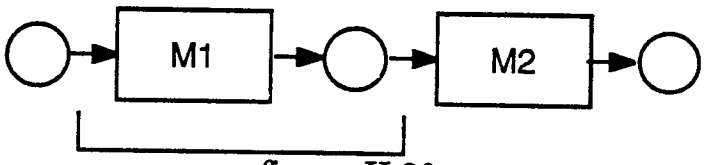


figure II.20

Dans cet exemple, le module M_2 consomme des données produites par le module M_1.

En plus de la synchronisation temporelle classique, apparaît une synchronisation par les données: il est tout à fait possible qu'un module ayant reçu du niveau supérieur un ordre d'activation, doive attendre que des données appartenant à son flux d'entrée soient prêtes.

Les flots de données entre les différents modules sont des entités dynamiques: en cours de fonctionnement de l'application, le flux de données entre M_1 et M_2 dans notre exemple peut très bien être coupé et d'autres flux créés (M_2 recevant donc son flot d'entrée d'un autre producteur que M_1); cette possibilité est fondamentale car une modification dynamique du séquencement temporel de modules opératoires s'accompagne très fréquemment d'une reconfiguration du graphe de données.

II.3.3.1 Réalisation sous CESAR

Tous les exécutifs temps réels possèdent de nombreux services de synchronisation tant temporelle que par les données.

Cependant, certains sont à utiliser avec précaution; ainsi, lors du développement de la "tâche généralisée" nous sommes nous interdit l'utilisation de la fonction **pause** sous CESAR ou son équivalent pour d'autres exécutifs temps réel: rappelons que cette fonction fait passer une tâche de l'état activable ou en attente à l'état dormant. Si l'on utilise cette fonction sans discernement, les résultats peuvent être catastrophiques:

Exemple: Pause d'une tâche d'asservissement en vitesse entre deux périodes d'échantillonage; la vitesse étant calculée par la formule approchée: $v=\backslash F(x_t - x_{t-1},\Delta t)$, si la période entre deux échantillons Δt n'est pas constante, les résultats peuvent être imprévisibles.

Cet exemple est, certes, caricatural, mais lors de la définition des différents états d'une "tâche généralisée" nous avons imposé une restriction: une tâche généralisée ne peut être désactivée qu'**en des points bien particuliers**; tout ordre de désactivation arrivant en dehors de ces points est cependant mémorisé et ne sera traité que lorsque la tâche généralisée sera de nouveau réceptive.

D'autre part, s'il y a synchronisation hiérarchisée, la désactivation d'une tâche de synchronisation doit provoquer la désactivation de toutes ses filles (à

Nous avons vu précédemment qu'il était possible de reconfigurer les flots de données d'une tâche généralisée, mais de même que ci-dessus, cette opération n'est autorisée qu'en certains points.

Les flots de données entrants et sortants de la tâche généralisée transiteront par des files d'échanges (boîtes aux lettres) de manière à laisser au système la gestion de la synchronisation entre les producteurs de données et leurs consommateurs. Afin de permettre les reconfigurations du graphe de données, les canaux de données entre deux tâches doivent pouvoir être rompus, et d'autres canaux créés; deux solutions sont alors possibles:

- soit une tâche généralisée reçoit toujours ses données d'entrée sur les mêmes canaux (boîtes aux lettres), et en fonction des reconfigurations, produit ses données de sortie sur des canaux qui peuvent varier dans le temps.
- soit la tâche généralisée reçoit ses données d'entrée sur des canaux variables dans le temps mais produit ses données de sortie toujours sur les mêmes canaux.

Dans les deux cas, une reconfiguration du graphe de données consiste à lier des sorties de tâches généralisées vers des entrées d'autres tâches.

Les deux solutions étant fonctionnellement équivalentes, nous avons retenu de façon tout à fait arbitraire, la seconde.

Enfin, le noyau de synchronisation laisse la possibilité de concevoir des tâches généralisées cadencées par un temporisateur: une fois lancées, de telles tâches généralisées seront réactivées périodiquement.

II.3.3.2 Description d'une tâche généralisée

A chaque tâche généralisée doit être associé un canal d'entrée qui lui permettra de recevoir et d'acquitter des ordres d'entités de niveau supérieur. Vue de l'extérieur, chaque tâche généralisée sera connue des autres entités par son numéro de canal d'entrée. De même que toute tâche standard, une tâche généralisée est composée d'une suite d'instructions ou **code**. De façon interne, ce code peut être divisé en deux parties: le code purement opératoire qui réalise la mission de la tâche d'une part, et le code nécessaire à la gestion des services fournis à la tâche généralisée.

De manière interne, une tâche généralisée est une machine à états (et en temps que telle, peut se représenter par un Réseau de Petri, comme nous le verrons par la suite), et le code de gestion est chargé d'assurer les changement d'états en réponse aux différents ordres provenant de la file d'ordre d'entrée de la tâche.

Cependant, afin d'éviter d'avoir à dupliquer inutilement une même séquence d'instructions, le code de gestion a été conçu de manière à être réentrant et ainsi partageable par toutes les tâches généralisées portées par un même processeur; de manière à assurer cette réentrance, à chaque tâche sera associée une table qui en résumera l'état actuel.

Dans la suite de ce paragraphe, nous expliquerons quelles sont les actions susceptibles de faire évoluer l'état d'une tâche généralisée.

II.3.3.2.1 Les ordres

Ce sont des commandes provenant d'un niveau supérieur qui vont faire évoluer l'état d'un tâche généralisée. Les différents ordres peuvent être regroupés en deux classes: les uns permettent de configurer l'environnement d'une tâche généralisée (condition d'activation, ressources de données d'entrée, ...) les autres permettent le contrôle de l'exécution de la tâche (activation, désactivation, suspension provisoire de l'activité, reprise de l'activité,...); de manière générale les premiers "encapsulent" les seconds. Enfin, à tout ordre est associé un ordre "inverse" (à partir d'un état initial E, l'application d'un ordre suivie de l'application de son ordre inverse ramène la tâche généralisée dans l'état E): ainsi à l'ordre d'activation est associé un ordre de désactivation, à l'ordre de suspension d'activité un ordre de reprise d'activité, etc.

II.3.3.2.2 Les événements

Dans le cas le plus général, une fois qu'une tâche est autorisée par le niveau supérieur à démarrer, il lui faut cependant attendre l'arrivée de conditions spécifiques au temps réel: il faut, par exemple, attendre soit que ses données d'entrée soient disponibles, soit tout simplement l'arrivée d'un top de cadencement généré par un temporisateur.

II.3.3.2.3 Conditions internes à la tâche:

Enfin, une tâche généralisée peut décider d'elle-même de suspendre son activité ou de la continuer en fonction de son état interne et de la mission qu'elle doit accomplir.

Exemple:

Soit une tâche généralisée assurant le pilotage d'un robot. Cette tâche reçoit en entrée la trajectoire à suivre décrite point par point.
A tout moment, elle peut consulter la position du point terminal du robot.
Tous les Δt, en fonction de la position actuelle du robot, elle élabore une nouvelle consigne destinée aux asservissements de position.
Tant que le dernier point de la trajectoire n'est pas atteint, la tâche effectue une passe de calcul puis se remet en attente du prochain top timer.
Une fois la position finale atteinte, la tâche cesse toute activité et redevient ainsi disponible pour une nouvelle activation correspondant au suivi d'une nouvelle trajectoire.

Le réseau de Petri de la figure suivante illustre les différentes possibilités d'évolution de l'état d'une tâche généralisée en fonction des différents ordres qu'elle reçoit.

Ce réseau de Petri correspond au cas le plus général: celui d'une tâche généralisée acceptant tous les ordres (c'est-à-dire que toutes les transitions du réseau sont à priori tirables).

Afin d'en simplifier la description, nous utiliserons les notations des RDP étiquetés ([COUR 80]) que nous rappelons ici:

à chaque transition est associée une étiquette de la forme (e; a) où les champs *e* et *a* peuvent éventuellement être absents.

e est une expression logique conditionnant le tir de la transition: les places d'entrée de la transition doivent être marquées **et** la condition *e* doit être **VRAIE.**

a est une action exécutée lors du tir de la transition.

Par convention nous représenterons dans le RDP l'arrivée d'un ordre par un symbole en CAPITALES dans le champ *e*, et par des symboles en minuscules (sauf la première lettre) les différentes conditions logiques dont le positionnement dépend de l'exécution de la tâche généralisée.

Les ordres destinés à la tâche généralisée lui parviennent du niveau supérieur par le biais de son canal d'entrée.

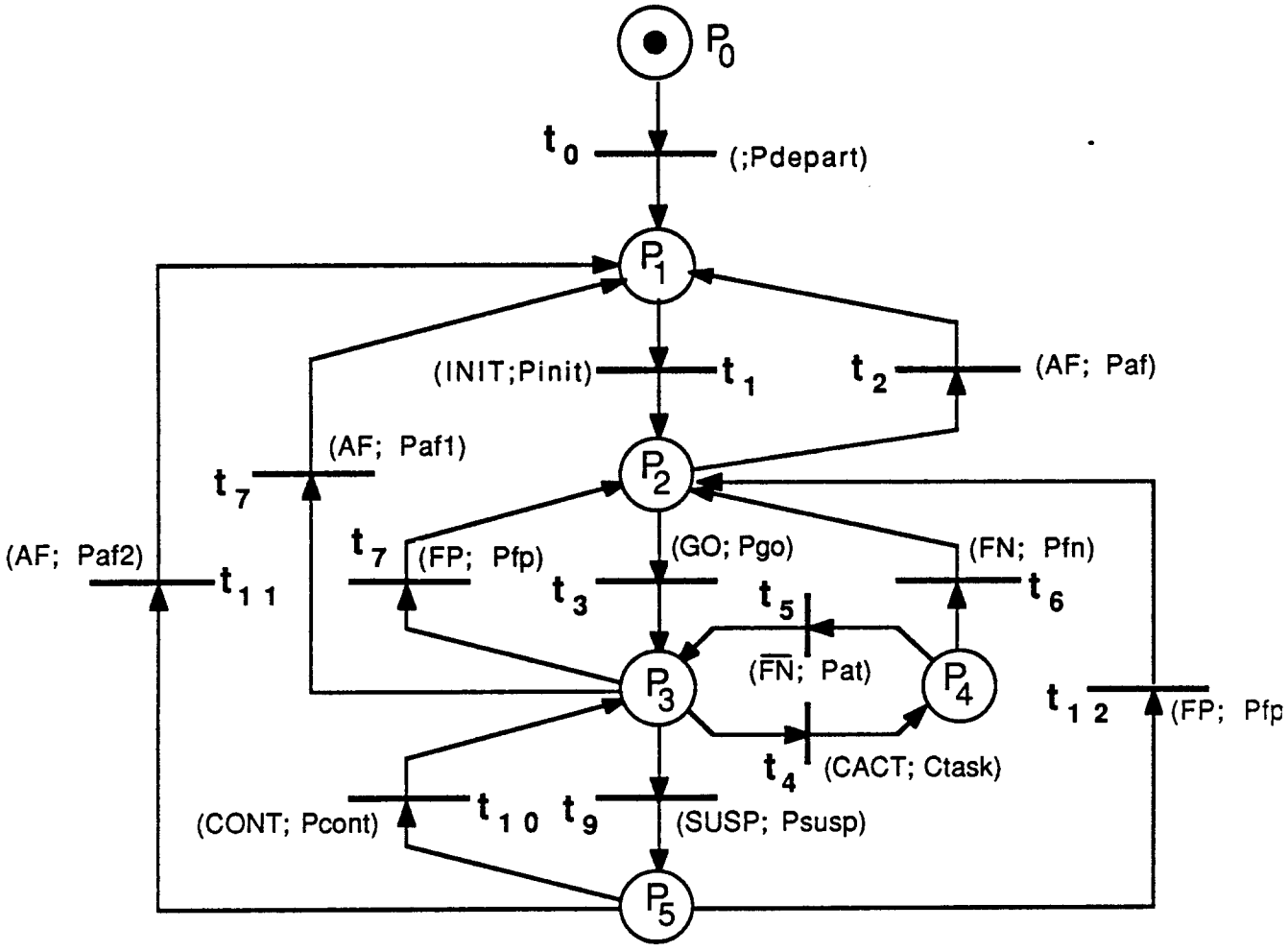


figure II.21

En place P_0 la tâche est connue du système temps réel, elle est alors dans l'état activable, état par défaut, lors de la création d'une tâche; elle n'a donc jamais été encore activée. Le tir de la transition t_0 correspond à la première activation de la tâche par l'ordonnanceur de l'exécutif (toutes les tâches plus prioritaires ne sont donc pas en l'état activable); dès ce moment la tâche exécute une séquence d'instructions qui vont la faire connaître du noyau de gestion des tâches généralisées; ce dernier reconnaît alors le canal d'entrée de la tâche, les différents ordres qu'elle accepte puis initialise les tables qui résument l'état de la tâche. En fonction de la mission qui lui est dévolue une tâche généralisée peut ne pas accepter tous les ordres; dans ce cas, suivant

revient à "effacer" purement et simplement une partie du réseau (ordres **AF**, **SUSP**, **CONT** et **FP**); dans le premier cas une procédure par défaut est exécutée lors du tir de la transition. Par la suite, pour la généralité de l'exposé du fonctionnement de la tâche généralisée, nous traiterons le cas d'une tâche qui accepte tous les ordres. Après initialisation de son environnement, la tâche effectue une lecture bloquante sur son canal d'entrée de manière à traiter les ordres à venir. Cette suite d'instructions constitue la procédure de départ ($P_{départ}$) associée au tir de la transition t_0; après exécution de la procédure de départ, la tâche se trouve en place P_1 en attente d'un ordre **INIT**.

L'arrivée d'un ordre **INIT** permet le franchissement de la transition t_1. La procédure associée à cette transition permet à la tâche généralisée de déterminer les numéros des ressources de données et de cadencement dont elle aura besoin pour être activée; après quoi, la tâche se connectera aux dites ressources de cadencement. Par le biais de l'ordre **INIT**, un niveau supérieur peut ainsi reconfigurer le graphe de données reliant entre elles des tâches généralisées de niveau inférieur; de même cet ordre permet de spécifier les conditions de cadencement d'une tâche généralisée (modification de la cadence entres deux activations successives, etc ...).

De plus, dans le cas de tâche généralisée possédant des filles, c'est-à-dire, dans le cas de tâches de synchronisation, il est possible au niveau supérieur de spécifier les numéros des canaux d'entrée des tâches filles, afin de paramétrer l'exécution de cette tâche de synchronisation. Ainsi, une tâche généralisée, lors d'une de ses exécutions pourra synchroniser *n* tâches filles suivant un certain schéma, et en synchroniser *n* autres suivant le même schéma lors d'une autre exécution.

Il existe un ordre qui consiste en "l'opposé" de l'ordre **INIT**: il s'agit de l'ordre Arrêt Final (en abbrégé: **AF**); il permet, quel que soit l'état de la tâche généralisée de la remettre en attente d'un nouvel ordre **INIT** en vue d'une reconfiguration de son graphe de données et de ses conditions d'activation. Lorsque cet ordre survient alors que la tâche généralisée est en place P_2, le noyau de gestion déconnecte la tâche généralisée de toutes les ressources de cadencement auxquelles elle avait pu se connecter lors de l'arrivée de l'ordre **INIT** précédent (exécution de la procédure P_{AF}); la tâche généralisée se retrouve alors en place P_1.

Le couple d'ordres **GO** et **AF** permet la reconfiguration dynamique du contexte d'exécution d'une tâche; les autres ordres ayant trait au contrôle de son exécution. Il est en découle que le marquage de la place P_1 représente l'état d'une tâche en attente de reconfiguration; à l'opposé le marquage d'une place de l'ensemble (P_2 , P_3 , P_4 , P_5) indique que la tâche a été configurée, elle peut alors être en activité ou en attente de certaines conditions.

Le tir de la transition t_3 correspond à l'arrivée d'un ordre **GO**. Cet ordre constitue réellement l'autorisation d'activation de la tâche généralisée: une fois que cet ordre est arrivé, il suffit que les données d'entrée de la tâche généralisée soient prêtes (c'est-à-dire, il faut que d'autres proccessus aient écrit sur toutes les boîtes aux lettres d'entrée de la tâche) et qu'un événement ait été généré à toutes les ressources de cadencement auxquelles la tâche généralisée est connectée pour que cette dernière débute son exécution. Notons, qu'afin d'augmenter la souplesse de traitement, il est possible pour le niveau

se retrouve en place P_3, en attente de ses données et signaux de synchronisation temporelle.

En place P_3, l'arrivée de toutes les conditions d'activation de la tâche (ressource de données et de synchronisation prêtes), symbolisée ici par l'étiquette CACT associée à la transition t_4, provoque l'exécution de la tâche généralisée proprement dite; la place P_4 est alors marquée. Dès ce moment là, aucun ordre, quel qu'il soit, ne peut interrompre l'exécution de la tâche généralisée tant qu'elle reste en place P_4. Cependant, ces ordres seront sérialisés et mémorisés; les différents ordres ne seront traités par ordre d'ancienneté que lorsque la place P_4 ne sera plus marquée. La sortie de la place P_4 est effectuée au choix de la tâche généralisée suivant deux possibilités fonctionnellement différentes:

- Soit l'exécution courante ne nécessite plus de traitement (Ex: arrivée sur le dernier point de consigne pour une tâche de pilotage), la tâche signale alors au noyau de synchronisation sa Fin Normale (tir de la transition t_6 étiquetée **FN**) et se retrouve en attente d'un ordre **GO** qui autorisera une nouvelle activation de la tâche généralisée.
- Soit, après une passe de calcul, il reste encore des opérations à effectuer (Ex: en reprenant la tâche de pilotage citée ci-dessus, tant que le dernier point de consigne n'est pas encore atteint), auquel cas, la tâche généralisée se retrouve en place P_3 en attente de nouvelles conditions d'activation (données prêtes et événements présents sur les ressources de cadencement). Ce cas intervient s'il y a tir de la transition t_5. Il est à noter que le conflit entre les transitions t_5 et t_6 est résolu par l'utilisation d'une variable logique FN (transition t_6) et de sa valeur complémentaire $\overline{FN}$ (transition t_5) indiquant le type de traitement à effectuer. Dans le cas de notre implantation du noyau de gestion des tâches généralisées, la valeur logique de la variable FN est liée à la valeur numérique de l'octet de poids faible du registre D0 du 68000 lors de la fin d'une passe de calcul; Si D0.B est nul, il y aura Fin Normale; dans le cas contraire, il y aura retour en place P_3. Le tir de la transition t_5 s'accompagne de l'exécution de la procédure P_{AT} dont le rôle est de relancer des lectures sur les éventuelles ressources de données de la tâche généralisée, afin de permettre le réveil de cette dernière dès que ses données d'entrées seront prêtes.

Le tir de t_6, par contre, déclenche le masquage des événements susceptibles d'être générés par les ressources de cadencement de la tâche généralisée si elle en possède; cette action correspond à l'exécution de la procédure P_{FN}.

De même qu'en place P_2, l'arrivée d'un ordre **AF**, alors que la tâche généralisée est en place P_3 provoque son retour en place P_1 (tir de la transition t_7). La procédure associée au tir de cette transition (P_{AF1}) effectue les mêmes actions que la procédure P_{AF} (transition t_2) plus une annulation de toutes les requêtes de lecture sur les éventuelles ressources de données de la tâche; cette dernière se retrouve ainsi prête à recevoir un nouvel ordre **INIT** susceptible de changer les ressources d'entrées de la tâche généralisée.

à l'état "en attente d'un ordre **GO**" (place P_2 marquée). Lors de l'arrivée d'un ordre **FP**, la transition t_8 est tirée et l'exécution de la procédure P_{FP} déclenchée. Cete dernière masque les événements pouvant provenir d'éventuelles ressources de cadencement de la tâche généralisée et elle annule les requêtes de lectures sur les ressources de données.

Le niveau supérieur peut, alors que la tâche généralisée se trouve en place P_3, suspendre provisoirement l'activité de cette dernière. Pour cela, il suffit d'envoyer un ordre **SUSP** à la tâche généralisée, il y a alors tir de la transition t_9 et passage en place P_5. Dans cet état, la tâche généralisée n'est plus sensible à l'arrivée des événements sur ses ressources de cadencement ni à celle des données sur ses ressources d'entrées. Hormis des moyens "radicaux" (envoi d'un ordre **FP** ou **AF** en place P_4, ces deux ordres ayantt alors le même effet que s'ils étaient reçus en place P_3), seule l'arrivée d'un ordre **CONT** permettra de quitter l'état suspendu pour revenir en place P_3 (transition t_{10}). L'ordre **SUSP** peut ainsi être comparé à la fonction **PAUSE** du système CESAR qui fait passer une tâche en l'état "dormant"; la seule différence est que la fonction **PAUSE** peut être exécutée par un niveau supérieur quelle que soit la position de la tâche à endormir à l'intérieur de son code opératoire, avec les dangers qu'une telle action comporte, alors que l'ordre **SUSP** ne peut être pris en compte que lorsque la tâche généralisée qui le reçoit se trouve en place P_3, c'est-à-dire, entre deux passes de calcul; le retour en place P_3 étant choisi à la discrétion de la tâche généralisée, celle ci peut prendre toutes les mesures possibles afin d'éviter les conséquences graves d'une suspension d'activité.

En place P_5 les ordres **AF** et **FP** ont le même effet qu'en place P_3.

N.B.:
Le réseau de Petri décrit figure II.21 a été légèrement simplifié, et ce, sur trois points:

- Les ordres arrivant mal à propos (Ex: ordre **GO** arrivant alors que la tâche généralisée est en place P_1, ...) déclenchent une procédure standard, non réprésentée dans le schéma, qui renvoie un code d'erreur au niveau appelant.
- Quand la tâche généralisée est activée (place P_4 marquée) tous les ordres qui arrivent sont mémorisés par ordre d'ancienneté et ne prendront effet que lorsque la tâche généralisée aura quitté la place P_4.
- Enfin, un "ordre" n'est pas représenté sur le réseau, car en fait, il s'agit d'une demande d'information sur l'état de la tâche plutôt que d'un ordre (ces derniers entraînent une évolution du marquage du réseau). L'"ordre" **FN?** est normalement acquitté immédiatement, quel que soit l'état de la tâche généralisée, sauf si elle se trouve en place P_3, P_4 et P_5; dans ces cas là, l'ordre est mémorisé et ne sera acquitté que lorsque la tâche repassera en place P_1 ou P_2. Cet ordre est surtout utilisé par une tâche généralisée de synchronisation: après envoi d'un ordre **GO** à une tâche fille, une mère peut en suivre l'évolution grâce à l'envoi d'un ordre **FN?** avec ou sans attente; en fonction de la réponse à l'ordre **FN?** la tâche mère pourra activer ou non d'autres filles.

Comme annoncé précédemment, une tâche peut accepter tout ou partie des différents ordres avec la restriction suivante: l'acceptation de l'ordre **AF**

n'est pas forcément vraie); il en est de même pour l'ordre **FP** avec l'ordre **GO** et les ordres **CONT** et **SUSP**.

De façon générale, chaque fois qu'un ordre reçu est traité, un compte-rendu est renvoyé à la tâche émettrice.

II.3.3.2.4 Propagation de certains ordres

Considérons une tâche généralisée de synchronisation TS_i qui possède des filles, qui peuvent à leur tour en posséder d'autres.

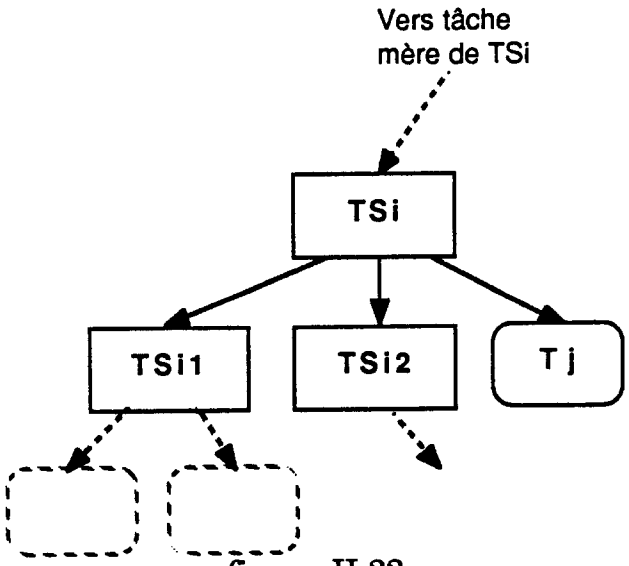


figure II.22

La tâche TS_i reçoit un ordre de sa tâche mère; si cet ordre est: **AF**, **FP**, **SUSP** ou **CONT**, il présente la propriété d'être propageable vers le bas, à savoir vers les tâches filles: à chaque fois qu'une tâche reçoit un tel ordre, elle le transmet à toutes ses filles du moment, et elle attend que toutes ses filles lui aient rendu le compte-rendu d'acquittement de cet ordre avant qu'elle même l'acquitte vers le niveau supérieur. Normalement, la propagation d'un ordre se fait jusqu'aux feuilles de l'arbre des tâches, sauf si, entre la tâche émettrice et une feuille, existe une tâche n'acceptant pas le dit ordre ou qui se trouve dans un état tel que l'ordre soit mal à propos (Ex: réception d'un ordre **CONT** alors qu'aucun ordre **SUSP** n'a été reçu, ...); dans ces cas, une telle tâche renverra néanmoins un compte-rendu signifiant que l'ordre reçu n'est pas connu; ce compte-rendu sera quand même considéré comme une réponse de manière à ne pas bloquer la propagation des réponses vers le haut de l'arbre.

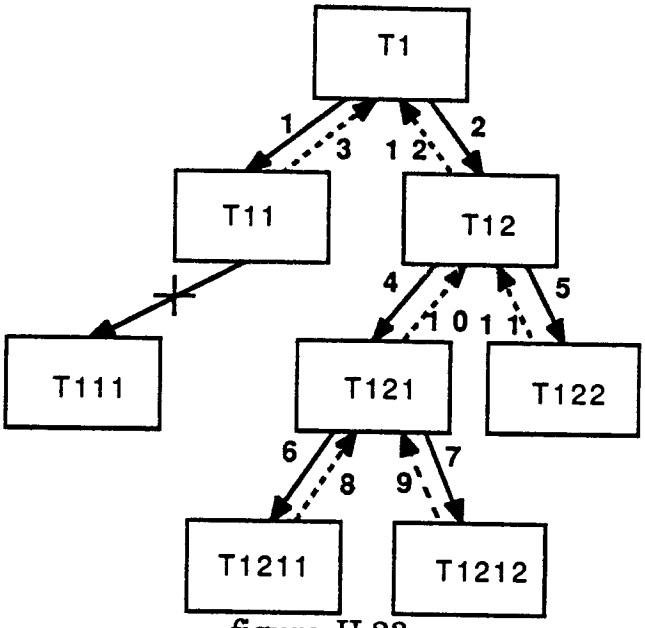


figure II.23

Dans cet exemple, la tâche T1 reçoit un ordre propageable et le propage vers ses filles T11 et T12 (flèches **1** et **2**); la tâche T11 n'accepte pas cet ordre et renvoie immédiatement un compte-rendu ordre non accepté (flèche **3**), elle ne propage donc rien vers sa fille T111; lors de la réception de cet ordre, T12, qui l'accepte, le propage vers ses filles T121 et T122 et ainsi de suite jusqu'aux feuilles de l'arbre. Chaque fois qu'une mère (T121 par exemple) a reçu l'accusé de réception de toutes ses filles (flèches **8** et **9**), elle envoie l'accusé de réception de l'ordre vers sa mère (flèche **10** dirigée vers T12), et ainsi de suite jusqu'à la tâche qui a émis l'ordre.

Les deux seuls ordres non propageables sont **INIT** et **GO**, et ce pour la raison suivante: une tâche généralisée de synchronisation n'envoie ces ordres à ses filles que lorsqu'elle se trouve en place P_4 (c'est-à-dire que la tâche mère a déjà reçu du niveau supérieur des ordres **INIT** et **GO** la concernant); on ne se trouve pas dans le cas de propagation expliqué plus haut. En fait, ces ordres constituent des autorisations qui ne sont envoyées que par le niveau immédiatement supérieur en fonction du schéma de synchronisation qu'il doit réaliser; ils font suite à l'arrivée d'un événement impliquant le déclenchement d'une nouvelle tâche (terminaison d'une autre fille, changement d'état d'une variable booléenne, d'un capteur, ...).

II.3.3.3 Structures utilisées

Avant d'analyser le fonctionnement du noyau de gestion des tâches généralisées, il nous a semblé utile de décrire en détail les structures qu'il manipule, et ce pour deux raisons:

- Il est toujours plus facile de comprendre le fonctionnement d'un programme quand on connait les structures de données qu'il manipule.
- De plus, lors de la définition du concept de tâche généralisée nous avons essayé d'une part de définir une interface utilisateur avec le noyau de gestion qui soit la moins contraignante possible pour le développeur d'une application; d'autre part, il était nécessaire de minimiser l'importance relative du code constituant le noyau de gestion des tâches

toutes les tâches généralisées d'un même processeur). Ces deux contraintes ont induit le choix de structures de données; une fois ce choix effectué, la programmation du noyau a été presque automatique.

Même si nous proposons par la suite une implantation du concept de tâche généralisée sous CESAR, les structures décrites ci-dessous peuvent facilement être adaptées de manière à réaliser le portage du noyau de gestion sous un autre exécutif temps réel.

Il est aisé de montrer que le réseau de Petri représentant les évolutions d'une tâche généralisée appartient à une classe bien particulière de réseau de Petri: celle des machines à états; de tels réseaux de Petri sont facilement implantables grâce à des méthodes tabulaires: ici, l'état de la tâche généralisée est résumé dans une table et le noyau de gestion "se contente" de faire évoluer la table (donc l'état de la tâche généralisée) en fonction des différents ordres reçus.

De plus, une telle approche est aisément extensible sous CESAR à n tâches généralisées, grâce à une particularité de certains exécutifs temps réel: il est possible de créer plusieurs tâches qui se partagent le même code exécutable **mais** qui ont des jeux de données différents (cette situation se retrouve aussi dans les systèmes multi-utilisateurs: soient deux usagers d'un système d'exploitation X qui utilisent en même temps un même programme, l'éditeur de texte par exemple; il n'existe en mémoire centrale qu'**une seule** copie du code exécutable de l'éditeur; pourtant chacune des invocations de l'éditeur travaille sur un jeu de données distinct: fichier f1 pour l'utilisateur 1, f2 pour l'utilisateur 2; en termes informatiques on a affaire à des processus qui ont une section texte commune mais des sections données différentes); lors de la création d'une tâche CESAR, l'utilisateur doit donner le point d'entrée de la tâche, lui réserver une pile et éventuellement fournir un pointeur de données; des tâches se partageant le même code exécutable ont en commun le même point d'entrée, mais ont des piles et des pointeurs de données différents. De cette manière, l'implantation du noyau de gestion des tâches généralisées est facilité: sur chaque processeur existera **une seule** image mémoire du noyau (ROMable qui plus est), commune à toutes les tâches généralisées du processeur, la différence entre les tâches généralisées se faisant par le pointeur de données tâche, qui pour chaque tâche, pointera vers la table résumant l'état de cette dernière.

L'interface utilisateur, telle que nous l'avons définie, a été réduite à sa plus simple expression de manière à entraîner le minimum de modifications aux codes des tâches déjà existantes lors de leur portage sous le noyau de gestion des tâches généralisées Pour cela, une tâche généralisée s'écrira patiquement de la même façon qu'avant: il suffit d'ajouter en fin du code source assembleur **un seul** appel à une macro-instruction prédéfinie: **ENDTSK**.

Utilisation de la macro ENDTSK:

<étiquette éventuelle> ENDTSK point_d'entrée,nom_table

Paramètres d'appel:

point_d'entrée est le début du code exécutable de la tâche généralisée. Ce point d'entrée sera mémorisé dans les tables.

nom_table est le nom qui sera attribué à la table d'état de la tâche généralisée. La macro ENDTSK crée la table d'état à la suite du code de

de façon à être utilisable en dehors du fichier source de la tâche généralisée, en particulier, sous le configurateur CNF.

N.B.:Il est impératif d'utiliser la macro ENDTSK en fin du code opératoire de la future tâche généralisée, en effet, cette macro est l'unique interface assembleur avec le noyau de gestion des tâches généralisées; de ce fait elle constitue aussi le "point de retour" vers le noyau et contient la détermination de la transition à tirer après une activation de la tâche généralisée (tir de la transition t_5 ou t_6); ainsi, suivant la valeur de l'octet de poids faible du registre de données D0, il y aura tir de t_6 (Fin normale: D0.B = 0) ou tir de t_5 (Remise en attente de nouvelles conditions d'activation: D0.B ≠ 0).

II.3.3.3.1 Dialogue par file d'ordre

Comme nous l'avons vu plus haut, chaque tâche généralisée possède un canal de communication avec le niveau supérieur; lequel ne connait ses filles que par leur numéro de canal d'entrée. Dans le cas de l'implantation que nous avons effectuée sous CESAR, le canal d'entrée d'une tâche est une file d'ordre; ce dispositif permet en effet (cf § I.2.4.1.3) de recevoir des ordres et de renvoyer des comptes-rendus par le même canal de communication.

Lorsqu'une tâche généralisée émet un ordre vers une ses filles, elle le fait grâce à une requête d'écriture sur la file d'ordre d'entrée de sa fille; les paramètres de la requête utilisée sont décrits ci-dessous:

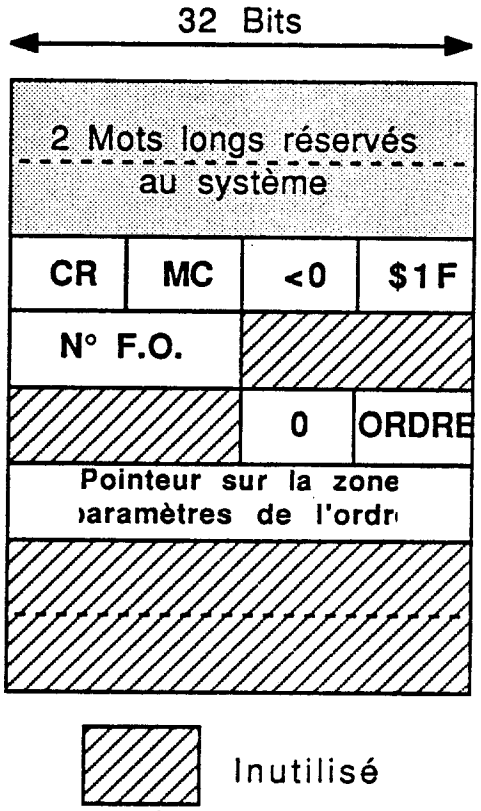


figure II.24

où *numéro* est le numéro de la file d'ordre sur laquelle on écrit.

Les différents ordres sont codés de la manière suivante sur un octet:

FN?	$08
SUSP	$0C
CONT	$10
AF	$14
FP	$18

Ces numéros ont été choisis de manière à servir d'index dans une table où chaque entrée pointe sur la routine de traitement de l'ordre correspondant. En fonction de l'évolution de l'état de la tâche généralisée, la routine de traitement d'un ordre peut varier dans le temps; ainsi, si la tâche généralisée est en place P_1, la routine de traitement de l'ordre **GO** sera une routine standard qui rend un compte rendu d'erreur (ordre arrivé mal à propos), tandis qu'en place P_2, la routine associée à l'ordre **GO** est la procédure P_{GO} permettant le passage de la place P_2 à la place P_3. Cette manière de faire permet un déroutement rapide vers la routine associée à chaque ordre à un moment donné.

Seuls les ordres **INIT** et **GO** peuvent avoir des paramètres; dans le cas de l'ordre **INIT,** ce sont les ressources de cadencement et de données dont la fille aura besoin pour être activée.

Le champ suivant l'ordre permet, si besoin est, de spécifier l'adresse de la zone mémoire qui contiendra les paramètres.

Pour l'ordre **GO** la structure de cette zone est éminemment variable et définie par l'utilisateur en fonction de ses besoins.

Pour l'ordre **INIT**, cette structure est figée:

Plutôt que de passer les numéros des ressources de cadencement et de données dont aura besoin la tâche généralisée, nous avons préféré passer l'adresse de blocs requêtes de connexion aux dites ressources de cadencement et de blocs requêtes de lecture sur les ressources de données. Cette approche permet de simplifier le code de gestion de la tâche généralisée et de conserver des blocs requêtes configurés une fois pour toutes, sans avoir à les rendre systématiquement à la banque.

Le format le plus général de cette zone est le suivant:

(1) 1 Octet négatif pour indiquer une suite de ressources de cadencement.

(2) Le nombre de ressources de ce type sur un octet

(3) Pour chaque ressource, l'adresse du bloc requête correspondant (un mot long)

(4) 1 Octet positif pour indiquer une suite de ressources de données.

(5) Le nombre de ressources de ce type sur un octet

(6) Pour chaque ressource, l'adresse du bloc requête correspondant.

Si la tâche n'utilise pas de ressources de cadencement, il faut créer une table uniquement consituée de la suite des champs (4), (5) et (6).

Si la tâche n'utilise pas de ressources de données, il faut créer une table constituée.des champs (1), (2) et (3) et terminée par un octet négatif.

Si la tâche n'utilise ni ressources de données, ni ressources de

Rappelons enfin, que du fait de l'utilisation d'une file d'ordre comme canal d'entrée d'une tâche généralisée, tous les ordres seront sérialisés et traités par ordre d'ancienneté du fait du mécanisme de fonctionnement interne d'une file d'ordre.

II.3.3.3.2 Table d'état d'une tâche généralisée:

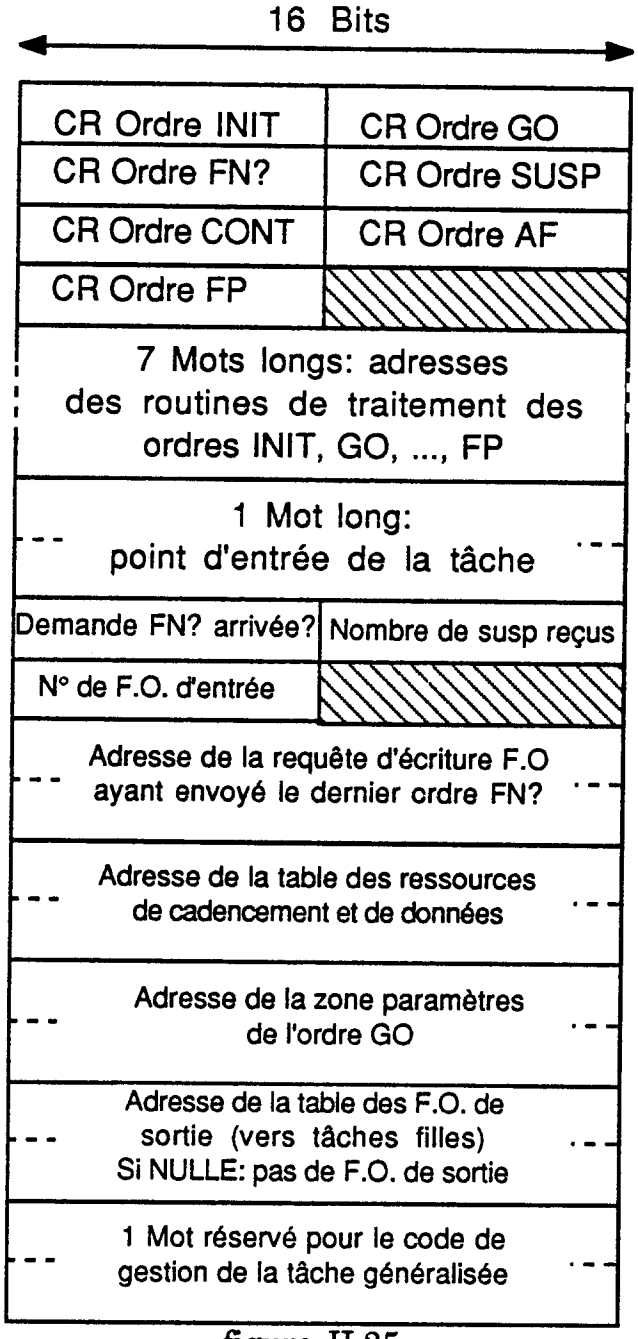


figure II.25

Les sept premiers octets de la table constituent les comptes rendus des différents ordres possibles: pour chaque ordre (sauf **FN?** toujours accepté), l'octet peut prendre deux valeurs:

- Soit la même valeur que l'ordre (valeur positive), auquel cas, il indique ainsi que l'ordre est acceptable.

La table contient ensuite les adresses des routines associées au sept différents ordres: chaque entrée, à un moment donné, contient l'adresse de la routine qui serait activée si l'ordre correspondant était reçu.. Cette routine peut être soit une des procédures associée au tir d'une transition du réseau de Petri, soit la routine standard qui renvoie le code de retour "ordre inconnu" dans le cas de l'arrivée d'un ordre non accepté ou mal à propos.

C'est cette zone qui résume plus particulièrement l'état de la tâche: à chaque tir d'une transition, la routine associée à l'arrivée d'un ordre peut être changée.

Vient ensuite le point d'entrée de la tâche généralisée: c'est l'adresse de début du code qui sera exécuté lors de l'arrivée de toutes les conditions d'activation de la tâche (tir de la transition t_4).

Le champ suivant est un compteur indiquant le nombre d'ordres **SUSP** non encore annulés par un ordre **CONT**. Ce champ est indispensable dans le cas de synchronisation complexes: arrivée de plusieurs ordres **SUSP**. Les deux exemples suivants illustrent les différents cas possibles; dans tous les cas T_1 posséde une fille T_2 qui possède elle-même une fille T_3.

Premier Exemple:

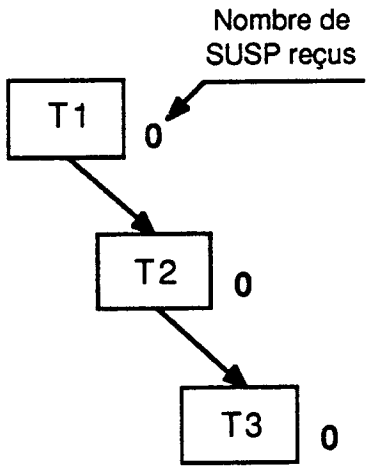


figure II.26

La tâche T_1 envoie un ordre **SUSP** qui se propage vers le bas, l'état suivant devient donc:

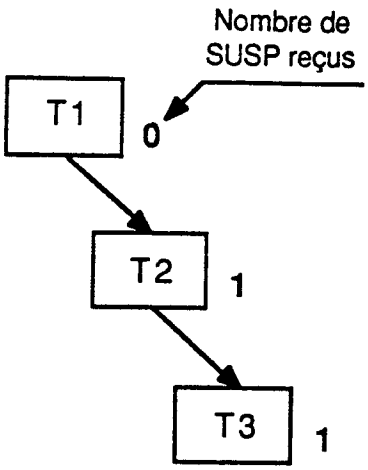


figure II.27

Le résultat de l'envoi de l'ordre **SUSP** est que T_2 et T_3 sont arrêtées momentanément. Lorsque T_1 enverra un ordre **CONT**, les deux tâches arrêtées devront redémarrer simultanément.

Modifions maintenant l'exemple précédent en ajoutant un ordre **SUSP** supplémentaire provenant de la tâche T_2.

Deuxième Exemple:

Etat initial: comme en figure II.26

Suite à l'arrivée d'un événement quelconque, T_2 décide de susprendre provisoirement l'exécution de sa fille T_3.

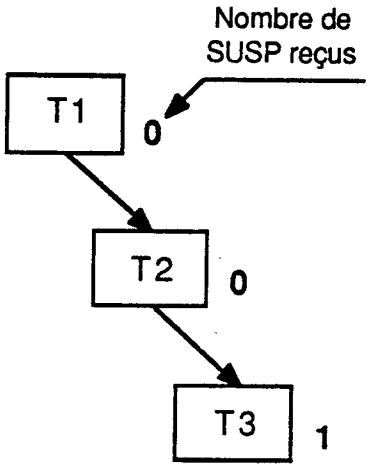


figure II.28

Peu après T_1 décide d'en faire autant avec T_2, ce qui nous donne la situation suivante (l'ordre **SUSP** étant propagé vers le bas):

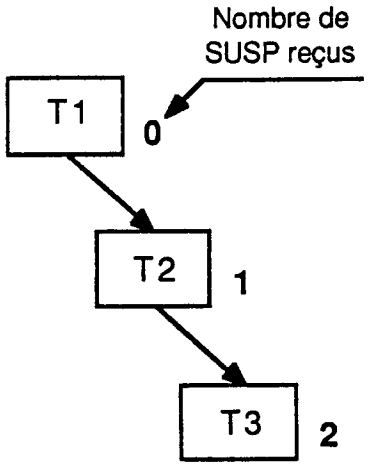


figure II.29

Ensuite, du fait d'un autre événement, T_1 envoie un ordre **CONT** (propagé lui aussi vers le bas), le compteur d'ordres **SUSP** non encore annulés est décrémenté pour les tâches T_2 (il passe alors à 0) et T_3 (il passe alors à 1); le compteur de T_2 étant redevenu nul, cette dernière est autorisée à reprendre son activité (nous retrouvons le cas de la figure II.28); au cours de son activité T_2 pourra envoyer un ordre **CONT** vers T_3 qui pourra ainsi redémarrer aussitôt.

En résumé, quand une tâche généralisée est en place P_5 (en attente d'un ordre **CONT**), chaque nouvelle arrivée d'un ordre **SUSP** incrémente le compteur, chaque arrivée d'un ordre **CONT** le décrémentant; le tir de t_{10} (passage de P_5 à P_3) ne sera possible que lorsque le compteur retombera à zéro.

Le numéro de la fille d'ordre d'entrée de la tâche généralisée est sauvegardé, sur un octet, dans le champ suivant; il est affecté à la configuration, lors de la création de la tâche généralisée. C'est une constante de la tâche généralisée; en pratique, une tâche généralisée n'est connue que par son numéro de file d'ordre d'entrée.

Comme nous l'avons vu précédemment, lorsqu'une tâche est en place P_3, P_4 ou P_5, tout ordre **FN?** arrivant à ce moment ne sera acquitté que lorsque la tâche généralisée repassera en place P_1 ou P_2. A cet effet, il est nécessaire de mémoriser si un ordre **FN?** non encore acquitté existe: c'est le rôle du champ suivant dans lequel est enregistrée l'adresse du bloc requête d'écriture file d'ordre correspondant au dernier ordre **FN?** reçu. Si le contenu de champ est nul, c'est qu'il ne subiste aucun ordre **FN?** non encore acquitté.

Le champ suivant est l'adresse de la table des ressources de cadencement et de données de la tâche généralisée. Cette adresse est passée comme paramètre de l'ordre **INIT** et est mémorisée dans cette zone. Il est à noter qu'à la réception de l'ordre **INIT**, les connexions aux ressources de cadencement se font sans génération d'événement afin d'empêcher l'arrivée d'événements "parasites" tant que la tâche généralisée n'a pas encore reçu l'ordre de démarrer.

Le champ suivant a un rôle équivalent: il mémorise l'adresse de la zone-paramètre de l'ordre **GO**.

Enfin, le dernier champ est un pointeur vers la table des files d'ordres de

synchronisation). Si la tâche généralisée n'a pas de files d'ordres de sortie (tâche de calcul), ce pointeur est nul. Pour chaque tâche généralisée, la structure de la table des files de sortie est la même et est décrite ci-dessous:

- Un mot, indiquant le nombre de files d'ordres de sorties.

Pour chaque file de sortie:

- Un mot, qui suivant sa valeur, indique que la tâche généralisée a pu ou n'a pas pu prendre le contrôle sur sa fille dont le numéro de file d'ordre d'entrée est donné dans le champ suivant.
- Numéro CESAR de la file d'ordre sur un mot.
- Un mot long destiné à la sauvegarde du dernier bloc requête d'écriture sur cette file d'ordre de sortie.

Note: Rappelons que certaines tâches généralisées peuvent avoir plusieurs mères potentielles (ces tâches généralisées peuvent alors être considérées comme des ressources logicielles non partageables). Dans ce cas, toute mère doit, avant d'envoyer un ordre vers une de ses filles, s'assurer que cette dernière est libre; à cet effet, nous avons implanté en mémoire globale du système une table de protection des files d'ordres. Cette table est constituée d'autant d'octets qu'il y a de ressources possibles (paramètre défini à la création des tables système de CESAR); elle est "parallèle" à la suite des descripteurs de ressources CESAR: c'est-à-dire que le premier octet de la table indique l'état de la première ressource CESAR et ainsi de suite; pour les ressources qui ne sont pas des files d'ordres, cet octet ne sert à rien. Initialement, cette table contient uniqement des octets nuls indiquant ainsi que toutes les ressources sont libres. Afin de prendre le contrôle sur un de ses filles, une tâche généralisée doit s'assurer que sa fille est libre (octet correspondant à la file d'ordre nul) puis mettre une valeur non nulle dans l'octet, indiquant ainsi que la fille est dorénavant indisponible. En contexte multi-processeurs, il est impératif que la suite TEST de l'octet puis AFFECTATION d'une valeur soit **INDIVISIBLE**; seule l'instruction **TAS** (Test And Set) du 68000 garantit cette condition. Quand une mère n'a plus d'ordres à envoyer à l'une de ses filles, elle doit remettre l'octet de verrouillage de sa fille à zéro. L'indicateur local de prise de contrôle d'une fille est le résultat de l'opération TAS sur l'octet de verouillage; ainsi, à tout moment, une mère peut déterminer quelles sont ses filles qui sont sous son contrôle. De manière à rendre les logiciels les plus transparents possible, une tâche généralisée ne doit pas connaître le numéro CESAR des files d'ordres d'entrée des ses filles (ces numéros sont en général attribués lors de la phase de configuration du système); ainsi, une tâche généralisée référencera en général ses filles par un numéro qui constitue un index dans sa table des files d'ordres de sortie: il suffit, connaissant le numéro d'index d'une des filles, d'une simple indirection dans la table des files d'ordres pour en connaître son numéro de file d'ordre (cette approche est celle qui nous semble la plus satisfaisante: il est en effet, plus parlant de dire qu'une tâche généralisée s'adresse à sa $n^{ième}$ fille que de dire qu'elle s'adresse à sa fille dont le numéro de file d'ordre d'entrée est x). Afin de faciliter ces opérations de mise à jour des tables, nous avons mis à la disposition de l'utilisateur deux macro-instructions: **GETFO** et **LIBFO** qui permettent respectivement de tenter de prendre le contrôle sur une fille et de la libérer; ces deux macro-instructions n'ont qu'un seul paramètre: le numéro d'index de la fille.

II.3.4 Fonctionnement du noyau de gestion des tâches généralisées

II.3.4.1 Préliminaires

Le noyau de gestion des tâches généralisées a été écrit en assembleur 68000. Nul n'ignore que ce "langage" qui permet d'atteindre des performances optimales du point de vue rapidité d'exécution, n'est pas réputé pour sa lisibilité. Cependant, l'utilisation de macros-instructions génériques et surtout l'emploi de méthodes tabulaires particulièrement bien adaptées à la programmation de l'automate que constitue notre noyau de gestion, nous ont permis d'améliorer grandement la lisibilité du code source du noyau.

Pendant toute l'analyse du fonctionnement du noyau nous ne nous préocuperons pas de l'état initial des tables (ordres acceptés, file d'ordre d'entrée de la tâche, etc ...). Comme nous le verrons par la suite, dans le paragraphe consacré à l'utilisation pratique du concept de tâche généralisée, ces tables sont initialisées lors de la phase de configuration du système CESAR.

II.3.4.2 Principes généraux

Hormis l'initialisation de son environement de travail, le noyau de gestion est uniquement constitué d'une boucle principale dont la structure est très simple:

```
Répéter
    Lecture avec attente de la file d'ordre d'entrée
    En fonction de l'ordre reçu aiguillage vers la routine de traitement
    de l'ordre
Pour Toujours
```

figure II.30

Lors de la réception d'un ordre, l'aiguillage vers sa routine de traitement est simplifié par l'utilisation d'une table de saut, contenant pour chaque ordre, l'adresse de la routine de traitement associée. Toute routine de traitement peut, lorsqu'elle détecte le franchissement d'une des transitions valides du réseau de Petri de la figure II.21, modifier le contenu de cette table, et ainsi associer de nouveaux traitements aux ordres à arriver. Le contenu de cette table est la représentation algorithmique du réseau de Petri et de son marquage.

II.3.4.3 Utilisation des logiciels développés

Afin de faciliter tant la programmation de tâches généralisées de synchronisation que la configuration de l'ensemble des tâches généralisées (création des liens de filiation, déclaration des ordres acceptés, ...), nous avons défini des jeux de macro-instructions d'une part et des fichiers de configuration d'autre part répondant aux besoins les plus courants en la matière.

II.3.4.3.1 Ecriture d'une tâche généralisée

Nous avons vu précédemment que l'interface avec le noyau de gestion des tâches généralisées était la même pour les deux types de tâches (de synchronisation ou opératoires) et se réduisait a l'appel d'une seule macro-instructions en fin de code

Exemple:

```
TSK1     PROG
DEBCOD   EQU      *          Point d'entrée de la tache
.
.    Corps de la tâche généralisée
.
* Ici D0.B doit être nul ou non suivant qu'il y a demande de
* Fin Normale ou pas.

         ENDTSK   DEBCOD,TB_TSK1
         END
```

Après appel à la macro ENDTSK, l'identificateur TB_TSK1 "pointe" sur la table d'état de la tâche généralisée TSK1. A chaque activation d'une tâche généralisée (marquage de la place P_4), le registre d'adresse A6 pointe vers la table d'état tâche, ce qui permet à chaque tâche généralisée de consulter sa table (en particulier, les champs contenant les numéros CESAR des files d'ordres d'entrée de ses éventuelles filles).

II.3.4.3.2 Configuration de l'application

Dans ce paragraphe nous exposerons une solution basée sur l'utilisation de fichiers de commandes de l'éditeur de liens/chargeur/configurateur CNF. La plupart de ces fichiers de commandes n'effectuent en fait que des écritures dans les différentes tables du système de gestion des tâches généralisées. En phase de développement cette méthode permet de relancer l'application avec une nouvelle configuration par un simple changement de fichier de commandes. Néanmoins, pour toute autre implantation sur un exécutif temps réel non lié à un outil tel que le configurateur, la création de tout l'environnement peut être laissée à la charge d'une routine d'initialisation créant dynamiquement les différentes tâches généralisées puis configurant les différentes tables grâce à une bibliothèque de fonctions (réalisant les mêmes actions que les fichiers de commandes CNF).

La configuration d'une application utilisant les services du noyau de gestion des tâches généralisées passe par 6 étapes consécutives. Certaines étapes font intervenir des fichiers de configuration prédéfinis. Plutôt que d'en décrire l'utilisation de façon abstraite, nous avons préféré donner un exemple que nous commenterons par la suite. Pour cela, reprenons l'exemple de la figure II.18; nous supposerons que TS1, TS2, ModA, ModC et ModD sont portés par le processeur 1, tandis que TS3, ModB et ModE le sont par le processeur 2; en outre, TS1, tâche de synchronisation "principale" n'accepte aucun ordre, TS2 n'accepte que GO, FP, SUSP et CONT, les autres tâches acceptent tous les ordres.

```
*---------- Phase 1 ----------
INISYNC          Init. de la table de protection des f.o.

*---------- Phase 2 ----------
PR 1
DEF .L,.L+256    Réservation des piles tâches
DEF PILE_TS1,.L
```

```
DEF .L,.L+256
DEF PILE_ModA,.L
DEF .L,.L+256
DEF PILE_ModC,.L
DEF .L,.L+256
DEF PILE_ModD,.L
INC CODSYNC,RAM1  Chargement du noyau de gestion
INC TS1           Chargement du code des différentes tâches
INC TS2
INC ModA
INC ModC
INC ModD
DEF RAM1,.L
LIMIT .L,RAM1
LOAD RAM1         Et chargement sur CESAR

*---------- Phase 3 ----------
*                           Création des différentes tâches CESAR

TA TS1,CODSYNC,PILE_TS1,,,,,TAB_TS1
TA TS2,CODSYNC,PILE_TS2,,,,,TAB_TS2
TA ModA,CODSYNC,PILE_ModA,,,,,TAB_ModA
TA ModC,CODSYNC,PILE_ModC,,,,,TAB_ModC
TA ModD,CODSYNC,PILE_ModD,,,,,TAB_ModD
GLOBAL TS1                  Pour utilisation par tous les processeurs
GLOBAL TS2
GLOBAL ModA
GLOBAL ModC
GLOBAL ModD

PR 2
* Phases 2 et 3 analogues aux phases 2 et 3 ci-dessus, mais avec
* TS3, ModB et ModE

*---------- Phase 4 ----------
INIFOS TS1,4                TS1 a quatre filles
INIFOS TS2,2                TS2 a deux filles
INIFOS TS3,2                Ainsi que TS3

*---------- Phase 5 ----------
SYNC TS1,TS2,FO_TS2         Dialogue TS1 -> TS2 par FO_TS2
SYNC TS1,TS3,FO_TS3         ...
SYNC TS1,ModA,FO_ModA
SYNC TS1,ModB,FO_ModB
SYNC TS2,ModC,FO_ModC
SYNC TS2,ModD,FO_ModD
SYNC TS3,ModD,FO_ModD
SYNC TS3,ModE,FO_ModE

*---------- Phase 6 ----------
*                           Déclaration des ordres acceptés
ACCORD TS1,,,,,,
ACCORD TS2,,GO,SUSP,CONT,,FP
ACCORD TS3,INIT,GO,SUSP,CONT,AF,FP
```

ACCORD ModE,INIT,GO,SUSP,CONT,AF,FP

La **phase 1** crée la table de protection des files d'ordres. Pour cela, le fichier de configuration INISYNC consulte la variable système indiquant le nombre maximum de ressources créables, soit *n* ce nombre; il réserve ensuite en mémoire globale une zone de *n* octets initialisés à zéro; ainsi, chaque file d'ordre créée par la suite sera considérée comme libre.

La **phase 2** consiste en la réservation des piles pour chaque tâche suivie du chargement du code des dites tâches. Pour chaque processeur portant des tâches généralisées, il est aussi nécessaire de charger le code exécutable du noyau de gestion des tâches généralisées (inclusion du fichier objet CODSYNC.O68, si ce dernier n'a pas été ROMé en même temps que le système).

Lors de la **phase 3**, il y a création des différentes tâches. Il est à noter que toutes les tâches ont comme point d'entrée de début du noyau de gestion des tâches généralisées (CODSYNC). Chaque tâche généralisée a cependant sa propre pile et sa propre table d'état (le nom de la table d'état est passé en dernier paramètre de la commande TAsk). Rappelons que plus une tâche est haut dans l'arborescence, plus sa priorité doit être grande; cette contrainte est respectée dans l'exemple, car en l'abscence de déclaration de priorité, CESAR attribue les priorités par ordre décroissant.

Les phases 2 et 3 sont à répéter pour chaque processeur portant au moins une tâche généralisée.

Pour chaque tâche possédant des filles (TS1, TS2 et TS3 dans notre exemple), le fichier de configuration INIFOS permet, lors de la **phase 4**, de réserver la table des files d'ordres de sortie d'une tâche généralisée et de créer le lien entre la table d'état tâche et la table des files d'ordres de sortie. Ce fichier de configuration nécessite un argument: le nombre de filles que la tâche a, il permet de dimensionner la table des files d'ordres (il faut, en effet huit octets par file d'ordre).

La **phase 5** permet la création effective des liens de filliation entre les tâches généralisées. Pour créer un tel lien, il faut faire appel au fichier de configuration SYNC:

Utilisation:
SYNC mère,fille,fo_fille

Chaque appel à SYNC crée un lien entre deux tâches généralisées. L'action du fichier de configuration SYNC peut se décomposer en 3 étapes:

- Création de la ressource file d'ordre CESAR de numéro *fo_fille*.
- Ce numéro est enregistré dans le champ file d'ordre d'entrée de la tâche généralisée de numéro *fille*.
- Enfin, ce numéro est enregistré dans la table des files d'ordres de sortie de la tâche généralisée de numéro *mère*.

La dernière phase (**phase 6**), permet par l'appel au fichier de configuration ACCORD (ACCepte ORDre) de définir pour chaque tâche généralisée les ordres qu'elle accepte. Ce fichier admet 7 arguments

et FP, si un ordre est accepté, le mot clé correspondant doit être présent, sinon, le mot clé doit être absent. L'ordre des mots clés est celui de la liste ci-dessus et est **impératif**, ainsi pour une tâche acceptant les ordres GO, SUSP, CONT et FP:

ACCORD Ti,,GO,SUSP,CONT,,FP est valable
tandis que
ACCORD Ti,GO,SUSP,CONT,FP ou
ACCORD Ti,,SUSP,GO,FP,,CONT ne le sont pas.

II.4 Intéret respectif des deux logiciels et exemples

Nous avons présenté dans ce chapitre deux concepts qui ont débouché sur la réalisation de deux logiciels. Même s'il est tout à fait possible d'utiliser ces deux logiciels de façon indépendante, en pratique, comme nous le prouverons les exemples qui suivent, l'utilisation du concept de tâche généralisée implique celle du "compilateur d'expressions logiques". En effet, l'écriture de tâches de synchronisation peut amener à envoyer de façon non séquentielle des ordres d'activation vers des tâches filles et d'en attendre les comptes-rendus suivant un schéma pas forcément séquentiel. Nous illustrerons notre propos par deux exemples de synchronisations simples mais d'utilisation fréquente en robotique.

II.4.1 Notations utilisées

Afin de rendre les exemples plus concis et plus parlants, nous les présenterons sous forme d'un "pseudo-langage" déjà utilisé précédemment.

D'autre part, pour les tâches généralisées de synchronisation, nous noterons de la façon suivante l'envoi d'un ordre vers une tâche fille:

ORDRE(tâche_destinataire)

Par défaut, nous considérerons que les ordres sont envoyés (par écriture sur la file d'ordre d'entrée de la fille) sans attente du compte-rendu. Pour spécifier le contraire, nous ajouterons "att" après l'ordre.

Exemples:

GO (TS2) att
FN? (ModA)

Par:
variable = ORDRE(tâche_destinataire)
nous indiquerons que *variable* se voit affecter l'adresse du bloc requête d'écriture file d'ordre correspondant à l'envoi de l'ordre.

Exemple:
req1 = GO(TS2) att

Enfin, nous omettrons les deux caractères $* devant les instructions *$WAIT et *$IF du "compilateur d'expressions logiques" et plutôt que d'utiliser des étiquettes après le champ THEN de l'instruction *$IF, nous emploierons

Exemples:
IF expression THEN action
ou
IF expression THEN
.
. actions
.
FinIF

II.4.2 Tâche gardée

Soit à réaliser un déplacement gardé d'un robot: nous désirons ordonner un déplacement à un robot; cependant, comme l'espace de travail est encombré, une collision avec un obstacle est possible; ayant équipé le robot d'un capteur d'effort, nous voulons arrêter le déplacement sur détection d'un effort supérieur à une valeur prédéterminée.

Pour cela, nous disposons de deux tâches généralisées Td réalisant le déplacement et Tg réalisant la garde en effort.

Td est une tâche cadencée tous les Δt, à chaque activation, elle exécute la suite d'actions suivantes:

```
Si (position courante = position finale) Alors
      FN
Sinon
      calcul de la nouvelle position de consigne et
      envoi aux actionneurs
FinSI
```

figure II.31

Pour son implantation sous forme de tâche généralisée, cette tâche ne nécessite qu'une seule ressource de cadencement: un timer de base Δt; comme Td doit connaître le numéro de ce timer pour fonctionner, elle doit accepter l'ordre INIT; il n'y a pas lieu de changer les ressources de cadencement de Td, elle n'accepte donc pas l'ordre AF. L'ordre GO sera bien sur accepté de manière à pouvoir activer un déplacement du robot depuis un niveau supérieur, le paramètre de cet ordre sera tout simplement la position finale du déplacement. Il est nécessaire que Td accepte l'ordre FP pour pouvoir interrompre un déplacement suite à la détection d'un effort trop grand. Les ordres SUSP et CONT inutiles dans cet exemple ne seront pas acceptés.

La tâche de garde Tg suit exactement le même principe, à part qu'elle est activée tous les $\Delta t'$ et que le paramètre de l'ordre GO est ici la valeur de l'effort qui déclenchera la garde. Le corps de la tâche Tg peut donc s'écrire:

```
Acquisition du capteur d'effort
Si (effort > effort max) Alors
      FN
FinSi
```

figure II.32

Nous supposerons que les deux tâches Td et Tg ont reçu, en phase d'initialisation de l'application, l'ordre INIT leur permettant de déterminer leur ressource de cadencement.

Le segment de code permettant de réaliser le déplacement gardé peut donc s'écrire:

```
GO (Tg) att
GO (Td)
Rg = FN? (Tg)
Rd = FN? (Td)
WAIT Rg, Rd
IF  Rg  THEN  FP (Td)
IF  Rd  THEN  FP (Tg)
```

L'envoi de l'ordre GO à la tache de garde est ici fait avec attente, de manière à être sûr que la garde est active avant de lancer le déplacement. L'envoi de l'ordre FN? aux deux tâches, suivi d'une mise en attente d'une réponse permet de se synchroniser sur le déclenchement de la garde ou l'arrêt du déplacement; après quoi, un test permet de déterminer laquelle des deux tâches s'est terminée la première; l'autre tâche est alors arrêtée par l'envoi d'un ordre FP.

N.B.: Les trois dernières instructions du bout de programme ci-dessus peuvent être remplacées par la séquence suivante:

```
WAIT  Rg, Rd
IF  Rg + Rd  THEN
        FP (Td)
        FP (Tg)
FinIF
```

qui permet aussi d'attendre la fin d'une quelconque des deux tâches et de leur envoyer un ordre FP (qui sera tout simplement ignoré par la tâche qui se sera terminée la première).

II.4.3 Autres synchronisations possibles

Reprenons l'exemple précédent, et remplaçons la tâche de garde Tg (que l'on peut considérer comme une "tâche prédicat") par une autre tâche d'action comme Td. Sans rien changer au bout de programme présenté ci-dessus, nous réalisons une synchronisation de type "coordonnée": lancement en parallèle de deux actions, la fin de l'une entraînant celle de l'autre.

Exemple: Après amenée d'un objet porté par un robot dans l'entremords d'un étau à mords parallèles, nous désirons fermer ce dernier, malgré une erreur possible de positionnement suivant l'axe perpendiculaire aux mords.

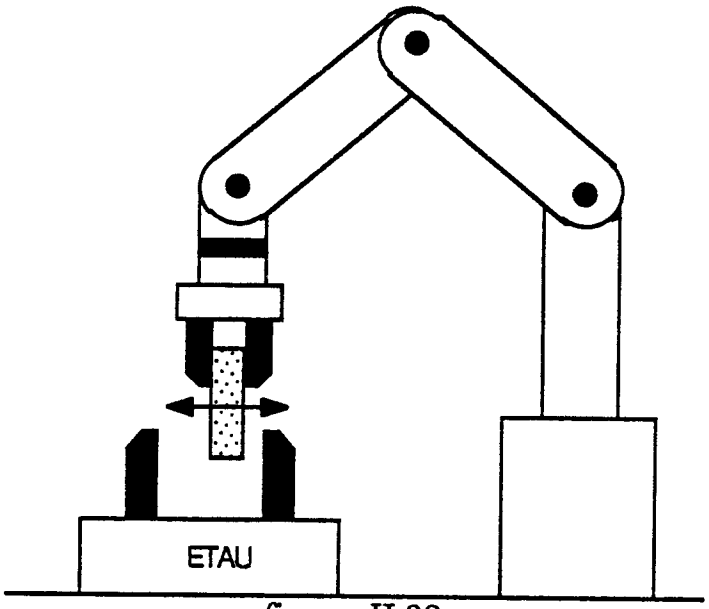


figure II.33

Afin de réaliser cette opération, le robot est equipé d'un capteur d'effort, et nous lancerons en "mode coordonnée" deux tâches: l'une (Tf) qui fermera l'étau, l'autre (Tl) qui libérera le robot en effort suivant l'axe indiqué par la double flèche (tout se passe comme si les asservissements du robot ne fonctionnaient pas suivant cet axe); la fermeture complète de la pince (fin de la tâche Tf) arrêtera alors la tâche Tl (les asservissements du robot conserveront alors la position actuelle du point terminal), la pièce portée par le robot s'autocentrera ainsi dans l'étau.

Avec les logiciels développés, il est de même très facile de lancer en parallèle deux tâches généralisées avant d'exécuter une autre action, comme le montre le bout de programme suivant:

```
GO (T1)
GO (T2)
R1 = FN? (T1)
R2 = FN? (T2)
WAIT  R1, R2
IF  R1.R2  THEN  action
```

II.5 Conclusion

Comme les exemples précédents nous le prouvent, l'utilisation du "compilateur d'expressions logiques" est possible à l'intérieur de tâches généralisées. En particulier, le "compilateur d'expressions logiques" facilite grandement l'écriture d'une tâche généralisée de synchronisation.

A l'heure actuelle, où nous disposons, en plus de l'assembleur, d'un langage de haut niveau (C) pour le développement d'applications temps-réel sous CESAR, un reproche que l'on pourrait formuler à l'égard des deux concepts présentés dans ce chapitre, est leur "relative rusticité": ils impliquent une programmation en assembleur[1].

En revanche, du fait de la programmation en assembleur il est possible d'atteindre une rapidité et une compacité de code optimales. Les défauts reprochés à l'assembleur (programmation non structurée, possibililité de produire des programmes nébuleux, ...) sont en fait gommés par la **structuration** et la lisibilité qu'apportent ces deux logiciels. En effet, le concept de tâche généralisée permet, comme dans les langages de haut niveau, l'analyse descendante et la validation ascendante d'une application, et donc en facilite son développement et sa maintenance.

---

1. En se basant sur les structures de données présentées dans ce chapitre, il serait néanmoins

Une preuve s'il en est:

Pour l'écriture de l'application (dont le scénario est donné ci-dessous) présentant le poignet actif au SITEF[1] 1985, nous avons pleinement utilisé ces concepts.

En s'appuyant sur le niveau réflexe (asservissements hybrides position-force) existant, nous avons modifié les primitives disponibles pour en faire des tâches généralisées, puis écrit et testé l'application en une semaine.

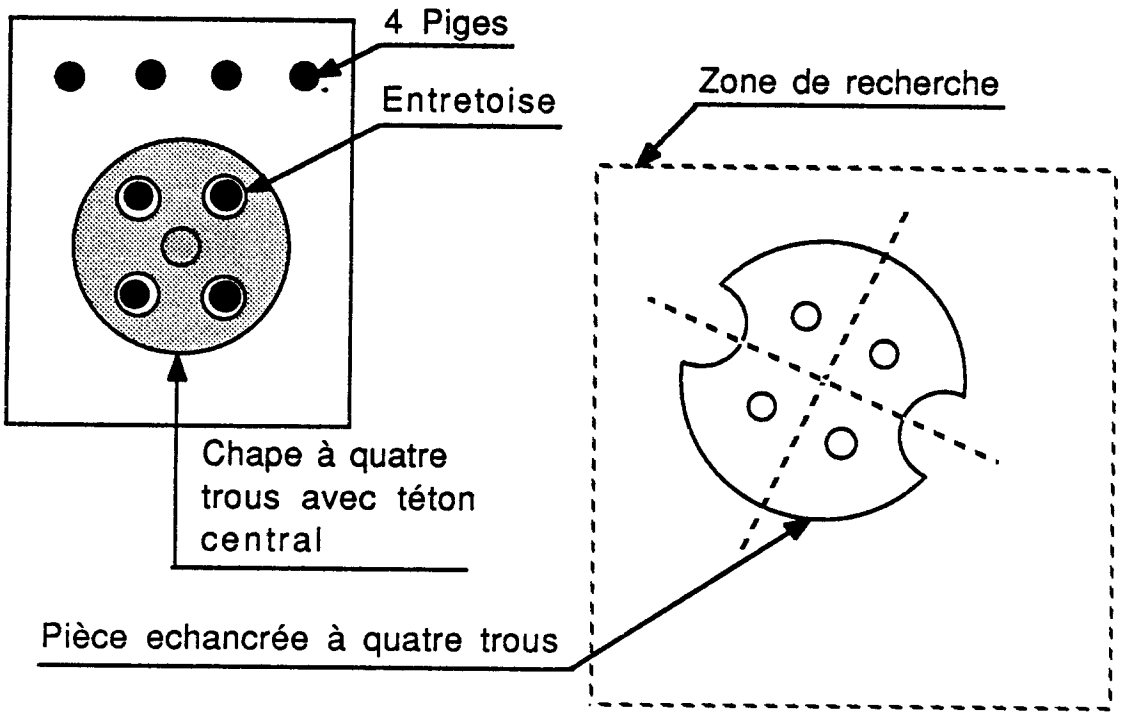


figure II.34

Avant de lancer la démonstation l'opérateur place au hasard la pièce échancrée dans la zone encadrée par un pointillé. Le poignet actif va tout d'abord parcourir cette zone suivant sa médiane verticale (sur ce schéma, correspondant à une vue de dessus), dès qu'il arrive au contact de la pièce, il effectue un suivi de contour de cette dernière de manière à déterminer ses axes de symétrie; ceci fait, la position des quatres trous de la pièce est connue. Le poignet va ensuite chercher une à une les quatres piges dans le magasin et les insère dans les trous de la pièce. A la suite de quoi, il fait de même avec les quatres entretoises pour les déposer autour des quatre piges précédemment insérées. Enfin, il va saisir la chape (maintenue dans le magasin par quatre

piges fixes) par son téton central et va l'insérer sur la pièce au sommet des quatre entretoises. Après un temps d'attente, le poignet va effectuer les opérations inverses de manière à remettre chape, entretoises et piges dans le magasin.

CHAPITRE III
L'ENVIRONNEMENT LOGICIEL LIPS

III. L'ENVIRONNEMENT LOGICIEL LIPS

LIPS ([BACQ87]) est un environnement logiciel qui a été tout d'abord défini en fonction des besoins bien particuliers de la robotique: en effet, la plupart des baies de commande de robot que l'on trouve dans le commerce sont en général très fermées et rajouter un nouveau capteur dans les lois de commande tient souvent de l'exploit. Lors de la définition de l'ensemble LIPS nous avons voulu conserver les "idées fortes" de CESAR, à savoir, modularité dans la puissance de calcul, modes de synchronisation évolués, et sur cette base bâtir un système de commande le plus ouvert possible.

Quelle que soit l'organisation logicielle retenue pour la conception d'une baie de commande de robot, il est toujours possible de distinguer deux niveaux logiques bien particuliers:

- Le niveau réflexe (ou noyau du robot): c'est un processus cadencé rapidement, jamais dormant, de priorité maximale, qui est chargé des asservissements des différents axes du robot, ainsi que de la gestion des sécurités élémentaires (activation du chien de garde, détection des butées, ...); il reçoit des consignes de position ou de vitesse de la part des autres niveaux et s'efforce de les réaliser.
- Le niveau supérieur (éventuellement subdivisé en plusieurs sous-niveaux) comprend un ou plusieurs processus chargés du "pilotage" du robot lors des actions demandant un déplacement du robot et de l'enchaînement des différents mouvements en fonction d'une logique interne ou en réponse à des sollicitations extérieures. En règle générale, le développeur d'une application robotique (à ne pas confondre avec le concepteur du logiciel de commande) a à sa disposition un éventail plus ou moins large d'actions indivisibles de base (recalage, déplacement articulaire, trajectographie cartésienne, ...) qu'il va séquencer et synchroniser de manière à réaliser son application; il n'a pas un accés direct au noyau du robot: ce dernier n'est vu qu'au travers d'une interface comprenant en général en entrée les consignes de position ou de vitesse à appliquer et en sortie un mot contenant l'état du robot. Ces actions de bases qui sont les "atomes" (au sens grec du terme: particule de base, indivisible) de l'application sont en général appelées **primitives**.

Lors de la définition des spécifications de LIPS nous avons voulu réaliser un environnement utilisable à la fois par le **concepteur** de logiciel de commande qui pourra soit créer un noyau et ses primitives *ex nihilo*, soit rajouter des primitives à un noyau existant ou par le **développeur** d'une application qui, à partir des primitives existantes, en assurera le séquencement en vue d'assurer la mission dévolue à son application

III.1 Organisation logicielle de LIPS

Il est possible de considérer le noyau LIPS comme un "sur-système" de l'exécutif temps réel CESAR. LIPS a été tout d'abord articulé autour de deux "services" principaux de ce sur-système: le **superviseur** et le **synchroniseur**. Par la suite un troisième service est venu grossir les rangs: l'**interpréteur CLIPS**.

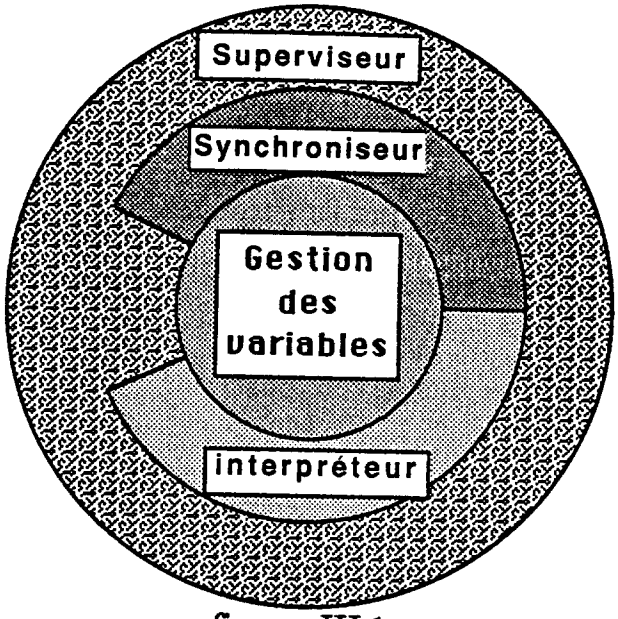


figure III.1

Le superviseur est plus particulièrement chargé du dialogue avec le "monde extérieur" (opérateur, ou calculateur hôte intégrant un rôle de planification): il reçoit en particulier les demandes d'activation de primitives, ou de programmes, gère le chargement et le déchargement de modules en mémoire, et "sous-traite" certaines tâches aux deux autres services.

Chaque fois qu'une requête au superviseur fait intervenir une synchronisation plus complexe qu'une activation simple, le superviseur, après décodage et validation de la requête, la transmet au synchroniseur qui est responsable de l'enchaînement temporel des différentes primitives appelées dans la requête. Enfin, chaque fois qu'il faut enchaîner de A jusqu'à Z les primitives composant une application (répétition de séquences complexes, traitement d'échecs, ...), les services de l'interpréteur deviennent nécessaires. Lorsque nous avons défini les fonctionnalités de l'interpréteur, notre but n'était pas de créer un niéme langage orienté robotique mais plutôt de définir un langage capable de séquencer avec une certaine flexbilité des appels aux primitives de base.

III.1.1 Initialisation de l'environnement LIPS

Lorsque l'environnement LIPS démarre, une seule tâche est créée: la tâche dite centrale. Bien qu'une fois LIPS complètement initialisé, cette tâche soit présente au même titre que les trois autres précédemment citées, nous avons préféré ne pas en parler lors de la description générale de LIPS car son rôle est épisodique. En fait, cette tâche est responsable de deux types d'actions:

- Initialisation de l'environnement complet.
- Enregistrement des nouvelles primitives.

La tâche centrale est responsable de la création dynamique des trois tâches principales composant LIPS. Avant la déclaration effective au système CESAR, chacune des trois futures tâches se voit attribuer une file d'ordre qui lui servira de canal de communication. Sitôt créée, chacune de ces trois tâches initialise son environnement de travail et se met en attente sur sa file d'ordre d'entrée (seule la tâche superviseur suit un chemin légèrement différent: elle se met en attente d'un événement provenant soit de sa file d'ordre, soit de la

dynamique des tâches, la tâche centrale affecte les différentes priorités de manière à ce que le superviseur soit plus prioritaire que le synchroniseur lui même plus prioritaire que l'interpréteur.

III.1.2 Connexion d'une primitive à l'environnement LIPS

Chaque primitive sous LIPS est une tâche CESAR (de manière à pouvoir être activée indépendamment des autres primitives). Toutes les primitives sont connues de l'environnement LIPS par un ou plusieurs mots-clé. LIPS met à la disposition du développeur d'une primitive une interface simple permettant la connexion de la primitive à l'environnement. LIPS distingue trois types de primitives:

- Les primitives de commande qui réalisent une action déterminée (déplacement du robot, commande d'actionneurs péri-robotiques, ...).
- Les primitives logiques qui rendent un résultat VRAI ou FAUX, elles sont surtout utilisées de manière à synchroniser des primitives de commande (activation d'une primitive de commande sur un changement d'état d'une primitive logique, gardes, ...).
- Les primitives d'information qui renvoient seulement un résultat.

Quel que soit son type, chaque primitive doit fournir trois fonctions (au sens **C** du terme) qui serviront d'interface avec LIPS:

- Une fonction d'évaluation: elle est chargée, comme son nom l'indique, d'évaluer les éventuels paramètres passés lors de l'appel à la primitive. Dans le cas de primitives possédant plus d'un mot-clé, cette fonction permet aussi de savoir sous quel "nom" a été invoquée la primitive.
- Une fonction d'exécution: à condition que l'évaluation des paramètres ait été bien effectuée (pas de paramètre erroné), LIPS passe le contrôle à la fonction d'exécution qui est chargée de la réalisation de l'objectif fixé à la primitive. Pour les primitives logiques, la fonction d'exécution est appelée automatiquement toutes les 5ms pendant toute la durée d'activation de la primitive.
- Une fonction d'information: cette fonction renvoie une information; le type d'information (sous forme de chaîne de caractères) est laissé au choix du programmeur: ce peut être la syntaxe d'appel de la primitive par exemple.

Enfin une même primitive peut être connue sous plusieurs noms (ou mots-clé suivant la terminologie LIPS).

Une fois ces définitions données, intéressons nous à la connexion à l'environnement LIPS; elle s'effectue par le biais d'une des trois fonctions suivantes: **cnxcom()**, **cnxeval()** ou **cnxinfo()** suivant qu'il s'agit respectivement d'une primitive de commande, d'évaluation logique ou d'information. Dans les trois cas, les arguments des ces fonctions sont au nombre de deux:

- Un pointeur sur la chaîne de caractères constituant la liste des mots-clé.
- L'adresse d'un tableau contenant les trois points d'entrée des fonctions d'évaluation, d'exécution et d'information. Si une (ou plusieurs) d'entre elle(s) n'est (ne sont) pas utilisée(s), il suffit de remplacer le(s) point(s)

Exemple:
Soit une primitive de commande ayant deux mots-clé mc1 et mc2 et ne possédant pas de fonction d'information. Le code **C** suivant illustre le mode de connexion à LIPS.

```
static const char listcle[ ] = "mc1 mc2";
static const (int *const listfonc[ ])() = {evalfonc, execfonc, NULL};
/* Point d'entrée de la tâche CESAR: prim */

void prim()
{
        /* Connexion à LIPS */
         cnxcom(listcle, listfonc);
         /* Normalement, on ne doit jamais passer ici (après cnxcom) */
}

int evalfonc(...)
{

}

int execfonc(...)
{

}
```

figure III.2

Chacune des trois fonctions cnx... effectue une écriture avec attente sur la f.o. d'entrée de la tâche centrale.

La tâche centrale, ainsi réveillée, attribue alors une file d'ordre d'entrée à la primitive puis l'enregistre dans ses tables; ceci fait elle renvoie le numéro de f.o. attribuée à la primitive qui va dès lors pouvoir se mettre en attente d'un ordre provenant de sa f.o.

Chaque primitive nouvellement créée est enregistrée dans un descripteur; c'est une structure dont les principaux champs sont:

- Un mot de chaînage permettant de lier les différentes structures entre elles. Ainsi, lors de la création d'une primitive, le descripteur de la nouvelle primitive est placé en tête de la liste des primitives.

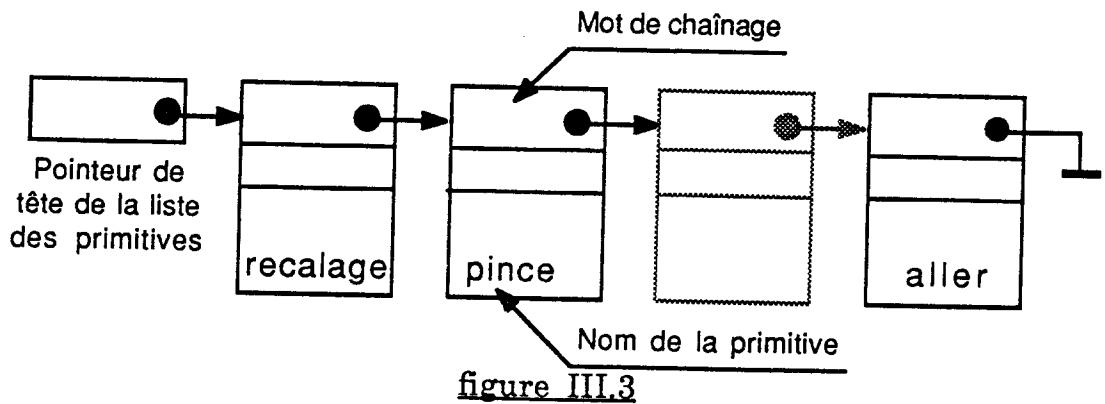


figure III.3

- La liste de ses mots-clés (ou "noms")
- L'adresse du tableau des trois fonctions d'évaluation, d'exécution et d'information.

III.1.3 Généralités sur le traitement des commandes LIPS

Chaque fois que le superviseur reçoit une commande sur sa liaison de dialogue, il analyse le premier mot de la ligne. Du fait de la syntaxe de LIPS, le premier mot d'une ligne de commande sert à déterminer le type de service demandé. Ceci fait, le superviseur scrute une table d'indirection composée d'une liste des mots-clé reconnus: à chaque mot-clé est associée l'adresse de la routine chargée de son traitement. Si le mot-clé n'est pas reconnu, LIPS affiche un message d'erreur; sinon, il se déroute à l'adresse de la routine de traitement du mot-clé. Chacune des différentes routines de traitement est chargée du décodage du reste de la ligne de commande en fonction de la syntaxe propre de chacune des commandes LIPS.

Par la suite, nous nous intéresserons tout particulièrement aux commandes concernant le chargement, le "déchargement", la destruction et l'exécution de modules.

III.1.3.1 Organisation des modules en mémoire

Les modules destinés à être interprétés sont chargés en mémoire globale au delà des différentes ressources CESAR. Lors du chargement d'un module, les différents caractères composant le code source du module sont rangés en mémoire à condition que de l'espace mémoire en quantité suffisante soit encore disponible. Si le module est détruit, l'espace mémoire qu'il occupait doit être rendu libre; il est donc nécessaire d'utiliser un mécanisme de gestion mémoire. Plutôt que de concevoir un mécanisme d'allocation dynamique tel qu'il existe en Pascal (fonction **new()**) ou en **C** (fonction **malloc()**)[1], nous avons préféré utiliser un mécanisme beaucoup plus simple analogue dans son principe à l'organisation d'une disquette ou d'un disque dur décomposé en secteurs de taille fixe et comportant une liste des secteurs libres et un répertoire des différents fichiers existant sur le disque; chaque fois qu'un fichier est crée, le système d'exploitation alloue au fichier **n** secteurs où:

$$n = \mathbf{E}\left(\backslash F(\text{Taille fichier}, \text{Taille utile secteur}) \right) + 1$$

avec **E**(): fonction partie entière

Du fait du mécanisme d'allocation et de libération de secteurs pour les fichiers, rien ne garantit que les différents secteurs composant un fichier soient contigus; aussi chaque secteur d'un fichier doit contenir une information indiquant la position du secteur suivant composant le fichier.

La transposition de ce principe pour les modules est relativement simple:

La liste des "blocs modules" (ou secteurs par analogie) libres est constituée par une ressource CESAR d'espace mémoire contenant au départ **m** blocs de 128 octets (m est déterminé au moment de la configuration du système en fonc-

---

[1] Un tel système peut, à la longue, laisser une mémoire complètement morcelée. Il est alors nécessaire de procéder à un recompactage de la mémoire, opération toujours dangereuse en

tion de la mémoire physique disponible); cette ressource va donc servir de réserve de blocs pour le chargement des futurs modules.

Le répertoire (initialement vide) des modules est constitué par une structure chaînée décrite ci dessous:

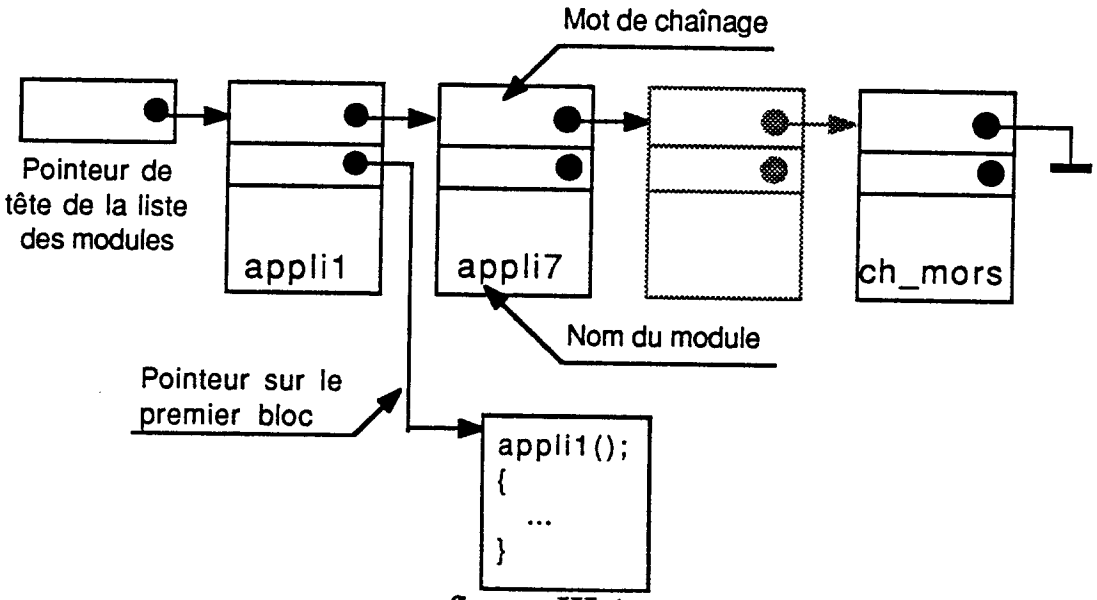


figure III.4

Le pointeur de tête du répertoire à la valeur NULL si aucun module n'est présent en mémoire; sinon, il pointe sur le descripteur du dernier module chargé. Schématiquement (nous verrons au paragraphe III.3.3.1 la structure complète), un descripteur de modules comporte au moins les trois champs suivants nécessaires à la gestion des modules:

- Un mot de chaînage, pointant sur le module suivant dans le répertoire.
- Le nom du module.
- L'adresse du premier bloc composant le module.

Enfin, sur les 128 octets composant un bloc, 120 octets sont effectivement utiles: 4 octets sont réservés en fin de bloc pour indiquer l'adresse du bloc suivant; ces 4 octets sont précédés d'une plage spéciale de 4 octets indiquant à l'analyseur lexicographique que les caractères qui vont suivre ne composent pas le module mais sont en fait un pointeur sur le bloc suivant. Un module occupe donc en mémoire:

n = **E** (\F(Taille module,120)) + 1 blocs, soit au total
n x 128 Octets.

Cette solution de gestion de la mémoire présente un inconvénient: pour de petits modules dont la taille n'est pas un multiple de 120, une portion non négligeable de la mémoire peut être perdue. Cependant ce système est plus facile à mettre en œuvre et ne nécessite pas de recompactage; de plus, chaque fois qu'il faut charger un nouveau module en mémoire, il suffit de consulter le compteur de blocs disponibles pour savoir immédiatement s'il y a assez de place pour cette opération.

La syntaxe de l'ordre de chargement d'un module LIPS est la suivante:

load <nom> <nombre>

où <nom> est le nom du module à charger et <nombre> est le nombre de blocs nécessaires pour le chargement du module. Lorsque le superviseur reçoit une telle commande, il vérifie dans un premier temps qu'il n'existe pas déjà un module de même nom que celui que l'utilisateur désire charger; ensuite, le superviseur vérifie en scrutant le descripteur de la ressource d'espace mémoire qu'il y au moins <nombre> blocs libres. Si ces deux conditions sont satisfaites, LIPS renvoie le message:

0000 Ok ready loading

pour signaler à l'utilisateur qu'il peut commencer à charger le programme; le superviseur demande alors un bloc à la ressource d'espace mémoire, enregistre son adresse dans le descripteur de module et attend alors <nombre> lignes de 120 caractères (à l'exception de la dernière qui peut comporter moins de 120 caractères) terminées par un caractère <CR>[1]. Chaque fois qu'une ligne est lue sur la RS232 de dialogue, le superviseur recopie la ligne dans un bloc module, demande éventuellement un nouveau bloc s'il reste de lignes à lire et fait pointer le bloc qui vient d'être rempli sur ce nouveau bloc. Une fois le chargement terminé, le superviseur envoie le message final d'acquittement:

0000 Ok load

la liaison est alors libre pour une nouvelle commande[2].

III.1.3.1.2 Déchargement d'un module: commande download

Cette commande permet de "récupérer" un module chargé sur la baie LIPS; il s'agit en fait d'obtenir une copie du module, car, après exécution de la commande, le module est toujours résidant dans la baie LIPS.

Dès que le superviseur reçoit une commande du type:

download <nom>

il vérifie que le module nommé <nom> est bien résident en scrutant la liste des descripteurs de modules, puis il explore la chaîne des blocs composant le module en recopiant chacun des blocs sur la liaison RS232 de dialogue. Lorsque le dernier bloc a été transmis, le superviseur envoie le message:

---

1 Il ne peut y avoir de conflit entre le caractère <CR> marquant la fin d'une ligne de 120 caractères à charger et une fin de ligne d'un module LIPS qui est matérialisée par le seul caractère <LF>. Certains systèmes d'exploitation tels que MS-DOS utilisent le couple <LF-CR> comme délimiteur de ligne; dans ce cas, le programme de téléchargement de modules vers la baie LIPS doit filtrer ces <CR> et ne conserver que les <LF>.

2 Cette procédure peut sembler fastidieuse: il faut en effet compter le nombre de blocs de 120 caractères composant le module, puis taper chacune des lignes. En fait, l'environnement LIPS est normalement connecté à un ordinateur hôte sur lequel sont effectués l'édition et la mise au point des modules, des programmes tels que PC-LIPS (©AICO) permettant le chargement des modules (par une commande du type: **load** <nom>) suivant le protocole LIPS sans que

0000 Ok download

indiquant ainsi que la ligne est libre pour une nouvelle commande.

III.1.3.1.3 Exécution d'un module: commandes call et step

Lorsque le superviseur reçoit une commande:

call <module> [([<paramètre> [, <paramètre> ...]])] ou
step <l1> <l2> <module> [([<paramètre> [, <paramètre> ...]])]

il explore la liste des modules à la recherche d'un module de nom <module>; si la recherche est infructueuse, le superviseur en avertit l'utilisateur par le message:

F00C module doesn't exist

dans le cas contraire, le superviseur initialise les champs n1 et n2 du descripteur de module (n1 et n2 sont les numéros de ligne entre lesquelles le mode pas à pas est actif) aux valeurs 0 et 0 dans le cas de la commande **call** ou aux valeurs de <l1> et <l2> dans le cas de la commande **step**. Ceci fait, le superviseur enregistre l'adresse du descripteur de module dans une variable (PGREQ initialement nulle) de façon à mémoriser le fait qu'un module est en cours d'exécution, puis transmet une requête d'exécution à l'interpréteur par le biais d'une écriture sur la file d'ordre d'entrée de l'interpréteur. Cette façon de faire permet de libérer le superviseur qui peut ainsi traiter toute nouvelle commande tandis que l'interpréteur fonctionne alors comme tâche de fond. La requête d'écriture sur la file d'ordre d'entrée de l'interpréteur est configurée comme suit:

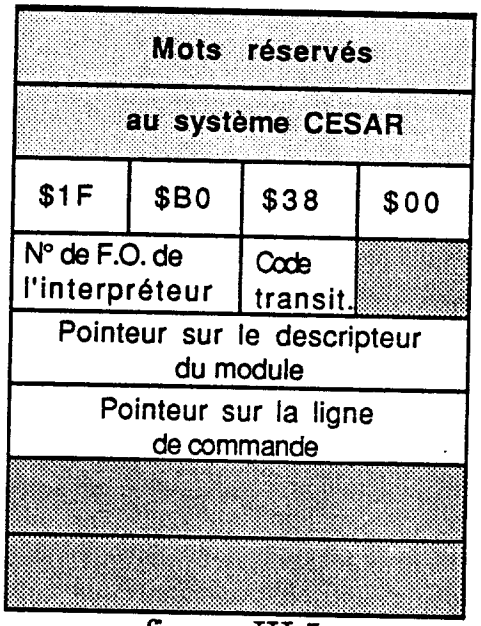


figure III.5

Le code fonction $1F indique un transfert par file d'ordre, le code complémentaire($B0) indique une écriture file d'ordre (car il est négatif) provenant du superviseur, la valeur $10 (code transition) servant à signaler à l'interpréteur qu'il s'agit d'une demande d'exécution (nous verrons plus loin que la tâche interpréteur peut recevoir plusieurs requêtes différenciées par leur code complémentaire).

Les deux pointeurs transmis dans la requête permettent respectivement à l'interpréteur de récupérer les informations concernant le module à exécuter et ses éventuels paramètres contenus dans la ligne de commande.

Une fois l'exécution du module terminée, l'interpréteur effectue une écriture sur la f.o. d'entrée du superviseur en passant un code transition #END ($10) indiquant une fin de module ainsi qu'un pointeur sur le message de retour (soit *0000 Ok <module>* indiquant une exécution correcte du module, soit *XXXX error XX, <module> line yy* en cas d'erreur). Le superviseur ainsi réveillé efface la variable PGREQ pour indiquer qu'aucun module n'est en cours d'exécution puis envoie sur la liaison RS232 de LIPS le message que lui a transmis l'interpréteur.

III.1.3.1.4 Arrêt d'un programme: commande stopg

La commande: **stopg**

permet d'arrêter le module en cours d'exécution.

Dès que le superviseur reçoit la commande **stopg**, il vérifie que la variable PGREQ n'est pas nulle; si tel n'est pas le cas, le superviseur envoie un message d'erreur sur la liaison. Dans le cas contraire, le superviseur effectue une écriture sur la f.o. d'entrée de l'interpréteur avec comme code transition la valeur #STOP ($20) qui signifie demande d'arrêt d'un module. Une fois que l'interpréteur arrive en un endroit où il peut effectivement arrêter l'exécution du module en cours, il le fait et renvoie un compte-rendu de fin d'exécution au superviseur comme dans le cas précédent.

III.1.3.1.5 Reprise de l'exécution d'un programme: commande resume

Chaque fois que l'interpréteur analyse l'instruction **pause** ou qu'il a fini d'analyser une ligne comprise dans l'intervalle où le mode pas à pas est valide (l1 ≤ numéro de ligne courant ≤ l2 en cas de commande **step**) , il écrit sur la f.o. d'entrée du superviseur avec un code transition égal à #STOP ($10) indiquant ainsi que le module en cours est pausé. L'interpréteur passe aussi un pointeur vers le message suivant:

module <nom> paused line xxx

Le superviseur , ainsi réveillé, par l'écriture sur sa f.o. d'entrée recopie le message reçu sur la liaison de dialogue.

Quand ce dernier reçoit la commande **resume**, il effectue l'opération inverse: écriture sur la f.o. de l'interpréteur avec comme code transition la valeur #DEPS ($18) indiquant qu'il faut reprendre l'exécution du module interrompu.

III.2 Fonctionnement du synchroniseur

Chaque fois qu'une commande fait intervenir une synchronisation plus complexe que la simple exécution d'une primitive, le superviseur transmet la commande au synchroniseur.

Ce dernier analyse la ligne de commande et la convertit sous forme d'un

Exemple:
ex paral(when(log1, guard(com1, log2)), seq(com2,coord(com3, com4)))

peut se traduire par l'arbre binaire suivant:

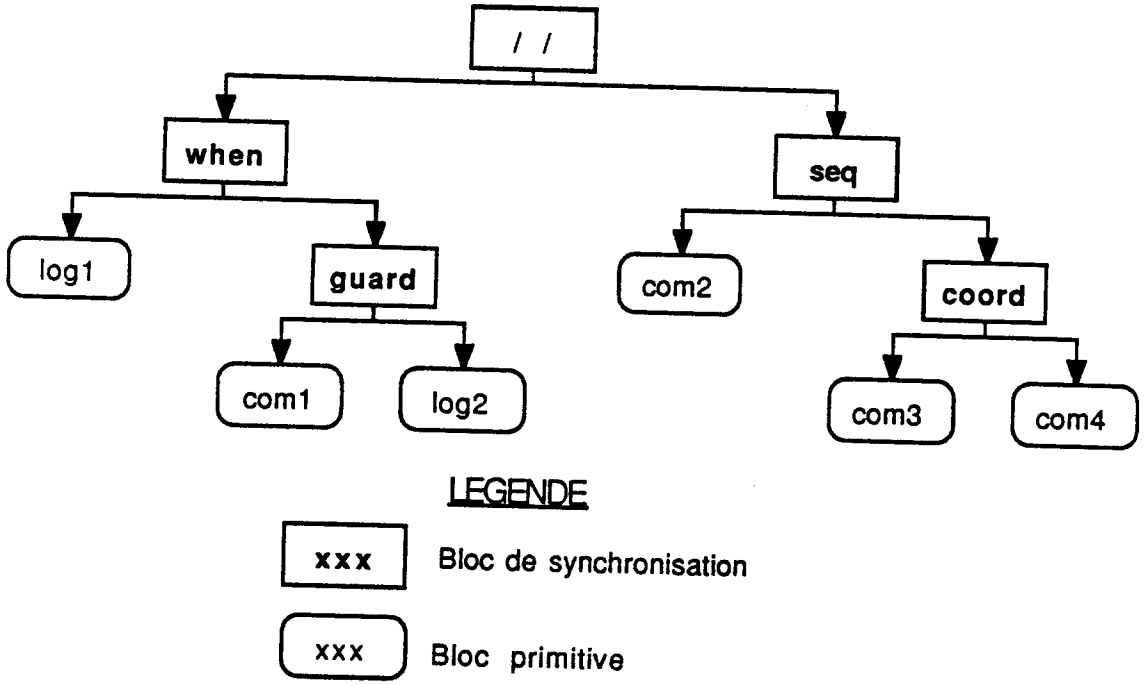


figure III.6

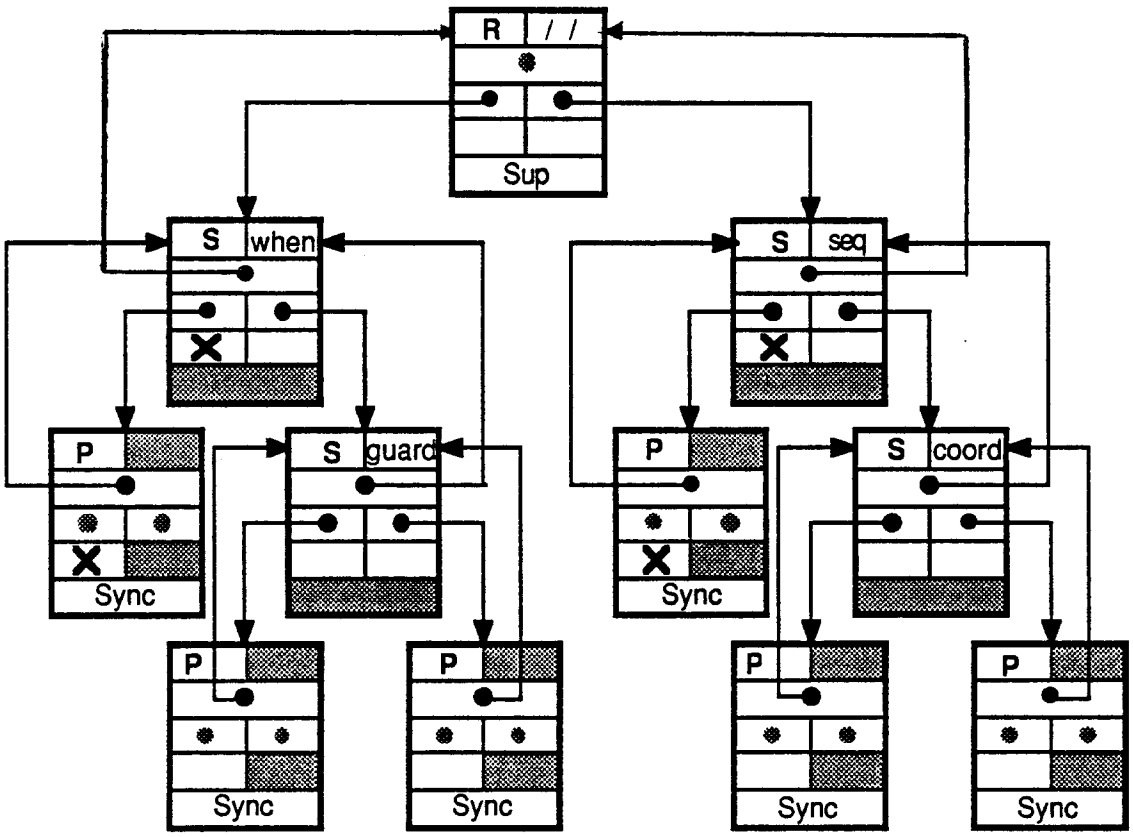
Après analyse de l'expression de synchronisation et vérification que les primitives apparaissant dans l'expression existent bien et ne sont pas déjà en cours d'utilisation, le synchroniseur active les procédures d'interprétation de toutes les primitives; dès que toutes les primitives ont décodé leurs paramètres et à condition que les dits paramètres ne soient pas erronés (si l'une quelconque des primitives détecte une erreur, la commande globale ne sera pas exécutée), le synchroniseur crée de façon interne une représentation informatique de l'arbre représentant la synchronisation. Du fait de la syntaxe choisie, les noeuds de l'arbre ne peuvent être que des blocs de synchronisation et les feuilles des blocs primitives.

Schématiquement, un bloc est composé des champs suivants:

- Nature du bloc (racine, bloc de synchronisation, bloc primitive). Par racine, nous entendons le bloc de synchronisation terminal de l'arbre, c'est-à-dire celui correspondant au premier opérateur de synchronisation de l'expression.
- Type de synchronisation (paral, seq, coord, guard, ...).
- Un pointeur sur le sous-arbre gauche (inutile pour un bloc primitive).
- Un pointeur sur le sous-arbre droit (Id°).
- Un pointeur sur le noeud précédent (NULL pour la racine qui n'a pas de noeud précédent).
- Deux champs indiquant respectivement l'état des sous-arbres gauche et droit (inutile pour un bloc primitive).
- Le numéro de la f.o. sur laquelle il y a lieu d'envoyer le compte-rendu de

primitives, numéro de la f.o. du superviseur pour un bloc racine quand l'ordre provient de la liaison LIPS ou numéro de la f.o. de l'interpréteur si l'ordre provient d'un module).

Exemple:
Reprenons l'exemple précédent. Sa représentation pour le superviseur est:



LEGENDE

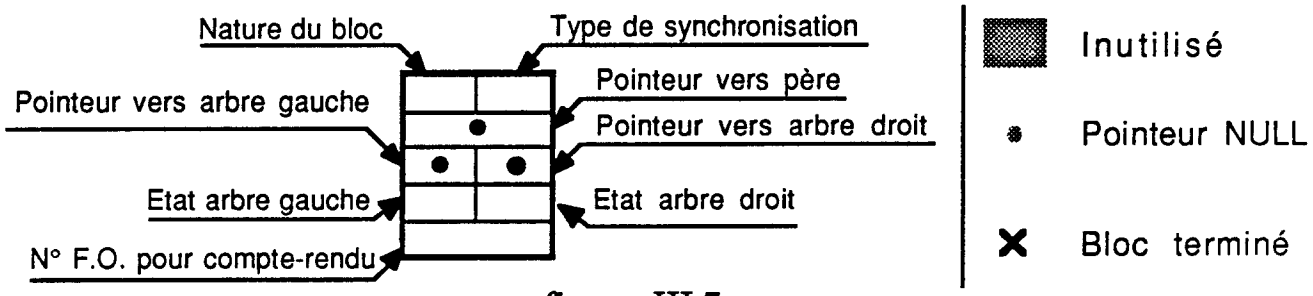


figure III.7

Après création de l'arbre, le synchroniseur part de la racine et active récursivement les blocs qui doivent l'être en fonction de la synchronisation désirée.

Exemple:
Pour un bloc de type **seq** le synchroniseur n'activera dans un premier temps que le sous-arbre gauche et attendra le compte-rendu de fin d'exécution de ce sous-arbre avant d'activer le sous-arbre droit.
Pour un bloc du type **paral**, **coord** ou **guard** le superviseur activera simultanément les deux sous-arbres.

L'exploration se poursuit jusqu'à ce que toutes les feuilles (blocs primitive) des sous-arbres à activer aient été atteintes.

Chaque fois qu'une primitive se termine, elle le signale au synchroniseur au moyen d'une écriture sur la f.o. de ce dernier; l'adresse du bloc de la primitive terminée est passée dans un champ de la requête d'écriture f.o.. Le synchroniseur est alors réveillé et récupère l'adresse du bloc qui vient de se terminer; il remonte alors au bloc père, marque le sous-arbre pointant sur le bloc terminé et en fonction du type du bloc père et de l'état du sous-arbre complémentaire choisit l'une des alternatives suivantes:

- Attendre la terminaison d'un autre bloc. Ex: dans le cas d'une synchronisation du type **paral**; si un des sous arbres se termine et que l'autre n'est pas terminé, il y a lieu d'attendre.
- Marquer le bloc comme terminé puis remonter d'un cran dans l'arbre ou envoyer un compte-rendu au niveau d'appel (superviseur ou synchroniseur) dans le cas où le bloc père est la racine. Ex: même exemple que ci-dessus, mais l'autre sous-arbre est déjà terminé.
- Marquer le bloc comme terminé, désactiver le sous-arbre complémentaire puis remonter d'un cran dans l'arbre. Ex: cas d'une synchronisation de type **coord** ou **guard**: si l'un des deux sous-arbres se termine, il faut désactiver l'autre puis remonter d'un cran dans l'arbre.
- Activer le sous-arbre complémentaire. Ex: cas d'une synchronisation de type **seq**; si le sous-arbre gauche se termine, il faut alors activer le sous-arbre droit.

Ce mécanisme se répercute de façon ascendante dans l'arbre jusqu'à ce que le bloc racine soit marqué comme terminé ou que le superviseur doive se mettre en attente de la terminaison d'un sous-arbre.

III.3 Structures et fonctionnement de l'interpréteur CLIPS

L'interpréteur CLIPS, implanté sous LIPS postérieurement au superviseur et au synchroniseur, a été tout d'abord défini en vue d'automatiser des séquences d'exécution de primitives. La syntaxe d'appel de ces primitives depuis CLIPS a été rendue la plus proche possible decelle du superviseur. Ainsi, si l'on veut automatiser la séquence d'appel suivante:

```
ex recalage(1.5, 0.36, 0)
ex guard(aller("400 400 -10 0 0 0"), effort(z,100))
```

il est possible de réaliser le programme LIPS donné ci dessous:

```
auto();
{
        recalage(1.5, 0.36, 0);
        guard(aller("400 400 -10 0 0 0"), effort(z,100));
}
```

et de lancer son exécution par la commande LIPS: call auto()

Afin d'assurer la flexibilité du séquencement des primitives, CLIPS gère des variables entières avec les opérations classiques (addition, soustraction, multiplication, division, modulo) associées à des structures de contrôle telles que l'alternative (blocs **if ... then ... else**), la sélection multiple (blocs **case**) ou la répétition (boucles **while ... do ...**). Il est ainsi possible de réaliser des programmes simples tels que le suivant:

```
perpete();
{
        recalage(1.5, 0.36, 0);
        while (1) do
        {
                aller("400 400 10 0 0 0");
                aller("0 400 10 0 0 0");
                aller("0 0 10 0 0 0");
                aller("400 0 10 0 0 0");
        }
}
```

décrivant perpétuellement une trajectoire carrée.

CLIPS permet aussi la possibilité de créer de sous-programmes de manière à mieux décomposer la programmation d'une application complexe. Enfin CLIPS utilise la notion de pointeur sur un octet, mot ou mot long machine ainsi que des opérations telles que **et** et **ou** logiques; ces notions permettent, par exemple, de modifier le déroulement d'un programme en fonction de valeurs logiques lues sur un périphérique.

Depuis les dernières versions de LIPS, plus ouvertes aux besoins de la production, il est possible garder des modules CLIPS résidents en mémoire sauvegardée et de lancer leur exécution depuis une boîte à boutons. Cependant en phase de développement la présence d'un calculateur hôte relié à la baie de commande par une ligne RS232 est nécesaire. Le calculateur hôte permet l'édition, le chargement et déchargement de modules vers la baie de commande et grâce à un logiciel séparé (PC-LIPS) la mise au point et le déboggage de modules à la manière d'un outil tel que **dbxtool** présent sur les stations de travail SUN.

III.3.1 Généralités

L'interpréteur CLIPS peut être décomposé en deux entités fonctionnelles distinctes:

- L'interface avec le superviseur.
- L'interpréteur de modules.

Le fonctionnement de l'interface avec le superviseur est fort simple; aussi nous ne nous y étendrons pas outre mesure. L'interface avec le superviseur est constituée surtout de la boucle principale de la tâche interpréteur que l'on peut résumer schématiquement par:

```
Répéter Toujours
    Lecture avec attente sur la f.o. d'entrée
    Si CodeComplémentaire = Exécution Alors
        Interprétation du module
    Sinon
        Rejet
    FinSi
FinRépeter
```

figure III.8

Dans la boucle principale, seules les demandes d'exécution peuvent être prise en compte, car les autres demandes (arrêt, reprise, ...) ne peuvent normalement être émises que si l'interprétation d'un moduleest en cours.

Le reste de l'interface, chargé du traitement des ordres mentionnés ci-dessus est constitué par des fonctions appelées par l'interpréteur de modules. En général, ces fonctions se contentent d'effectuer une scrutation non bloquante de la f.o. d'entrée de la tâche interpréteur afin de déterminer si un de ces ordres n'est pas arrivé depuis que l'interprétation a commencé.

L'interprétation des modules CLIPS est effectuée suivant un mécanisme à deux niveaux que l'on trouve classiquement dans les compilateurs et interpréteurs ([GALL 77]): analyse lexicographique suivie d'une analyse syntaxique (ou analyse sémantique).

Le rôle de l'analyseur lexicographique est d'explorer, caractère par caractère, le fichier (cas le plus général) ou la zone mémoire dans notre cas, contenant le programme, et de regrouper ces caractères en "paquets" ou éléments terminaux de la syntaxe suivant des règles prédéfinies (Ex: tout caractère non alphanumérique est un signe de ponctuation, un identificateur est une chaîne de caractères alphanumériques de longueur non nulle et commençant obligatoirement par un caractère alphabétique, etc ...). L'analyseur lexicographique est en particulier responsable de la reconnaissance des mots-clé, identificateurs, nombres, symboles spéciaux, ...

L'analyseur syntaxique, lui, vérifie la cohérence grammaticale du programme. Chaque fois qu'il reconnaît qu'une "phrase" est syntaxiquement significative, il déclenche une série d'actions correspondant au traitement de la phrase (Ex: génération de code objet, gestion de la table des symboles, ...).

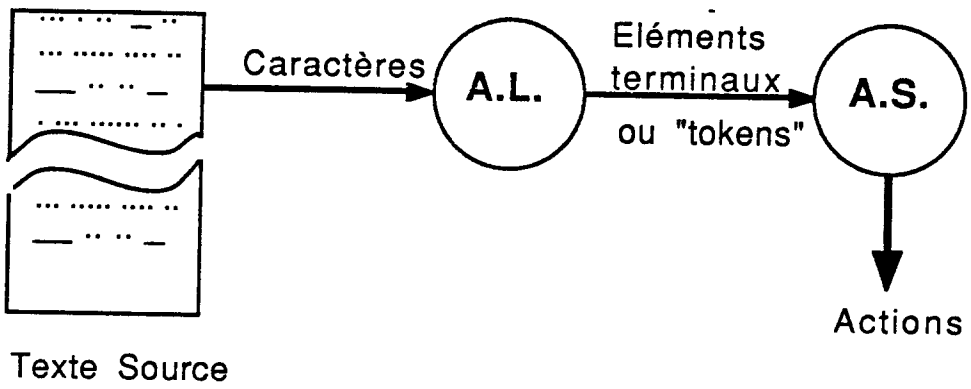


figure III.9

En appliquant ces principes à l'analyse d'un texte écrit en langage humain on pourrait définir la répartition des tâches suivante (cette décomposition est très schématique, car ne prenant pas en compte les ambiguïtés du langage naturel; elle est ici donnée à titre d'exemple pour mieux décrire le rôle des deux phases intervenant lors de la compilation d'un programme):

-1°) L'analyseur lexicographique va explorer le texte caractère par caractère de manière à former des mots séparés par des signes de ponctuation. Après consultation de son dictionnaire interne, l'analyseur lexicographique attribuera un "type terminal" à chaque mot (nom, verbe, adjectif, ... , nombre). A chaque "élément terminal" pourra être associée une valeur qui pour un mot peut être la chaîne des caractères composant le mot, ou pour un nombre, la valeur numérique de ce nombre.

Exemple:

	Le	petit	garçon	a	10	billès	bleues	.
Type:	Ar	Ad	N	V	C	N	Ad	SP
Valeur:	"Le"	"petit"	"garçon"	"a"	10	"billes"	"bleues"	'.'

Où: Ar = Article
Ad = Adjectif
C = Chiffre (ou nombre)
N = Nom
SP = Signe de ponctuation
V = Verbe

Pour le champ Valeur: les chaînes de caractères sont comprises entre deux " (Ex: "Le"), les caractères seuls entre deux ' (Ex: '.'); quand il s'agit d'un nombre, sa valeur numérique est donnée telle quelle.

2°) Ensuite, l'analyseur syntaxique, en fonction des "types terminaux" des mots constituant une phrase, va reconnaître le genre de phrase (affirmative, négative, interrogative, elliptique, ...) auquel il a affaire, et va déterminer sa signification en fonction de la valeur des "éléments terminaux" composant la phrase.

De manière générale, quand la syntaxe est "suffisamment régulière", les deux analyseurs peuvent être implantés sous forme de machines à états: chaque fois qu'un caractère (resp. un "élément terminal") est reçu, une transition est franchie, et l'analyseur lexicographique (resp. syntaxique) change d'état; les méthodes tabulaires sont donc particulièrement adaptées pour la réalisation des ces deux analyseurs: chaque analyseur est alors constitué d'un "moteur" gérant le fonctionnement de l'automate et de tables décrivant les transitions entre les différents états ainsi que les séquences de franchissement valides. Cette méthode est particulièrement avantageuse en phase de développement: à chaque modification de syntaxe, il suffit de modifier les tables ainsi que les actions associées aux franchissements des transitions alors que le "moteur" n'est pas touché.

La coopération entre les deux phases peut se faire de deux façons différentes:

1) Fonctionnement séquenciel:

L'analyseur lexicographique explore caractère par caractère le fichier source et produit un fichier intermédiaire composé des "éléments terminaux" (ou "tokens" en anglais) reconnus. Après analyse lexicographique, l'analyseur syntaxique va explorer le fichier intermédiaire et déclencher les actions appropriées.

2) Fonctionnement alterné:

Comme dans le cas précédent, l'analyseur syntaxique, pour effectuer son travail, se base sur des règles décrivant la syntaxe du langage (il existe plusieurs manières de décrire ces règles: Backus Naur Form, Diagrammes syntaxiques, ...). Chaque fois que l'analyseur syntaxique reçoit un "élément terminal" il applique ses règles de manière à analyser les instructions reçues. Mais après application d'un certain nombre de règles, l'analyseur syntaxique se retrouve "bloqué": plusieurs règles peuvent être appliquées. Pour lever l'ambiguïté, l'analyseur syntaxique appelle l'analyseur lexicographique qui lui renvoie alors l'élément terminal suivant reconnu dans le fichier source; l'arrivée d'un nouvel "élément terminal" permet en général à l'analyseur syntaxique de choisir entre les différentes solutions possibles; il applique alors les règles adéquates jusqu'à se trouver bloqué de nouveau, et le cycle se poursuit.

Il est nécessaire d'employer cette méthode chaque fois que le fonctionnement de l'analyseur lexicographique doit être modifié en fonction des éléments de grammaire déjà reconnus par l'analyseur syntaxique.

Exemple:

En règle générale, l'analyseur lexicographique "comprime" les caractères "blancs" et assimilés comme tels (Espace <SP>, Tabulation <TAB>, Nouvelle Page <FF>, Retour Chariot <CR>) qui servent de séparateurs: ainsi la séquence **<TAB><TAB><TAB>** est fonctionnellement équivalente à: **<SP>**. Mais à l'intérieur d'une chaîne de caractères (délimitée dans de nombreux langages par deux ") ces caractères ne doivent pas être sautés et doivent être traités de la même manière que tout autre caractère d'une chaîne.

C'est cette solution que nous avons utilisée pour la réalisation de l'interpréteur CLIPS, d'une part à cause de la remarque précédente, d'autre part parce que c'est la méthode utilisée par les outils de génération d'analyseurs présents sous le système UNIX: "**yacc**" et "**lex**" ([JOHN78] et [LESK]). YACC (pour Yet An other Compiler Compiler) et LEX sont respectivement des générateurs d'analyseurs syntaxiques et lexicographiques.

YACC convertit un fichier source décrivant les règles de grammaire du langage ainsi que les actions (instructions **C**) à effectuer chaque fois qu'une règle est applicable, en un programme **C**. La syntaxe du langage est introduite sous une forme pratiquement similaire à la notation **BNF** (Backus Naur Form). Le fichier **C** résultant peut pratiquement être décomposé en trois parties: le "moteur" de l'automate valable quel que soit le langage, les tables, calculées en fonction de la syntaxe du langage, qui donnent pour chaque état les transitions valides, et enfin les actions (spécifiées par le concepteur du

LEX fait de même avec les règles décrivant comment associer entre eux les caractères pour former des "éléments terminaux" de la syntaxe; chaque fois qu'un élément terminal est reconnu, il est possible de déclencher des actions (Ex: calcul de la "valeur" associée à l'élément terminal). Les fichiers C générés par LEX peuvent, de même, être décomposés en trois parties analogues à celles des fichiers produits par YACC.

La génération de l'interpréteur ou du compilateur se fait ensuite par compilation des deux fichiers ainsi obtenus (au moyen du compilateur "natif" pour des interpréteurs ou compilateurs destinés au monde Unix, ou d'un compilateur croisé pour un autre système cible: c'est le cas de l'interpréteur CLIPS sous CESAR).

La figure suivante résume les étapes de développement d'un interpréteur ou d'un compilateur sous Unix:

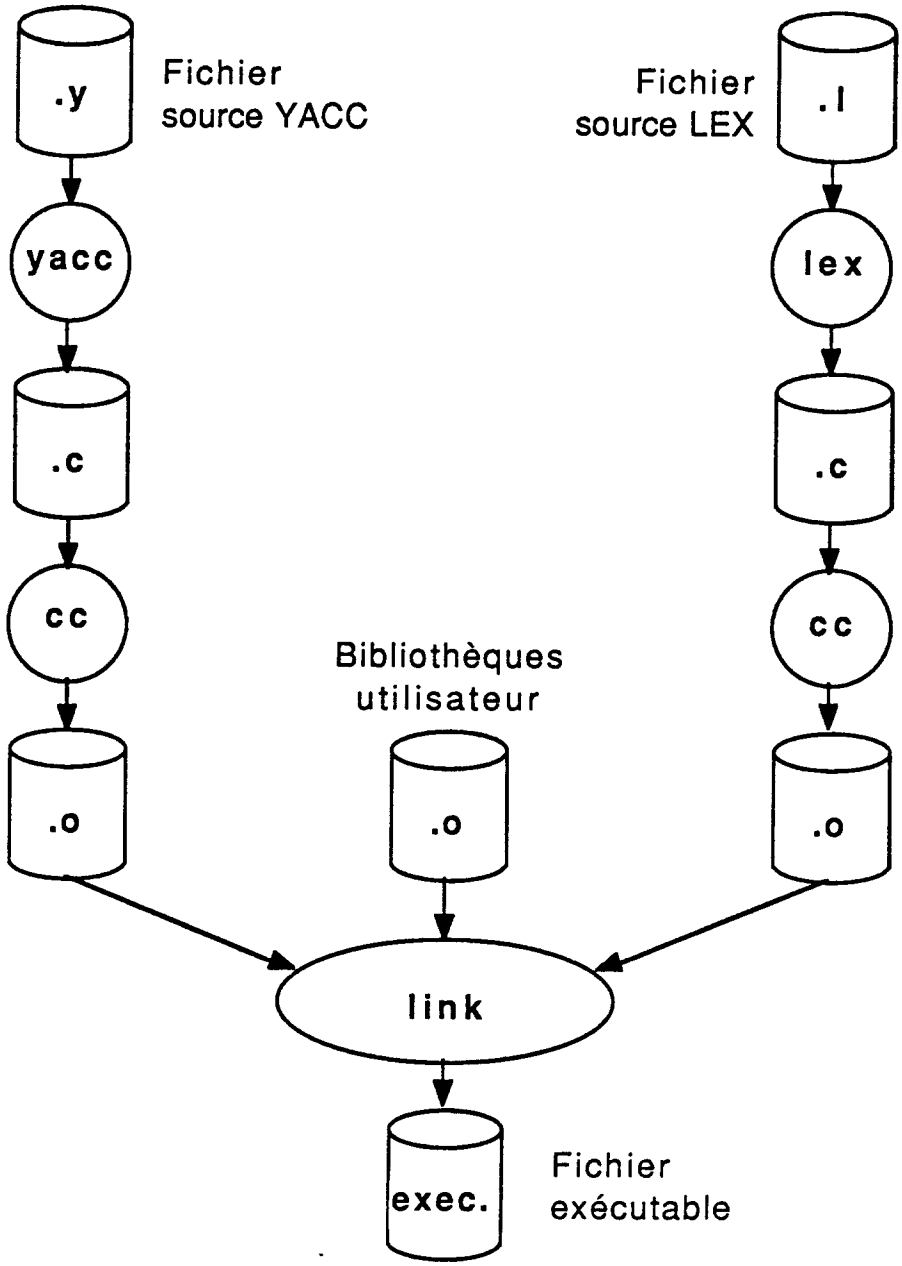


figure III.10

III.3.2 Structure des fichiers sources - Communication entre les analyseurs

La structure d'un fichier source YACC est généralement la suivante:

déclarations

%%

Règles et actions **C**

%%

Code **C** propre à l'application (en général: ouverture de fichiers et appel à **yyparse**())

La fonction yyparse[1] constitue le point d'entrée de l'analyseur syntaxique, cette fonction renvoie une fois terminée (lorsque la fin du fichier source à compiler ou à interpréter est atteinte) la valeur 0 si aucune erreur n'a été détectée et une valeur non nulle dans le cas contraire.

Après utilisation de YACC le fichier C résultant aura la structure décrite ci-dessous:

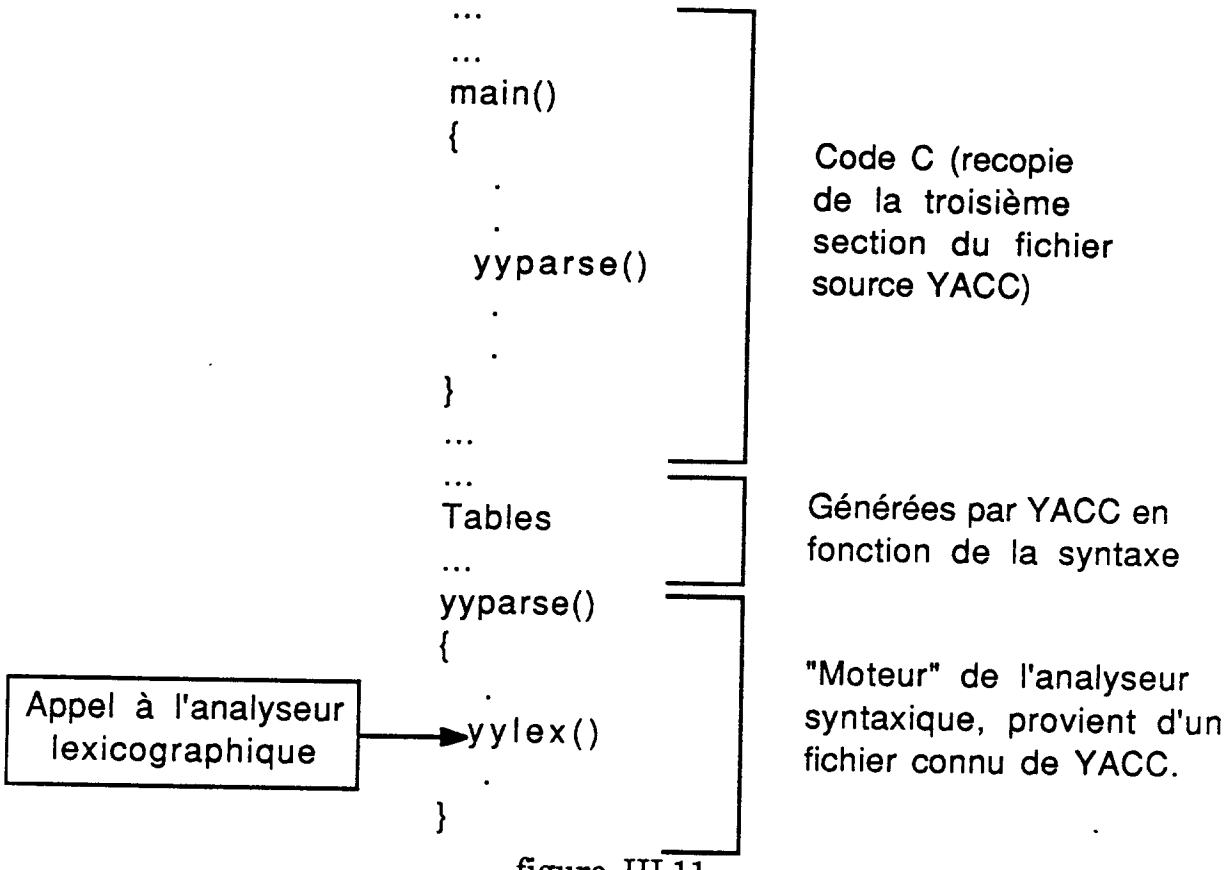


figure III.11

1 Tous les identificateurs débutant

La fonction **yylex**, qui est appelée dans le corps de la fonction **yyparse** est l'interface entre l'analyseur syntaxique et l'analyseur lexicographique. Chaque fois que l'analyseur syntaxique a besoin de connaître l'élément terminal suivant, il fait appel à yylex(); cette fonction après exploration du fichier source, lui renvoie l'élément terminal suivant codifié sous forme numérique avec la convention suivante:

n= 0	Fin de fichier
0 < n <128	Code ASCII d'un caractère, dans le cas où un caractère seul constitue un "élément terminal" (Ex: signes de ponctuation: , ; , opérateurs: + - = < >, ...).
n > 255	Numéro d'élément attribué par YACC chaque fois qu'un élément terminal est non atomique: c'est-à-dire plus complexe qu'un caractère (Ex: nombre, mot-clé, identificateur, opérateur composé: <= >=, ...).

De plus, à chaque "élément terminal" peut être associée une valeur dont le type est défini par l'utilisateur. Cette valeur est calculée par la fonction yylex().

Exemple:
Chaque fois qu' yylex() reconnaît un nombre (décimal, hexadécimal, ...), il peut être programmé pour associer à l'élément terminal NOMBRE sa valeur décimale. Ainsi: valeur de "123" = 123_{10}.

Chaque fois qu'un identificateur (de variable, de sous-programme, ...) est reconnu, une valeur possible associée à l'élément terminal IDENTIFICATEUR peut être un pointeur sur la chaîne de caractère constituant l'identificateur.

Cette valeur est passée dans la variable externe **yylval**; qui peut être, en fait, une union de types chaque fois que des valeurs de types différents peuvent être renvoyées (comme dans l'exemple précédent, où yylex peut retourner des variables de type entier ou de type pointeur sur une chaîne de caractères).

Lex convertit un fichier source (de suffixe .l) dont le format est le suivant:

Déclaration des variables externes et des classes de caractères
...
%%
Règles et actions

en un fichier **C** dont la structure générale est la suivante:

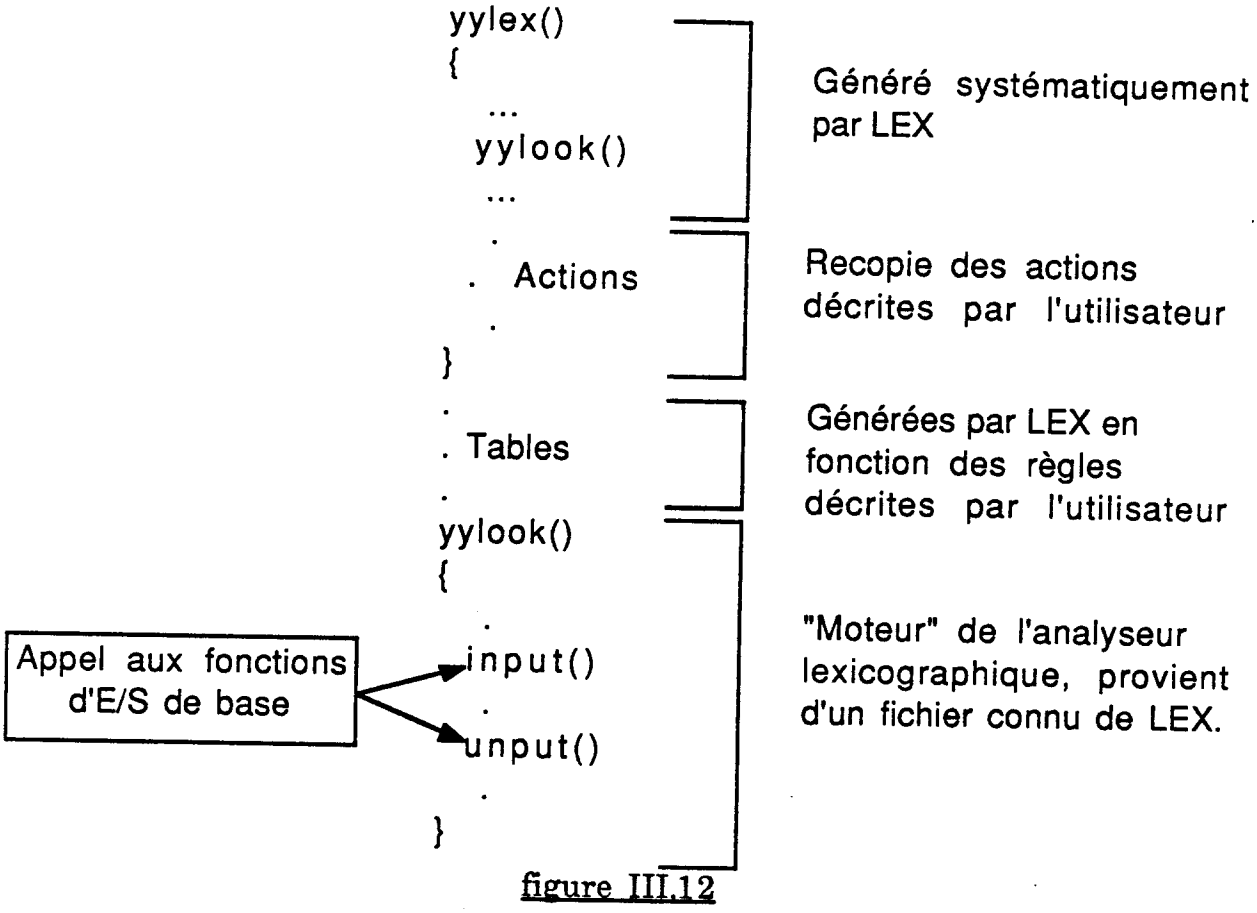


figure III.12

III.3.3 Avant-Propos

Lorsque nous avons décidé de doter l'ensemble LIPS d'un interpréteur, le superviseur et le synchroniseur étaient déjà achevés et ils disposaient alors, à quelques détails prés, des même fonctions qu'aujourd'hui. Pour effectuer le décodage d'instructions telles que "*ex primitive(arguments)* " ou "*ex operateur(primitive(arguments) ,primitive(arguments))* " LIPS posséde un jeu de fonctions permettant une analyse syntaxique sommaire; lors de la définition des fonctionnalités de l'interpréteur, il fut décidé de réutiliser ces fonctions chaque fois qu'apparaîtrait une requête au superviseur à l'intérieur d'un module CLIPS; pour cela l'interpréteur doit procéder à une "interprétation à la volée", c'est-à-dire qu'il travaille directement sur le code source du module et non sur un code intermédiaire. Ce choix présente aussi l'avantage de simplifier le dialogue entre le calculateur de développement et la baie de commande: lors des phases de chargement ou de déchargement d'un module, la tâche du calculateur hôte se résume à un transfert de fichier sans aucun transcodage; en contrepartie, l'interprétation de structures telles que les boucles ou les appels à des sous-programmes n'est possible qu'au prix de manipulations syntaxiques relativement peu orthodoxes.

III.3.4 Fonctionnement de l'analyseur lexicographique

Le programme source de l'analyseur lexicographique est donné en Annexe C.

Les descriptions des programmes sources de l'analyseur lexicographique et de l'analyseur syntaxique font souvent référence aux syntaxes d'entrée de

rons en cours de description certains aspects de la syntaxe d'entrée de ces deux "langages". Cependant, notre but n'est pas de remplacer les manuels d'utilisation de YACC et LEX; nous ne pouvons qu'en conseiller la lecture au lecteur (anglophone malheureusement!!) désireux d'en savoir plus.

La première section du fichier (du début du fichier jusqu'aux premiers %%) est composée de la déclaration des variables externes à l'analyseur lexicographique, de l'inclusion de certains fichiers "headers" (comme ytab.h, généré par yacc, qui attribue à chaque élément terminal non atomique son numéro associé) et de la définition des différentes classes de caractères.

Chaque classe de caractères est identifiée par une lettre et suivie de la liste de caractères la composant, ou de la liste de caractères n'en faisant pas partie:

Exemples:

```
    O   [01234567]
ou  O   [0-7]
```

Ces deux formes sont équivalentes et désignent la classe des caractères pouvant composer un nombre octal.

```
    Z   [^0123456789]
ou  Z   [^0-9]
```

désignent l'ensemble de tous les caractères qui ne sont pas des chiffres: le caractère ^ indique ici l'ensemble complémentaire.

L'analyseur lexicographique de l'interpréteur CLIPS définit les classes décrites ci-dessous:

L Classe des caractères alphabétiques (ou **L**ettres): de 'a' à 'z', de 'A' à 'Z' ainsi que le "blanc souligné" ('_').

C Classe des **C**hiffres: caractères de '0' à '9'.

H Classe des caractères pouvant composer un nombre **Hexadécimal**: '0' à '9', 'a' à 'f' et 'A' à 'F'.

A Classe des caractères **A**lphanumériques: union des classes L et C.

B Séparateurs ou caractères "**B**lancs" à savoir:

Symbole	Désignation	Représentation C
<SP>	Espace	' '
<TB>	Tabulation	'\t'
<FF>	Saut de page	'\f'
<CR>	Retour Chariot	'\r'

S **S**éparateurs d'instructions: ; , (et)

P Le reste des caractères, sauf \ (délimiteur de commentaires) et \n (saut de ligne). Classe des caractères de **P**onctuation.

W Ensemble [blwBLW], utilisé uniquement après une adresse pour spécifier la taille de l'élément (suivant syntaxe Motorola: B = Byte = Octet, W = Word = 16 bits et L = Long word = 32 bits) pointé par cette adresse.

La deuxième section du fichier (des %% à la fin du fichier) est composée des règles permettant de détecter les différents éléments terminaux, ainsi que des actions déclenchées chaque fois qu'une suite de caractères correspond à une règle.

Les différentes règles de l'analyseur lexicographique de CLIPS sont décrites ci-dessous dans leur ordre d'apparition dans le fichier source (les règles sont en caractères gras, leur signification ainsi que les actions effectuées en caractères normaux):

- **{B}** Tous les caractères "blancs" sont sautés.
- **"\n"** Chaque fois qu'un saut de ligne est rencontré, le compteur de lignes (nligne) est incrémenté. Si l'interpréteur analyse le programme principal d'une application, la valeur du compteur de lignes est comparée aux deux bornes (n1 et n2) du mode "pas à pas" pour savoir s'il y a lieu de suspendre l'interprétation jusqu'à l'arrivée d'un ordre **resume** (cas où n1 <= nligne <= n2). Si tel est le cas, l'analyseur lexicographique appelle la fonction C **wtresum()**; cette fonction génère le message *Program xx paused* .... puis appelle à son tour la fonction assembleur **ykpause()**; cette dernière est chargée de l'interface avec le superviseur: elle écrit sur la f.o. du superviseur pour lui signaler que le module en cours est pausé et effectue une lecture **avec attente** sur la f.o. d'entrée de l'interpréteur. Si la première requête reçue est un ordre **resume** la fonction ykpause() renvoie une valeur booléenne FAUX et l'interprétation reprend alors son cours; au contraire, si la première requête reçue est un ordre d'arrêt, la fonction ykpause() renvoie une valeur boolénne VRAI: l'interprétation sera alors arrêtée.
- Les cinq règles suivantes ont pour but de reconnaître les opérateurs composés suivants:

  <= Inférieur ou égal
  >= Supérieur ou égal
  != Différent de
  | | "connecteur logique" OU
  && "connecteur logique" ET
- Le groupe suivant de règles (de **break** à **while**) est chargé de la reconnaissance des mots-clé du langage. Sa description se passe de tout commentaire. Aucune valeur n'est associée à un mot-clé, sauf pour le mot-clé **while** pour lequel yylex retourne la valeur du pointeur de décodage courant (la valeur retournée pointe sur le w du while).
- La règle suivante: **{L}{A}*** , ce qui en terme de syntaxe LEX signifie, au moins une lettre suivie d'un nombre, éventuellement nul, de symboles alphanumériques consécutifs, est la représentation classique d'un identificateur.

  Chaque fois qu'un identificateur est reconnu, la chaîne de caractères le composant est copiée dans une zone mémoire tampon avec troncature éventuelle à la taille maximale d'un identificateur (15 caractères).

  N.B.: Il est très important que cette règle soit située APRES les règles décrivant les mots-clé. En effet, les mots-clé sont, eux aussi, composés d'une suite de caractères alphanumériques débutant par une lettre;

décrivant un identificateur; dans un tel cas de conflit, LEX accorde la priorité à la règle en conflit définie la plus tôt dans le fichier source. L'inversion de l'ordre des règles se traduirait par la conséquence fâcheuse suivante: tout mot-clé serait reconnu comme un identificateur, ce qui n'est quand même pas le but recherché. Le corollaire de ce choix est que les mots-clé sont réservés: un identificateur ne peut avoir le même nom qu'un mot-clé. Par contre, si LEX rencontre la suite de caractères suivants:

<mot-clé><identificateur>

deux interprétations sont alors possibles: soit l'on reconnaît un identificateur composé de la concaténation des deux chaînes, soit un mot-clé suivi d'un identificateur. Dans un tel cas de conflit, LEX donne la priorité à la règle qui permet de concaténer le maximum de caractères; dans ce cas, LEX reconnaîtra donc un identificateur.

Exemple: "pauser" est un nom d'identificateur valable, bien qu'il commence par "pause" (mot-clé).

- Les deux règles suivantes: **\${H}+** et **{C}+** décrivent respectivement la représentation des nombres hexadécimaux et décimaux: un nombre hexadécimal commence par le caractère '$' et est composé d'une suite de caractères hexadécimaux (longueur minimale de la suite = 1); un nombre décimal étant composé d'une suite de longueur minimale 1 de caractères décimaux. Les actions associées à ces deux règles consistent en la conversion de la chaîne de caractères (exprimée en base 16 dans le premier cas, et en base 10 dans le second) en le nombre qu'elle représente. La valeur ainsi calculée est la "valeur associée" à l'élément terminal NOMBRE.

- La règle qui suit: **@{W}** reconnaît le symbole @ suivi par un des caractères b, w, ou l (ou leur équivalent en majuscule) comme étant l'opérateur d'adressage. La valeur associée à l'opérateur d'adressage est le caractère qui suit le @: l'interpréteur saura ainsi quelle est la taille de l'objet auquel il faut accéder (8 bits pour b, 16 pour w et 32 pour l).

- L'antépénultième règle: **{S}** est chargée de tester par appel à la fonction externe **ykscrute()** l'arrivée éventuelle d'un ordre d'arrêt de l'interprétation en provenance du superviseur, et ce, uniquement après un séparateur d'instructions: ; , { ou } . Ainsi un module ne peut être arrêté qu'entre deux instructions. En cas de demande d'arrêt (si la valeur rendue par ykscrute() est non nulle), certains drapeaux (dont le rôle sera expliqué dans le paragraphe concernant l'analyseur syntaxique) sont levés, et yylex renvoie alors l'élément terminal 0 (qui indique une fin de fichier) de manière à arrêter l'interprétation.

- L'avant dernière règle permet, dès qu'elle détecte un '\', de sauter tous les caractères jusqu'au '\' suivant, en prenant la précaution pendant cette recherche de tester l'absence de tout caractère '\0' (ou <NUL>); dans le cas contraire, cela signifierait qu'un commentaire ouvert avant la fin de fichier n'a pas été fermé, l'analyseur syntaxique rapportera ainsi une erreur de syntaxe du type "fin de fichier inattendue" ou "unexpected end of file". Le saut des caractères compris entre deux '\' se fait par appel explicite à la fonction input() plutôt que par l'utilisation d'une règle du type \.*\ qui pourrait d'une part saturer les tampons in-

rait de concaténer le plus de caractères possibles jusqu'à rencontrer le dernier \; dans notre exemple, ce qui serait reconnu comme un commentaire est tout le texte encadré par les \ en gras, ce qui n'est pas le fonctionnement désiré.

\ Ceci est un commentaire\
... Instructions
\ Nouveau commentaire \

d'autre part, ce type de règle interdirait la possibilité de définir des commentaires "enjambant" plus d'une ligne du fait de la restriction imposée par LEX de façon à éviter une possible saturation..

- Enfin, la dernière règle renvoie à l'analyseur syntaxique tous les caractères qui n'ont pas été utilisés dans les règles précédentes: c'est-à-dire tous les caractères isolés, dits "de ponctuation": **+** , **-** , , , ***** , . , / , etc ... ou des caractères du type **<** ou **>** quand ils n'ont pu être associés avec un autre caractère pour former un opérateur composé.

Pour explorer caractère par caractère le fichier source ou la zone mémoire contenant le programme à interpréter, l'analyseur lexicographique fait appel à la fonction **input()** chaque fois qu'il a besoin de "lire" le caractère suivant. Cependant, certaines règles amènent l'analyseur lexicographique à lire "un caractère de trop": c'est le cas des règles définissant les identificateurs, les mots-clé ou les nombres par exemple; en effet, supposons que l'analyseur lexicographique a déjà lu les quatre lettres du mot-clé **case**, deux solutions sont possibles: soit l'analyseur lexicographique a affaire au mot-clé case, soit il a affaire à un identificateur commençant par les quatre lettres "case". L'analyseur lexicographique doit alors lire le caractère suivant; si ce caractère est alphanumérique l'analyseur lexicographique considère qu'il a alors reconnu une partie d'un identificateur et il continue en séquence; par contre, si le dernier caractère lu n'est pas alphanumérique, l'analyseur lexicographique déduit qu'il a bien rencontré le mot-clé **case**, dans ce cas, il aura eu besoin de lire un caractère supplémentaire pour lever l'ambiguïté, il y a alors lieu de "remettre" ce caractère dans le fichier d'entrée: c'est le rôle de la fonction **unput()**. Dans les bibliothèques standards livrées avec le système Unix, les fonctions input() et unput() sont réalisées grâce aux fonctions **getc()** et **ungetc()** chaque fois que l'utilisateur n'écrit pas ses propres versions d'input() et unput(). Dans le cas de notre application, ces versions par défaut ne conviennent pas: d'une part la notion de fichier standard d'entrée n'existe pas sous CESAR; d'autre part, l'analyse se fait en mémoire, de plus, afin de permettre le chargement et la destruction de programmes nous avons du utiliser une structure particulière de "blocs mémoire" facilitant la gestion dynamique des programmes. La structure d'un programme est présentée ci-dessous:

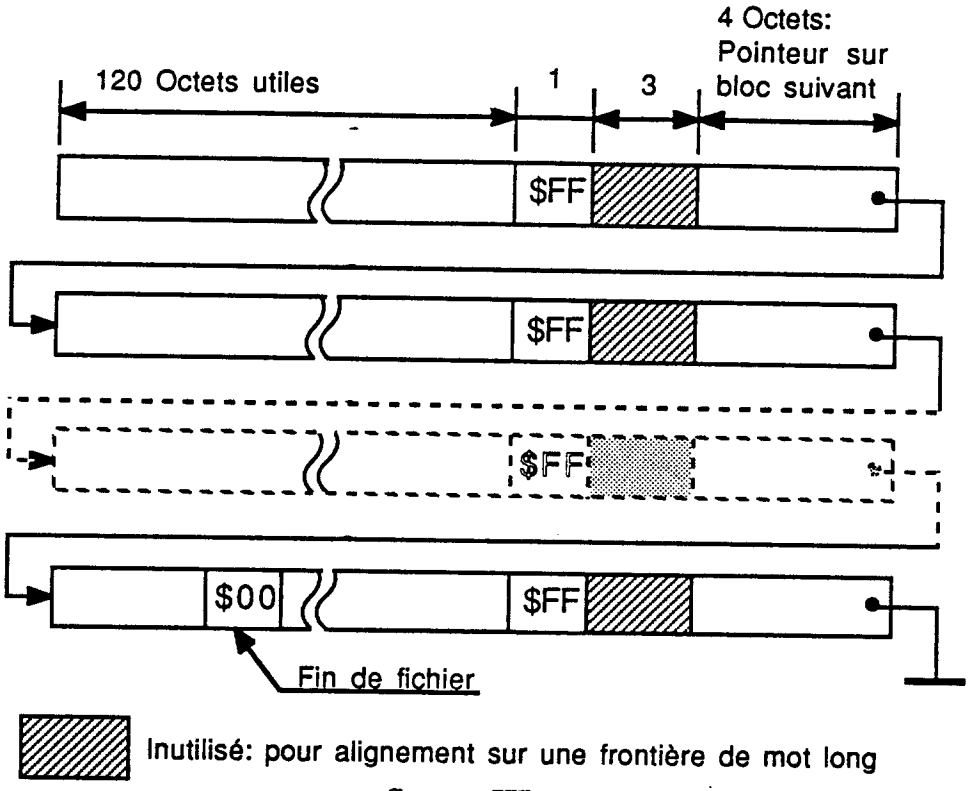


figure III.13

Chaque programme est composé d'un ensemble de "blocs" chaînés les uns aux autres; à l'intérieur d'un bloc, un caractère spécial, non utilisé par l'interpréteur ($FF), signale la fin d'un bloc; il est suivi de trois octets inutilisés (dont la valeur est $FF) servant à assurer l'alignement du dernier champ sur une frontière de mot long, lequel champ est un pointeur sur le bloc suivant. La fin du module est signalé par un caractère nul.

Les deux fonctions input() et unput() que nous avons écrites se partagent deux variables de type pointeurs: **nextchar** et **lastchar**. Nextchar pointe sur le prochain caractère à lire, et lastchar sur le caractère qui vient juste d'être lu. D'où les algorithmes des fonctions input() et unput():

```
input:
Début
    Lire le caractère c pointé par nextchar
    Si (c = $FF) Alors
        Sauter la zone inutilisée
        Affecter à nextchar l'adresse du bloc suivant
        Lire le caractère c pointé par nextchar
    FinSi
    lastchar = nextchar
    Incrémenter nextchar
    Retourner(c)
Fin

unput:
Debut
    nextchar = lastchar
Fin
```

Il apparaît clairement que telle qu'elle est décrite, la fonction unput() ne peut être utilisée **deux** fois de suite: un appel au moins à input() est nécessaire entre deux appels à unput(). Nous avons utilisé une version simplifiée de la fonction unput() en nous basant sur le fait qu'après un appel à unput(), LEX fait **toujours** appel ensuite à input() (c'est-à-dire qu'il ne lui arrive de lire qu'un seul "caractère en trop", mais pas plus); dans le cas contraire nous aurions été obligés de modifier la structure des blocs programmes en utilisant un double chaînage (chaînage direct tel qu'il est représenté dans la figure précédente, plus chaînage inverse) ce qui aurait entraîné une gestion beaucoup plus lourde des blocs programmes. Après écriture des ces deux fonctions en **C** et examen du code généré (vraiment peu efficace !!) nous avons du re-écrire ces deux fonctions en assembleur de manière à améliorer les performances de l'interpréteur: ces deux fonctions sont en effet les routines de plus bas niveau de l'interpréteur et sont appelées très fréquemment (surtout input()).

III.3.5 Fonctionnement de l'analyseur syntaxique

Il est indéniable que YACC est un logiciel très pratique et puissant chaque fois que l'on a besoin de générer un programme capable de traiter un flot de "données structurées", donc en particulier pour concevoir des analyseurs syntaxiques de compilateurs.

Une chose est sûre, sans YACC l'interpréteur CLIPS serait resté à l'état de projet.

Cependant, si la génération de compilateurs mono-passe grâce à YACC ne pose pas de gros problèmes, celle d'interpréteurs relativement "évolués" soulève de nombreuses difficultés (en particulier pour l'analyse répétitive de structures telles que les boucles "while"). De plus, YACC et LEX s'échangent des informations par le biais de variables globales (cette solution est incontournable, car imposée par le code correspondant aux moteurs des analyseurs, ce code n'étant pas modifiable), ce qui empêche de rendre l'ensemble analyseur lexicographique + analyseur syntaxique réentrant (rappelons qu'en C, ou une variable est "dynamique" et donc uniquement locale à la fonction dans laquelle elle est déclarée, ou elle est de portée globale; il n'y a pas de "juste milieu" comme en Pascal où une variable peut être locale à un ensemble de procédures [KRUT 79]). Rendre l'interpréteur réentrant nous aurait permis de simplifier l'analyse des sous-programmes. Tout ceci fait que certaines règles de syntaxe ou actions apparrament bizarres[1] pourront surprendre.

III.3.5.1 Interface avec le superviseur

Toute requête d'exécution d'un module adressée au superviseur peut s'écrire sous la forme:

call nom_module '(' paramètres éventuels ')'
ou **call** nom_module

s'il n'y a pas de paramètres. Ce type de requête provoque l'exécution inconditionnelle du module de nom *nom_module*. Il est possible d'exécuter un module

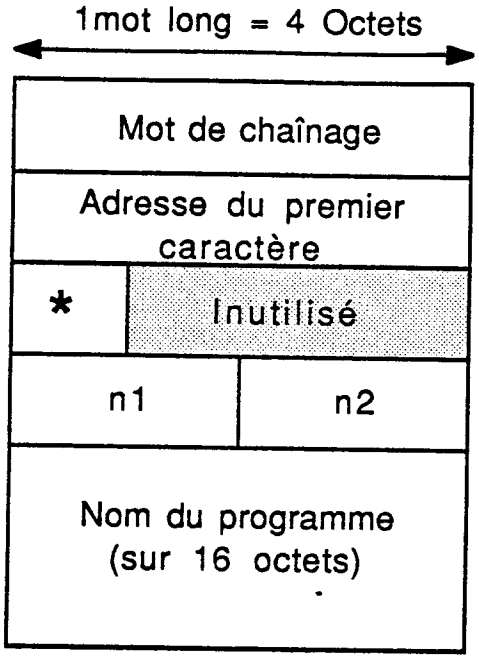
---

[1] Caractères cachés, fausses erreurs de syntaxe pour déclencher le mécanisme de récupération d'erreurs dans certains cas, ... Malgré tout, nous avons essayé de minimiser le nombre

en mode pas à pas: l'utilisateur fournit à l'interpréteur un intervalle de lignes, chaque fois que l'interpréteur a fini d'analyser et d'exécuter une ligne comprise dans cet intervalle, il suspend son action et attend un ordre **resume** sur la ligne de dialogue; ce mode permet la mise au point de modules (trace de variables utilisées entre deux pas, décomposition d'un mouvement complexe d'un robot en plusieurs mouvements plus simples et contrôle des points de passage intermédiaires, ...); la syntaxe de la ligne de commande au superviseur est alors:

step <l1> <l2> <nom_module> [(<paramètres>)]
où paramètres est défini comme
<paramètre> [, <paramètre> [, ...]]
Les parenthèses ne sont pas indispensables s'il n'y a pas de paramètres à passer au module.

Lorsque le superviseur reçoit une de ces deux commandes, il explore la chaîne des modules présents en mémoire et recherche un module dont le nom correspond à *nom_module*. En cas d'échec, un message d'erreur avertit l'utilisateur que le module en question n'a pas été trouvé. Dans le cas contraire, le superviseur appelle la routine principale de l'interpréteur (fonction **interp** dans le listing donné en Annexe D) en lui passant un pointeur sur le descripteur de programme correspondant au module à interpréter, ainsi qu'un pointeur sur la ligne de commande au superviseur (par ce pointeur, l'interpréteur pourra, par la suite, avoir accès aux différents paramètres qui ont pu être passés au module). Comme à l'accoutumée dans LIPS, cette ligne de commande est contenue dans un tampon d'E/S que l'interpréteur devra utiliser pour y placer son message de retour (*0000 Ok module xx* ou *XXXX erreur yy module xx, line zz*). Cependant, avant d'appeler la fonction interp, le superviseur, suivant la commande (**call** ou **step**), initialise deux champs (non décrits dans le §III.1.3.1.3) du descripteur de programme. Ces deux champs sont les numéros de lignes entre lesquelles il y aura lieu de passer en mode pas à pas si cela a été demandé par la commande **step**; dans le cas d'une commande **call**, ces deux champs sont initialisés à zéro pour indiquer que le mode pas à pas n'est pas actif. La structure complète d'un descripteur de module est donc la suivante:



* Indicateur d'exécution

figure III.15

III.3.5.2 Initialisation de l'interpréteur

Avant de lancer l'interprétation proprement, la fonction **interp** exécute une série d'actions destinées à préparer l'environnement de travail de l'interpréteur:

- Dans un premier temps, elle initialise le compteur de ligne (**nligne**) à un et récupère dans le descripteur de module l'adresse du premier bloc programme et la recopie dans la variable nextchar qui pointe ainsi sur le tout premier caractère composant le module. Puis en fonction des informations trouvées dans le descripteur de module, les variables ligne1 et ligne2, respectivement numéro de ligne de début et de fin du mode pas à pas sont calculées; si le mode pas à pas est désélecté, ces deux variables se voient affecter une valeur nulle, ainsi le test de déclenchement du mode pas à pas (ligne1 <= nligne <= ligne2) sera toujours faux.
- A partir du pointeur sur la ligne de commande, la fonction interp va ensuite sauter les caractères jusqu'à rencontrer une parenthèse ouvrante ou un octet nul (indiquant ici une ligne de commande sans paramètres) de manière à pointer sur le début d'une éventuelle zone de paramètres; cette zone sera décodée ultérieurement par l'analyseur syntaxique.
- Après initialisation de différents pointeurs de piles utilisées par l'analyseur syntaxique, la fonction interp active le mécanisme de récupération d'erreur bus (appel à la fonction berron), puis lance l'interprétation du module.

  La fonction assembleur **berron()** a pour rôle de modifier certains vecteurs d'exception du 68000 (erreur bus: vecteur numéro 2, adresse $8 et erreur d'adressage: vecteur numéro 3, adresse $C) de manière à les faire pointer sur une routine spécifique de récupération de ces erreurs et non sur la routine standard CESAR qui provoquerait un CRASH sur

retour" de la fonction yyparse de façon à ce qu'en cas d'erreur bus ou d'adressage, la routine de récupération provoque l'arrêt de l'interprétation: yyparse renverra alors un code spécifique d'erreur. Les vecteurs normaux de CESAR sont cependant sauvegardés et seront restaurés après interprétation du module par la fonction inverse de berron: **berroff()**. Il est possible de décrire le mécanisme de récupération d'erreur en s'inspirant de fonctions bien connues de tout programmeur système sous UNIX, mais inexistantes sous CESAR:

```
main()
{
      /* Activation du mécanisme de récupération d'erreur */

      signal(BUS_ERROR, berr_handler);
      if (yyparse() == 0)
            ERRFLAG = OK;

      /* retour au traitement standard des erreur bus */

      signal(BUS_ERROR, SIG_OFF);
      ...
}
...
jumpbuff  parser_abort;

yyparse()
{
      ...
      ...
      return(xxx);

      /* En cas d'erreur bus, on aboutira ici */

      setlongjmp(parser_abort);
      return(BUSERR);          /* Echec interpréteur: erreur bus */
}
...
berr_handler()
{
      longjmp(parser_abort);
}
```

figure III.16

III.3.6 Structures et fonctionnement de l'analyseur syntaxique

Plutôt que de nous lancer dans une explication linéaire du fonctionnement de l'analyseur syntaxique, qui serait fastidieuse et nous obligerait à faire de nombreuses digressions, nous avons préféré exposer les principaux problèmes auxquels nous avons été confrontés; des solutions retenues découlent aisément les structures de données employées. Ainsi, la compréhension de certaines règles (à priori en contradiction avec les diagrammes syntaxiques du langage présentés en Annexe E) ou de certaines actions "bizarres" sera grandement améliorée.

III.3.6.1 Pile des drapeaux

Le langage CLIPS comprend plusieurs instructions permettant des "ruptures de séquence": bloc **if-else**, instruction **case**, boucle **while**, appel de sous-programme, ... Aussi, en fonction de la véracité des expressions logiques de contrôle des blocs if-else ou des boucles while, ou en fonction de la valeur numérique des sélecteurs des instructions case, certaines instructions devront être exécutées, tandis que d'autres devront au contraire être sautées. Considérons le segment de programme suivant:

```
int i = 1;
if (i)
{
      ... Instructions 1
}
else
{
      ... Instructions 2
}
```

figure III.17

Dans ce cas, les instructions *instructions1* seront analysées **ET** exécutées, tandis que les instructions *instructions2* **NE** seront **QU**'analysées (dans le but de détecter des erreurs de syntaxe éventuelles). Chaque fois qu'une instruction (rappelons que syntaxiquement parlant, un bloc **if-else** ou une boucle **while**, par exemple, sont considérés comme une instruction unique) permettant une rupture de séquence du programme est exécutée, l'interpréteur positionne une variable logique (**evflag**: 1 octet) en fonction de la valeur logique de l'expression de contrôle de l'instruction analysée. Si evflag à la valeur VRAI, les instructions suivantes seront exécutées; sinon, elle ne seront que seulement analysées. Cependant, avant toute affectation d'une nouvelle valeur à evflag au début d'un bloc, il est nécessaire d'enregistrer la valeur actuelle d'evflag; cette valeur sera restaurée à la sortie du bloc. Nous avons affaire ici à un mécanisme de mémorisation de type FIFO dont l'implantation se fait classiquement sous forme de pile. La pile des flags est composée d'un tableau suffisamment grand (**pilef**) et d'un pointeur de pile **pf_ptr**, qui pointe toujours sur la dernière valeur placée dans la pile; au début de l'interprétation pf_ptr pointe derrière le dernier octet du tableau pilef; afin de standardiser la gestion de cette pile nous avons défini deux fonctions: **fempile** et **fdepile** respectivement chargées de la sauvegarde et de la restauration des différentes valeurs d'evflag dans la pile, suivant le principe suivant:

```
fempile (flag)
Debut
      Décrémenter le pointeur de pile pf_ptr
      Si pf_ptr pointe au dela du tableau pilef Alors
            Erreur: pile pleine
      FinSi
      Enregistrer flag dans la case pointée par pf_ptr
Fin

fdepile()
Debut
      flag = contenu de la case pointée par pf_ptr
      Incrémenter pf_ptr
Fin
```

figure III.18

L'exemple suivant illustre les différentes actions effectuées lors de l'analyse de blocs **if-else** et les différents états de la pile qui en résultent:

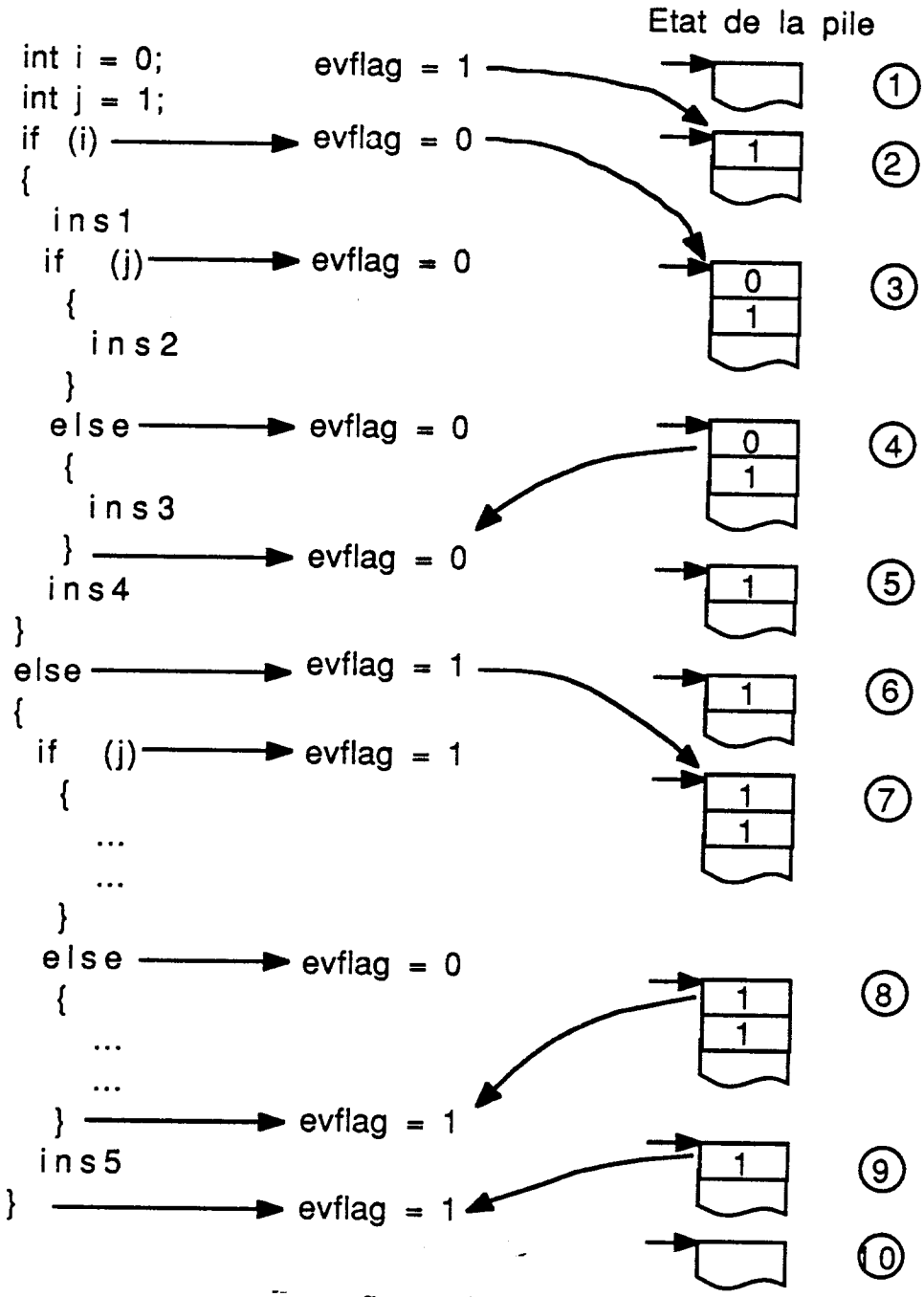


figure III.19

En 1: Etat initial, evflag est supposé être VRAI pour les besoins de la démonstration.

En 2: L'évaluation de i (= 0) donne un résultat FAUX, evflag prend donc la valeur FAUX, la valeur précédente (VRAI) est empilée.

En 3: evflag étant FAUX, le deuxième bloc if-else n'est donc pas exécutable: il est interdit de modifier la valeur d'evflag qui reste FAUX (même si l'expression de contrôle du bloc est VRAIe, ce qui est le cas ici), la valeur précédente d'evflag est néanmoins empilée.

En 4: Rencontre d'une clause else; cependant la dernière valeur d'evflag (présente au sommet de la pile) est fausse, indiquant que le bloc if-else

En 5: Le '}' fermant un bloc if-else est rencontré. La dernière valeur d'evflag (FAUX) est extraite au sommet de la pile et affectée à evflag. Ce dernier étant faux, les instructions qui suivent le bloc if-else interne (instructions *ins4*) ne seront pas exécutées.

En 6: Rencontre d'un ordre else. Si la valeur au sommet de la pile est vraie (c'est-à-dire: le bloc if-else est contenu dans un bloc dont l'exécution est autorisée; ce qui est le cas ici), evflag est complémenté: il passe de FAUX à VRAI ici.

En 7: L'évaluation de l'expression conditionnant le bloc est possible (evflag = VRAI); elle donne le résultat VRAI, la derniére valeur d'evflag (VRAI) est empilée et evflag reçoit la valeur de l'expression évaluée (VRAI).

En 8: Cas inverse du cas 6: evflag passe de VRAI à FAUX.

En 9: Rencontre d'un '}', d'où dépilage de la dernière valeur d'evflag qui devient donc VRAI; les instruction *ins5* peuvent donc être exécutées.

En 10: Cas identique au cas 9.

D'où les trois règles utilisées pour le traitement des blocs if-else:

- Lors de la rencontre d'un ordre **if**, il y a empilage de la valeur actuelle d'evflag, et si cette dernière est vraie, l'expression de contrôle du bloc est évaluée et affectée à evflag. Ce traitement est effectué par les instructions nommées **IF1** dans le listing de l'interpréteur donné en Annexe.D
- S'il existe un ordre **else**, la valeur au sommet de la pile est testée, et si elle est vraie, la valeur d'evflag est complémentée: instructions **IF2**.
- En fin d'un bloc if-else, il y a dépilage de la dernière valeur d'evflag: instructions **IF3**.

III.3.6.2 Traitement d'un bloc case

L'ordre **case** sert à implanter des sélections multiples, sa syntaxe est la suivante:

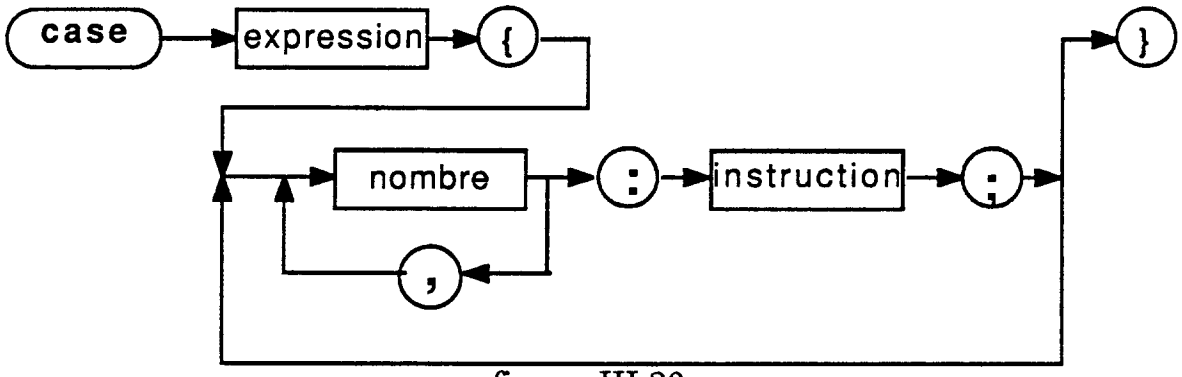


figure III.20

L'évaluation d'un tel ordre commence tout d'abord par l'évaluation de l'expression de sélection du bloc case; ensuite la valeur de cette expression est comparée aux valeurs présentes dans les différentes listes de constantes; dès qu'il y a correspondance entre la valeur de l'expression et une constante,

fault est présente, ce sera l'instruction associée à la clause default qui sera exécutée. L'ordre case est en quelque sorte la généralisation d'un bloc if-else if-... -else. De ce fait, le traitement de l'ordre case est analogue à celui d'un bloc if-else:

En tête d'un bloc case, comme pour un bloc if-else, la valeur actuelle d'evflag est empilée, et si evflag était VRAI l'expression de sélection de l'ordre CASE est évaluée et affectée à, la variable **casval** (instructions **case1** dans le listing). De même, cette valeur sera dépilée en sortie du bloc (instructions **case2**).

Chaque liste de constantes se voit attribuer lors de son analyse une valeur logique vraie ou fausse suivant qu'une des constantes de la liste est égale ou non à casval. Une telle liste est définie de manière classique sous YACC en utilisant une notation récursive que nous retrouverons plusieurs fois par la suite: une liste (sous-entendue de longueur *n*) est soit un élément isolé (une constante ici), soit une liste (sous-entendue de longueur *n-1*) suivie d'un élément isolé. En se basant sur cette propriété, nous pouvons alors définir la procédure qui permet d'attribuer une valeur logique à la liste:

- Si la liste est composée d'un seul élément, la valeur de la liste est VRAIe si l'élément est égal à casval.
- Si la liste est de longueur n, la valeur de la liste est VRAIe si sa "sous-liste de tête" (longueur n-1) a la valeur VRAI, ou si le dernier élément de la liste est égal à casval.

Ce traitement est illustré par les deux règles (et leurs actions associées) qui sont annotées **case5** sur le listing.

L'interprétation de chaque groupe *liste de constantes ':' instruction* est effectuée en utilisant les règles de modifications d'evflag suivantes:

- En début d'un bloc case (à condition qu'il soit exécutable) evflag est VRAI.
- Pour chaque groupe, si evflag est VRAI (indiquant ainsi que le bloc case actuel est évaluable ET qu'aucune liste de constante ne contenait jusqu'à présent la valeur de la variable de sélection du bloc case), evflag prend la valeur logique assoicée à la liste de constantes (VRAI ou FAUX selon que la variable de sélection prend ou non sa valeur parmi la liste). Cependant, cette action risque de modifier la valeur actuelle d'evflag, qui doit donc être empilée au préalable. Instructions **case3**.
- En fin d'un groupe, evflag est dépilé; et s'il était VRAI avant dépilage (indiquant que le groupe qui vient d'être interprété était exécutable), evflag devient FAUX de manière à interdire l'exécution de tous les blocs suivants. Instructions **case4.**
- Si une clause **default** est présente, et qu'evflag est VRAI (aucun groupe précédent n'a été exécuté), l'instruction associée à cette clause sera exécutée.

III.3.6.3 Traitement des boucles while

A l'opposé des deux structures vues précédemment, une boucle **while** est une structure susceptible d'être exécutée plusieurs fois. Il faut donc prévoir au niveau de l'interpréteur le moyen d'exécuter plusieurs fois, si nécessaire, un

- Soit, lors de la première analyse de la boucle, mémoriser les instructions rencontrées sous forme d'un code intermédiaire (du type P-Code, code machine, etc...), qui sera exécuté lors des itérations suivantes dans la boucle.
- Soit, en fin de chaque itération à travers la boucle, forcer l'interpréteur à "revenir en arrière" et à ré-analyser la boucle tant que l'expression de contrôle de la boucle reste vraie.

La première solution comporte plusieurs inconvénients: il est tout d'abord nécessaire de doter l'interpréteur d'une fonction de type génération du code intermédiaire; d'autre part, plus le nombre d'instructions englobées dans une même boucle est grand, plus la taille du code intermédiaire sera importante, ce qui, à moins de disposer d'une mémoire suffisament importante, interdit les boucles trop longues. Aussi, nous nous sommes orientés vers la deuxième solution.

Si faire revenir l'interpréteur en arrière ne pose pas de problème (il suffit de mémoriser la valeur du pointeur de décodage courant *nextchar* en début de boucle, et de restaurer cette valeur en fin de boucle), il faut trouver un moyen de forcer une nouvelle analyse de la boucle. Pour cela il a été nécessaire de modifier légèrement la syntaxe du langage vue de l'interpréteur:

Alors qu' "officiellement" (c'est-à-dire, telle qu'elle est décrite dans les diagrammes syntaxiques donnés en Annexe E) la syntaxe d'une boucle while est:

boucle while:

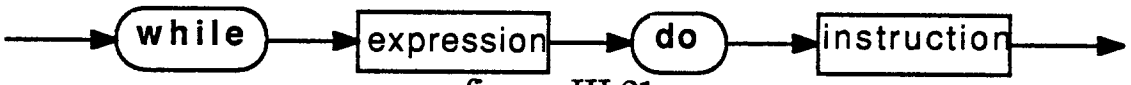


figure III.21

une telle boucle est définie dans les tables de l'interpréteur comme:

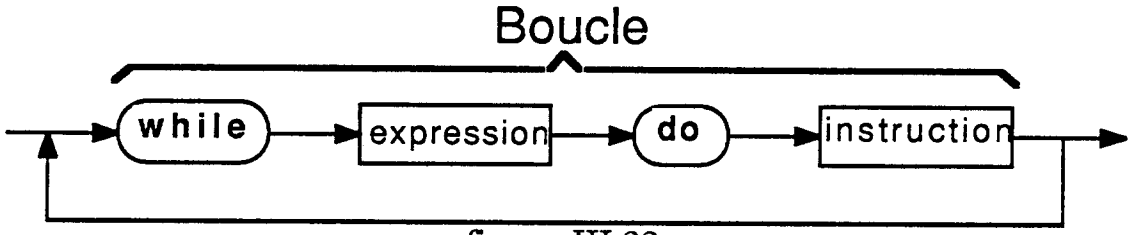


figure III.22

de telle sorte que l'interpréteur analysera la même boucle tant que l'expression de contrôle de la boucle restera vraie (à condition que le pointeur de décodage courant nextchar soit ramené en début de boucle chaque fois qu'il en atteint la fin). Cependant, cette solution peut laisser à l'utilisateur la possibilité d'écrire une suite d'instructions comme ci-dessous:

while *expression1* **do**
    instruction1
while *expression2* **do**
    instruction2

Cette suite d'instructions comporte d'une part une erreur de syntaxe (pas

comme une nouvelle itération de la première. Aussi, pour ce genre de "corruption" de la syntaxe, l'interpréteur doit trouver ,entre deux itérations d'une même boucle, un caractère caché (<CTRL-A> ou '\001' en notation **C**) que l'utilisateur ne peut inclure dans son fichier source; la version finale de la syntaxe interne d'une boucle while devient donc:

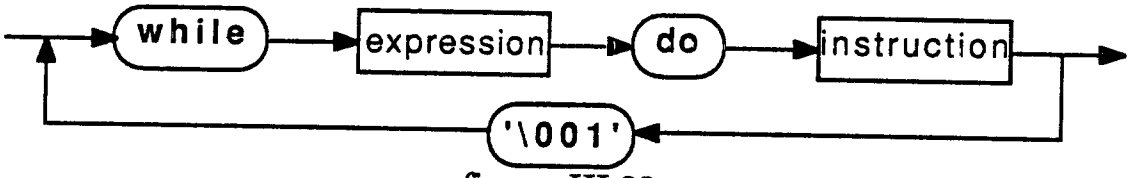


figure III.23

Les manipulations du pointeur de décodage deviennent donc un peu plus compliquées: en fin de boucle, le pointeur de décodage nextchar est déplacé de telle manière qu'il pointe sur une zone mémoire (**bidbuf[]**) contenant un <CTRL-A> (instructions **while3**) ; une fois que ce caractère a été analysé, nextchar est ramené sur le début de la boucle while (instructions **while1**). Cependant, les opérations de modification de nextchar ne représentent qu'une partie des instructions citées ci-dessus; en effet, la plus grande partie des opérations exécutées lors de l'analyse d'une boucle ont pour rôle d'empiler et de dépiler certaines variables de l'interpréteur.

III.3.6.4 Manipulations de pile

Instructions **while 2**:

En début de boucle le traitement est classique: la valeur actuelle d'evflag est empilée, si evflag était vrai, sa nouvelle valeur est celle de l'expression contrôlant la boucle while, puis le compteur de boucles imbriquées (**bcompte**) est incrémenté. Nous avons vu précédemment que le traitement d'une boucle **while** imposait à l'interpréteur de revenir en arrière chaque fois qu'une fin de boucle était rencontrée. Ceci impose donc, de sauver en début de boucle une partie du contexte de l'interpréteur pour le restaurer en fin de boucle avant d'analyser une nouvelle itération de la boucle. Aussi, en début de boucle, une variable structurée décrite ci-dessous est créée et empilée:

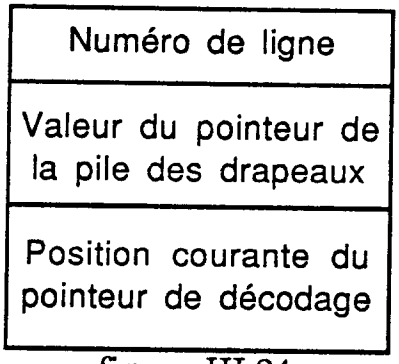


figure III.24

Lors de la création de la structure, la valeur du pointeur de décodage *nextchar* est celle qu'il avait alors qu'il pointait sur le w du while commençant la boucle (rappelons qu'à l'inverse de tous les mots-clé qui n'ont pas de valeur associée, que l'analyseur lexicographique yylex() associe au mot-clé while une valeur égale à la position actuelle du pointeur de décodage); le numéro de la

lors du retour en arrière); la valeur actuelle du pointeur de la pile des drapeaux est aussi mémorisée, ceci afin de permettre le traitement des instructions **break** et **continue**, comme nous le verrons par la suite. Cette structure sera mémorisée dans une pile spéciale: **pilep[]**. De la même façon que pour la pile des flags, l'empilage et le depilage se font respectivement par le biais de deux fonctions: **pempile()** et **pdepile()**.

Instructions **while3**:

En fin de boucle, la valeur actuelle d'evflag est mémorisée dans LastFlag avant qu'un élèment ne soit extrait de chacune des piles pilef[] et pilep[]. Si LastFlag est vrai (indiquant ainsi que la boucle qui vient d'être terminée était évaluable), evflag est modifié de manière à ce qu'il pointe sur la zone mémoire contenant le caractère "caché" <CTRL-A>; sinon (LastFlag faux), aucune modification n'a lieu et l'interpréteur passe à l'instruction suivante. Après rencontre du <CTRL-A>, l'interpréteur récupère dans la dernière structure dépilée de pilep[] les valeurs du pointeur de décodage et du compteur de ligne et les réaffecte respectivement à nextchar et nligne: une nouvelle exécution de la boucle est alors possible.

En tête de boucle:

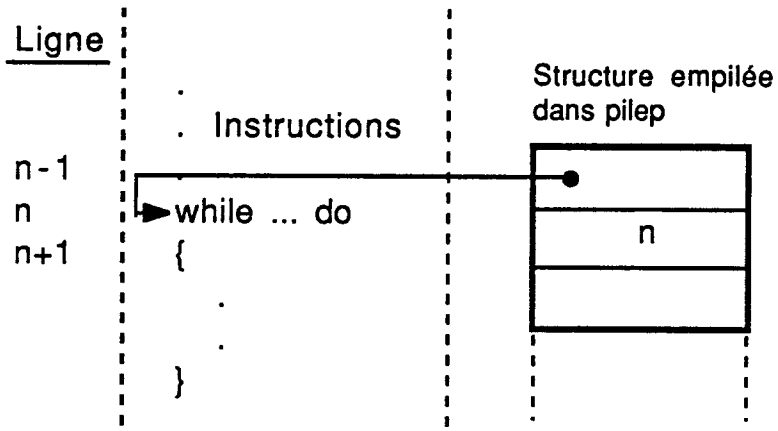


figure III.25

En fin de boucle les valeurs de nextchar et nligne mémorisées dans les deux premiers champs de la structure empilée seront restaurées; la boucle pourra être analysée de nouveau.

III.3.6.4.1 Traitement des ordres break et continue à l'intérieur d'une boucle while

Dès qu'un ordre "**break** n" ou "**continue** n" (où n indique le nombre de boucles dont il faut sortir) est rencontré, l'interpréteur appelle la fonction **brkhandle** en lui passant comme arguments le type d'ordre à traiter et la valeur de n (si n est absent, sa valeur par défaut est de 1). Cette fonction va vérifier dans un premier temps que le nombre passé en argument n'est pas supérieur au nombre de boucles actuellement imbriquées; si tel était le cas, une erreur de syntaxe serait générée et l'interprétation arrêtée.

La figure suivante montre l'état des deux piles pilep[] et pilef[] au moment où la fonction brkhandle est appelée.

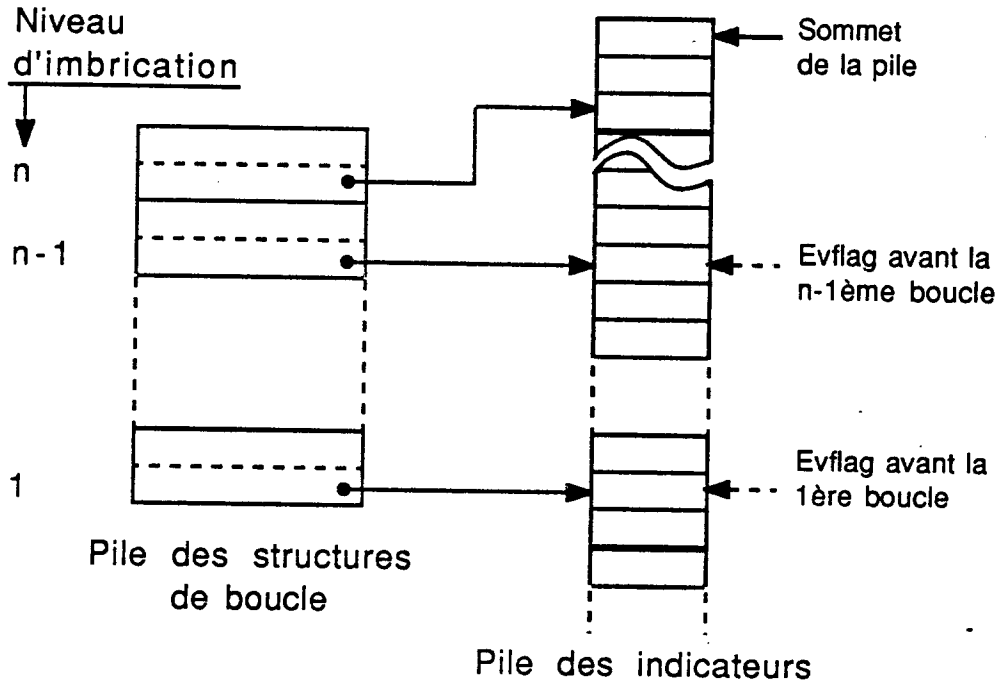


figure III.26

Dans tous les cas, brkhandle va récupérer dans la n^{éme} structure de boucle à partir du haut de la pile pilep[], la valeur du pointeur vers la pile des indicateurs d'évaluation pilef[]; cette valeur est celle qu'avait le pointeur pf_ptr en début de la boucle concernée par le break ou le continue (si m est le nombre de boucles imbriquées, il s'agit de la (m-n+1)ème boucle); nous noterons symboliquement cette valeur pf_ptr_n. Tous les indicateurs depuis le haut de la pile pilef[] compris et pf_ptr_n sont rendus FAUX de manière à interdire ainsi l'exécution de tous les blocs contenus à l'intérieur de la boucle concernée.

Dans le cas de l'ordre **continue**, la fonction brkhandle ramène les deux pointeurs de pile sur les positions qu'ils avaient avant l'entrée dans la boucle concernée (en ce qui concerne la pile pilep[], cette opération équivaut à un dépilage de n structures). L'exemple ci-dessous montre l'état des deux piles après l'exécution d'une instruction:

continue *n-1*

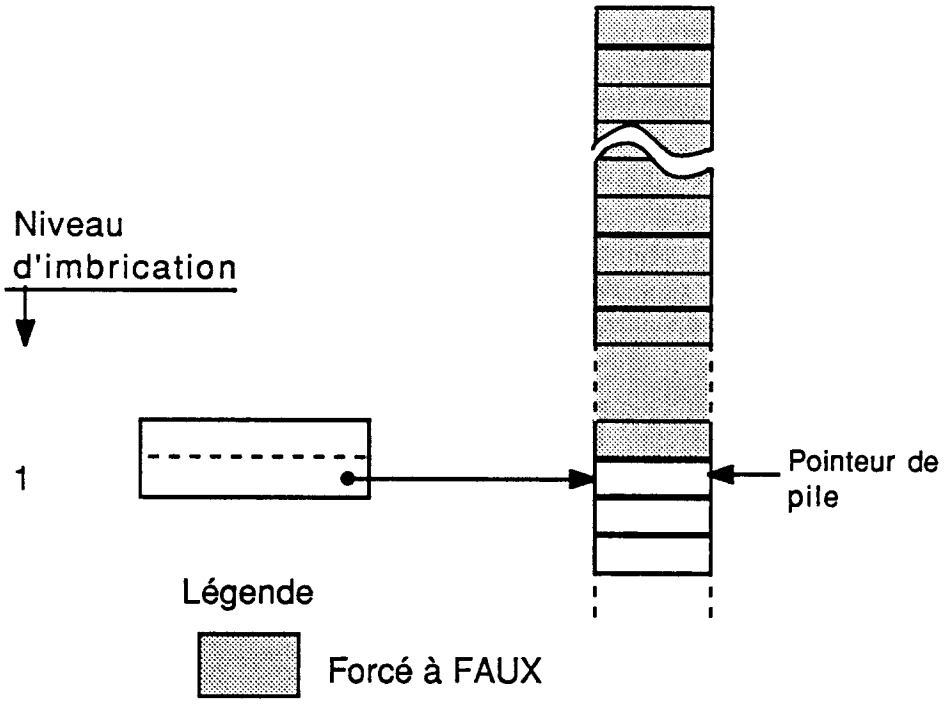


figure III.27

Dans le cas d'un ordre **break**, après retour de la fonction brkhandle, l'interpréteur positionne la variable evflag à FAUX de manière à interdire l'exécution de toutes les instructions suivant l'ordre break dans le bloc structuré le plus interne; l'analyse syntaxique des n boucles est effectuée mais aucune action n'est effectuée du fait du positionnement à FAUX de tous les indicateurs d'évaluation. L'exemple ci-dessous montre l'état des deux piles après l'exécution d'une instruction:

break *n-1*

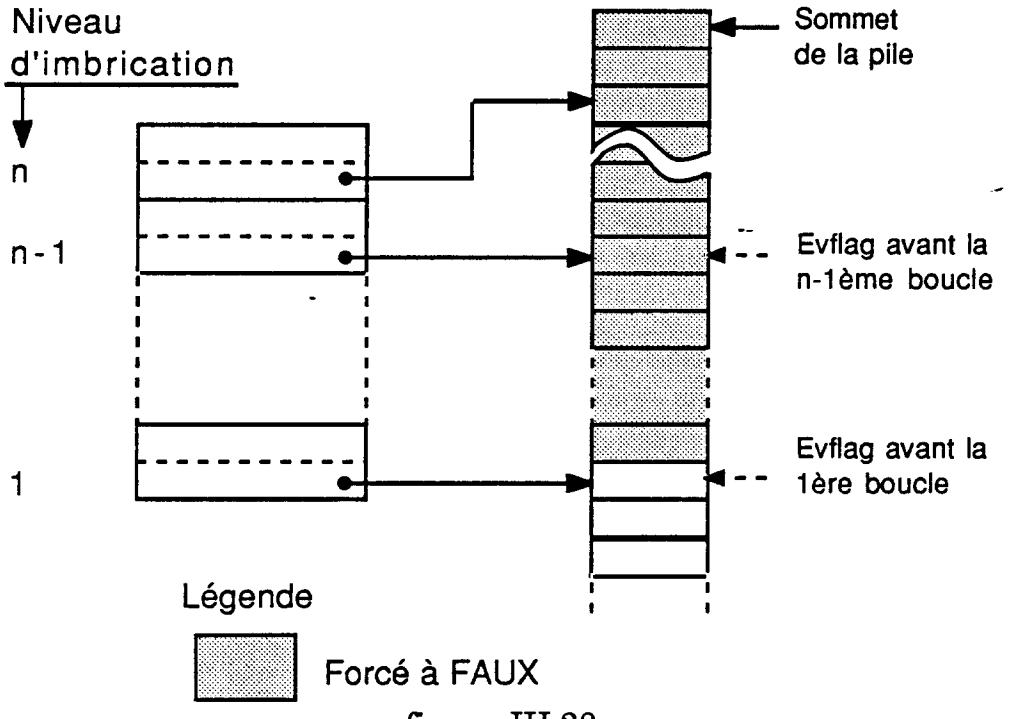


figure III.28

Exemple:

```
Instructions                    Notes
...
while (...) do {
        ...
        while (...) do {
                ...
                if(...) {
                        ...
                        break 2;
                        ...             (1)
                }
                ...                     (2)
        }
        ...                             (2)
}
```

figure III.29

(1) L'exécution des instructions suivant l'ordre break à l'intérieur du bloc structuré le plus interne est interdite par le fait qu'evflag a été positionné à FAUX par l'interpréteur.

(2) L'exécution de toutes ces instructions est interdite du fait des manipulations effectuées par la fonction brkhandle.

Dans le cas d'un ordre **continue**, le traitement est un peu plus complexe: il faut en effet revenir au début de la boucle; nous avons préféré "remonter" directement au début de la boucle, plutôt que de l'analyser entièrement sans exécuter les instructions comprises entre l'ordre continue et la fin de la boucle puis d'utiliser le mécanisme normal de remontée en début de boucle. La première des deux solutions est, en effet, beaucoup plus efficace s'il y a de nombreuses instructions entre l'ordre continue et la fin de la boucle; cependant, cette solution implique le déclenchement du mécanisme de récupération d'erreur de YACC de manière à forcer l'interpréteur à "oublier" le fait que n boucles (et éventuellement les bloc structurés qui y sont contenus) n'ont pas été "syntaxiquement fermées". Comme nous utilisons pour la première fois le mécanisme de récupération d'erreurs, nous décrirons ci-dessous le traitement des erreurs que permet YACC.

Le fonctionnement par défaut du mécanisme de récupération d'erreurs offert par YACC peut se résumer ainsi: chaque fois que l'analyseur syntaxique détecte une erreur, il fait appel à la fonction yyerror(), fonction généralement chargée d'afficher les messages d'erreurs, et éventuellement de prendre la décision d'arrêter toute analyse.

Cependant, ce mécanisme par défaut est un peu frustre: dans le cas d'un compilateur, il est en général hors de question de s'arrêter dès qu'une erreur de syntaxe est détectée: l'analyse syntaxique continue de manière à montrer à l'utilisateur toutes les erreurs de syntaxe qu'il a pu commettre. Cependant, l'oubli d'un simple caractère de ponctuation, par exemple, peut générer de nombreuses erreurs en cascade; aussi, après la détection d'une erreur YACC permet de reprendre l'interprétation en des points bien précis; pour celà, il

est en général suivi d'autres éléments terminaux et est traité de la façon suivante: si une erreur est détectée, l'analyseur syntaxique explore élément terminal par élément terminal le fichier d'entrée jusqu'à trouver une séquence d'éléments terminaux identique à celle suivant l'élément terminal **error**, l'analyseur syntaxique est alors resynchronisé; tous les éléments terminaux trouvés entre le moment où l'erreur a été détectée et celui où elle a été "récupérée" sont tout simplement sautés.

Exemple:

Considérons un langage compilé où le séparateur d'instruction est le caractère ';' (Ex: **C**, ou pourquoi pas une version compilée de CLIPS). L'élément (non terminal) syntaxique *instruction* pourrait être ainsi décrit en terme de syntaxe d'entrée YACC:

```
instruction    : ordre ';'
               | '{' instructions '}'
               | error ';'
               ;

instructions   : instruction
               | instructions instruction
          ;

ordre          : bloc_if
               | bloc_while
               | bloc_case
               | ...
               | affectation
               ;
```

Du fait de ces règles, en cas d'erreur de syntaxe détectée à l'intérieur d'une instruction, l'analyseur syntaxique sautera tous les caractères dans le fichier d'entrée jusqu'à trouver un ';' , il considère alors que l'instruction erronée est terminée et qu'il peut passer sans risque (du moins on l'espère) à l'analyse de l'instruction suivante.

Enfin, il existe un moyen de signaler en cours d'analyse à l'analyseur syntaxique une erreur alors que la phrase analysée est syntaxiquement valable: ce peut être, par exemple, le cas dans une affectation décrite sous la forme:

```
affectation    : identificateur '=' expression
```

où tous les élèments qui la composent sont corrects, mais où l'identificateur et l'expression ne sont pas du même type. Pour cela, il existe une macro-instruction: **YYERROR** qui signale une erreur à l'analyseur syntaxique et qui permet, au cas où cela est explicitement prévu, de déclencher le mécanisme de récupération d'erreurs.

Etudions tout d'abord le mécanisme employé dans le cas d'un ordre continue sans argument: c'est le cas par défaut, il faut remonter au début d'une seule boucle. Si l'on fait abstraction des actions, la syntaxe d'un bloc while (wbloc dans le source yacc) telle qu'elle est connue de l'analyseur syntaxique est la suivante:

```
wbloc    : WHILE expr DO ordre_sp              (1)
         | WHILE expr DO error '\003'      (2)
         ;
```

Dès qu'un ordre continue vient d'être traité par la fonction brkhandle, l'interpréteur déplace le pointeur de décodage nextchar vers une zone mémoire contenant le caractère '\003' (ou <CTRL-C>), et positionne la variable *errflag* (qui indique le type d'erreur détectée) à la valeur NOERROR de façon à ce que la fonction yyerror() ne génère aucun message ni n'arrête l'interprétation, puis déclenche le mécanisme d'erreur par l'utilisation de la macro-instruction YYERROR. Une fois l'erreur recupérée par la règle (2) ci-dessus (tout le bloc while est considéré comme terminé), le pointeur de décodage est amené sur la zone mémoire contenant le caractère '\001' (séparateur caché de boucles whiles) de manière à retomber sur le traitement classique et à ré-analyser la boucle si son expression de contrôle est toujours vraie.

Le cas d'un ordre continue suivi d'un argument **n** est une généralisation du cas précédent: il suffit de déclencher n fois le mécanisme précédent de manière à "remonter" n boucles: la variable *ContCompte* initialisée à n en début de traitement est décrémentée chaque fois qu'une "erreur" vient d'être récupérée; une fois qu'elle devient nulle le traitement normal peut reprendre.

Le déclenchement du mécanisme est effectué par les instructions annotées **cont1**, la récupération, par les instructions **cont2**.

III.3.6.5 Traitement de l'appel à un sous-programme

Une solution élégante pour traiter les appels à des sous-programmes à l'intérieur d'un module LIPS, eût été, après analyse et vérification de la validité de l'instruction d'appel au sous-programme, que l'interpréteur se rappelle de manière récursive. Cette solution impose évidemment que l'interpréteur soit réentrant. Or l'interpréteur, comme nous l'avons vu précédemment, est composé de plusieurs modules C (analyseur syntaxique, analyseur lexicographique, fonctions input() et unput()) qui communiquent entre eux par des variables communes.

Le langage **Pascal** offre plusieurs niveaux de visibilité de variables:

soit le programme:

```
Procedure P1
    VAR x1: INTEGER;
    ...
    Procedure P11
    BEGIN
        ...
    END;            (* Of P11 *)

    Procedure P12
    BEGIN
        VAR x12: INTEGER;
        ...
    END;            (* Of P12 *)
BEGIN               (* Of P1 *)
    ...
```

Une variable en Pascal n'est visible que dans la procédure où elle est définie et dans toutes les procédures qui y sont contenues. Ainsi la variable x12 n'est visible qu'à l'intérieur de la procédure x12 (c'est une variable dite "locale"), par contre la variable x1 est visible à l'intérieur de P1, donc de P11 et P12, mais pas à l'extérieur de P1.

Un tel système de gestion des variables permet de rendre réentrant un groupe de fonctions qui se partagent des variables communes. Or en **C**, dès qu'une variable n'est pas locale (donc dynamique: c'est-à-dire allouée provisoirement dans la pile), elle devient statique (elle réside au même endroit en mémoire quelles que soient les circonstances); de ce fait, la réentrance d'un groupe de fonctions se partageant des variables communes est compromise. Trois solutions sont alors possibles:

- Soit supprimer les variables globales: les différentes fonctions se passent alors les valeurs communes sous forme de paramètres.
- Soit empiler toutes les variables communes avant tout appel réentrant au groupe de fonctions et les dépiler en sortie; on simule ainsi ce que ferait Pascal de façon transparente à l'utilisateur.
- Développer en Pascal !!!

Aucune de ces trois solutions n'est satisfaisante: la troisième est rapidement éliminée car YACC ne génére que du code **C** ([KERN 78]); les deux premières imposent de modifier le code généré par YACC ce qui est loin d'être évident.

Comme nous le voyons, il est très difficile de rendre l'interpréteur ré-entrant; nous nous sommes donc orientés vers une solution moins "élégante" mais qui permettait de pallier relativement facilement cet inconvénient:

La syntaxe interne de l'instruction d'appel à un sous-programme est (une fois encore) différente de celle qui est donnée dans les diagrammes syntaxiques:

Si l'instruction d'appel à un sous-progamme (abrégé S/P) est décrite comme:

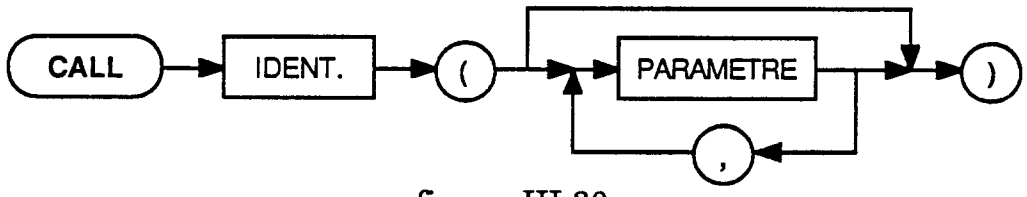


figure III.30

elle est en fait implantée comme:

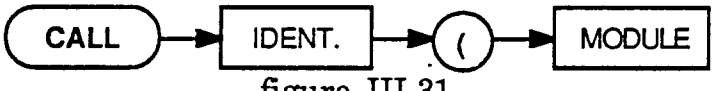


figure III.31

où un module est défini comme:

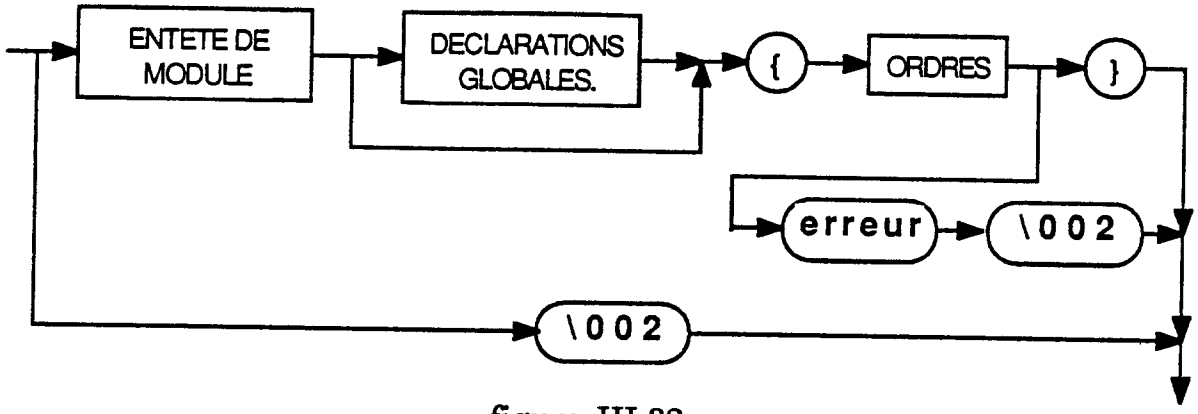


figure III.32

Dès que l'interpréteur reconnaît la première partie d'un appel à un S/P, il recopie les éventuels paramètres d'appel du S/P dans un tampon mémoire grâce à la fonction **skiparg()**[1] (s'il n'y a pas d'erreur de syntaxe dans l'appel au S/P, le pointeur de décodage doit être ainsi amené entre la parenthèse fermant la liste des paramètres et le ';' marquant la fin de l'instruction); après quoi, l'interpréteur teste la valeur de l'indicateur d'évaluation evflag, deux cas sont alors possibles:

a) Evflag est FAUX:

Il n'y a donc pas lieu d'analyser le S/P. Or, après la parenthèse ouvrante, l'analyseur syntaxique attend un module; il faut donc satisfaire cette règle sous peine d'erreur de syntaxe. Pour cela, l'interpréteur mémorise la position actuelle du pointeur nextchar puis le positionne a l'intérieur d'une zone mémoire contenant le caractère '\002'. Ceci fait, ce caractère est ensuite lu par l'analyseur lexicographique, la règle suivante décrivant un module "fantôme":

```
module  : '\002'
        ;
```

est alors satisfaite ainsi que celle définissant un appel à un S/P; l'interpréteur réaffecte alors au pointeur nextchar la valeur qu'il avait avant la manipulation précédente, et le tour est joué. Ce traitement est effectué par les instructions annotées **CALL4** ainsi que par la dernière instruction du bloc **CALL5**.

b) Evflag est VRAI:

Dans un premier temps l'interpréteur vérifie par un appel à la fonction **chkmod()** que le module appelé a bien été chargé en mémoire. Soit chkmod renvoie une valeur nulle (le module appelé est inexistant), l'interprétation est alors arrêtée un message d'erreur est envoyé à l'utilisateur. Soit le module demandé existe bien, auquel cas chkmod renvoie un pointeur sur le descripteur du module appelé (cf. structure paragraphe III.3.3.1). Après vérification que le nombre maximal de modules imbriqués n'est pas dépassé, l'interpréteur mémorise les valeurs du pointeur de décodage, du compteur de ligne et du pointeur de la pile pilef[] dans la structure adéquate (cf. plus haut, paragraphe sur le traitement des boucles) qu'il empile (instructions **CALL2**) avant d'initialiser les compteurs de ligne et de boucles imbriquées puis de positionner

---

1 Tout en recopiant les arguments, skiparg() compresse la zone argument: toute suite de

le pointeur de décodage sur le premier caractère du module appelé (instructions **CALL3**) qui est ensuite analysé. Une fois le module appelé terminé, l'interpréteur libére les variables locales à ce module (appel à la fonction **clearniv**()) puis dépile la structure empilée avant l'analyse du S/P; les valeurs contenues dans cette structure sont ensuite réaffectées aux variables correspondantes (instructions **CALL5**).

III.3.6.6 Traitement de l'instruction exit

Dès que cet ordre est rencontré, l'interpréteur appelle (à condition qu'evflag soit vrai, bien sur) la fonction **killprog**(). Cette fonction à pour rôle de libérer l'espace occupé par les variables dont le niveau est compris entre 1 (programme principal) et le niveau actuel, ce qui revient à effacer toutes les variables non globales; après cette opération, la fonction killprog() positionne le pointeur de décodage nextchar sur une zone mémoire contenant un octet nul (qui par convention marque la fin du flot d'entrée de l'analyseur syntaxique), ce qui provoquera l'arrêt de l'interprétation lors du prochain appel à yylex(). La macro-instruction **YYACCEPT** qui suit l'appel à killprog() indique à l'analyseur syntaxique qu'il est dans un état où il peut valablement rencontrer une fin de flot d'entrée; si cette macro-instruction n'était pas présente, l'analyseur syntaxique détecterait une erreur de syntaxe en trouvant une fin de flot d'entrée alors que le module en cours n'est syntaxiquement pas terminé.

III.3.6.7 Traitement de l'instruction return

Si l'instruction return est rencontrée à l'intérieur du programme principal, elle équivaut à une instruction exit; dans ce cas, le traitement est strictement identique à celui de l'instruction exit (instructions **return1**).

Par contre, si l'instruction return est rencontrée à l'intérieur d'un S/P, il faut immédiatement quitter le sous-programme actuel sans analyser les instructions qui peuvent suivre. Pour cela, l'interpréteur positionne le pointeur nextchar sur une zone mémoire contenant le caractère '\002' et signale une "fausse" erreur de syntaxe; le mécanisme de récupération d'erreurs de YACC va ensuite amener l'état interne de l'analyseur syntaxique en une position où le caractère '\002' est valide après une erreur, la seule règle satisfaisante est:

```
module    : entet_mod dcl_glos '{' ordres error '\002'
```

De ce fait l'analyseur syntaxique considére le module en cours comme terminé.

Cette manière de faire permet un retour immédiat au programme appelant: elle évite d'avoir à analyser les instructions suivant l'ordre return sans les exécuter.

III.3.6.8 Traitement de l'appel à une primitive

Dans ce paragraphe nous traiterons indifféremment de l'appel d'une primitive depuis l'interpréteur CLIPS ou de la synchronisation d'un groupe de primitives: partout où est utilisé le mot primitive, il est possible de le remplacer par "expression synchronisée de primitives". Un appel de primitive peut apparaitre dans deux cas:

(1) Soit comme une instruction simple.

(2) Soit comme composant d'une expression entière. En pratique les cas les plus fréquents sont:

int *variable* = *appel_de _primitive* ou
case (*appel_de _primitive*) ou
if (*appel_de _primitive*) ou
while (*appel_de _primitive*) ou
...

Un appel de primitive peut intervenir dans le calcul d'une expression entière car une primitive renvoie un résultat entier dont la signification varie suivant son signe:

Un code retour nul indique qu'il n'y a pas eu d'erreur, la primitive s'est bien terminée.

Un code retour négatif indique qu'une erreur a été détectée (primitive inexistante, erreur de syntaxe dans les arguments d'appel de la primitive,). Quel que soit le type d'appel (appel simple de primitive, ou appel de primitive intervenant dans une expression), un tel code retour est considéré comme irrécupérable: l'interpréteur terminera aussitôt l'exécution du programme en cours.

Un code retour positif indique un échec (Ex: déplacement cartésien de l'organe terminal d'un robot impossible sans changement de nappe, arrêt d'une action suite au déclenchement d'une garde, ...). Un tel cas de figure est considéré comme récupérable, donc si l'utilisateur a prévu une procédure de récupération d'échec (l'appel à la primitive intervient dans une expression), l'interpréteur n'arrêtera pas le déroulement du programme en cours; par contre dans le cas contraire (appel simple de primitive), un échec implique l'arrêt du programme.

Comme le superviseur de LIPS possède les routines permettant l'interprétation d'une phrase de type "appel de primitive", le traitement effectué au niveau de l'interpréteur se réduit à sa plus simple expression (cf. instructions **app_prim**): par un appel à la fonction skiparg(), l'interpréteur recopie l'appel à la primitive dans une zone mémoire tampon qu'il transmet à la fonction ykexec() qui assure l'interfaçage avec le superviseur: préparation du décodage de la zone tampon et transmission au superviseur, suivie de la mise en attente du compte-rendu de ce dernier; la fonction ykexec() transmet en retour ce compte rendu à l'interpréteur qui déclenche le traitement approprié suivant le signe:

S'il est négatif, l'analyseur syntaxique marque qu'il a rencontré une erreur détectée par le superviseur (la variable errflag, indiquant le type d'erreur reçoit la valeur SSERR) puis signale l'erreur au "moteur" YACC par l'emploi de la macro YYABORT.

III.3.7 Gestion des variables LIPS

La plupart des règles non encore décrites concernent la gestion des variables (affectation, évaluation des expressions, ...). Du fait de raisons histo-

riques[1], le traitement des variables de type entier est dissocié de celui des variables des autres types.

Dans tous les cas, ce sont des fonctions du gestionnaire de variables qui permettent d'affecter une valeur à une variable ou de récupérer la valeur d'une variable. La différence de traitement entre les variables entières et les autres se situe au niveau du calcul d'une expressions: quand l'analyseur syntaxique rencontre une expression entière, il demande au gestionnaire des variables la valeur de toutes les variables composant l'expression puis il effectue lui-même le calcul de l'expression; quand il rencontre une expression d'un autre type, il laisse le soin au gestionnaire des variables d'analyser et d'évaluer l'expression.

III.3.7.1 gestionnaire des variables et traitement des expressions

III.3.7.2 Principes généraux du fonctionnement du gestionnaire des variables

Le gestionnaire des variables utilise une structure à double indirection pour consituer la table des symboles connus: il comprend tout d'abord un tableau de pointeurs; pour chaque niveau une entrée dans ce tableau pointe vers un nouveau tableau de pointeurs, cette fois-ci organisé par types. Pour chaque type, le pointeur associé est dirigé vers la première variable dont le niveau et le type correspondent.

---

[1] Lorsque nous avons défini les caractéristiques de l'interpréteur, il existait déjà une couche logicielle capable de traiter des expressions simples mettant en jeu des variables surtout

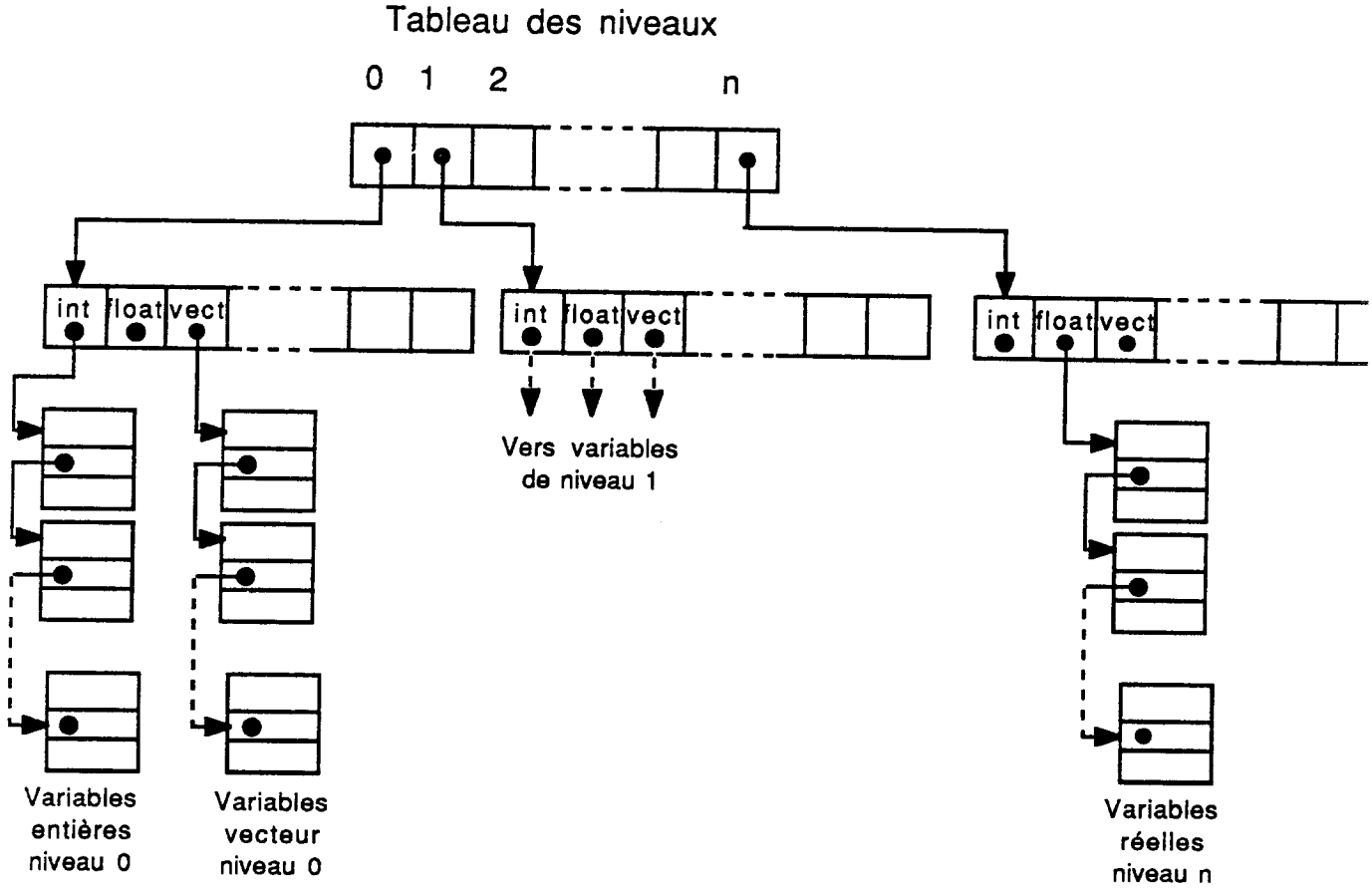


figure III.33

Chaque variable dans la liste est représentée par la structure suivante:

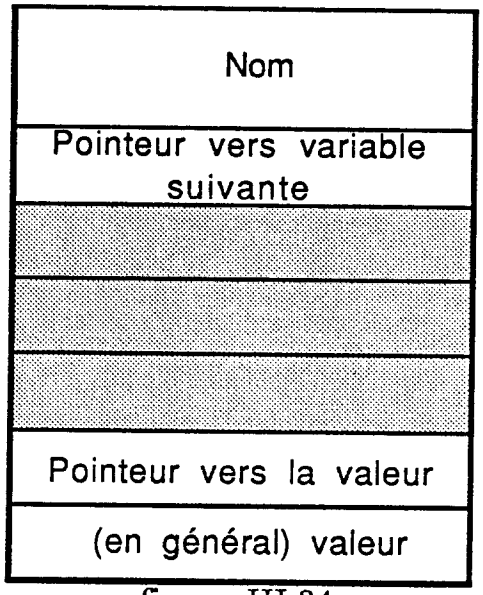


figure III.34

Le premier champ (chaîne de 16 caractères, octet nul final compris) est le nom de la variable.

Le deuxième est un pointeur sur la variable suivante dans la liste.

Les troisième à cinquième champs compris sont inutilisés par la majorité

Le sixième est un pointeur vers la zone contenant la valeur de la variable; à l'exception des variables système (non destructibles), la valeur de cette variable est stockée dans le septième champ immédiatement contigu.

Chaque fois qu'une variable est créée, le gestionnaire des variables alloue dynamiquement (par demande d'un bloc mémoire dans une ressource espace mémoire réservée à cet usage) une structure du type précédent pour enregistrer la valeur de la variable. A l'inverse, quand une variable est détruite (soit lors de la terminaison d'un S/P, ou par la commande **kill** du superviseur) la zone mémoire contenant la valeur est rendue à la ressource.

III.3.7.2.1 Cas particulier des variables de type repère

Les repères sous LIPS ont la particularité de pouvoir être chaînés: il est possible de lier un repère *r1* à un autre *r2* par une situation (ou attitude suivant la terminologie LIPS) définissant la transformation géométrique qui ferait coïncider le trièdre de référence de *r1* avec celui de *r2*. Si la position absolue de *r1* évolue, celle de *r2* évoluera en conséquence. Cette particularité est très importante, surtout en ce qui concerne les repères mobiles: le gestionnaire des variables de LIPS connaît une variable de type repère dénommée **rm** (pour repère mobile) contenant la position de l'organe terminal du robot, cette position est mise à jour en permanence par le noyau du robot. Il est possible d'avoir des outils déportés par rapport au point terminal, et ainsi de définir des trajectoires dans le repère outil et non dans le repère mobile.

Exemple:

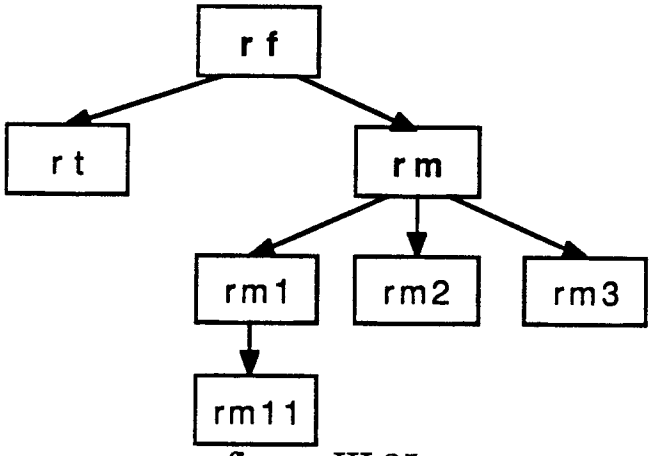


figure III.35

Dans l'exemple ci-dessus, les deux repères en gras sont les repères "systèmes" **rf** (repère fixe, origine de tous les repères) et **rm** (repère mobile). Un repère de travail: rt est défini par rapport à rf, et trois repères rm1 à rm3 sont liés au repère mobile; enfin, le repère rm11 est lié au repère rm1.

Soit un robot SCARA portant en bout de son axe terminal (orienté suivant Z) une pince. Du fait de la mécanique de la pince, son entremors est décalé d'un certaine quantité Δz, vers le bas par rapport au point terminal. Nous lierons donc le repère pince au repère mobile par une situation correspondant à une translation Δz. Lorsque l'on désire prendre un objet, il est plus naturel de raisonner au niveau du repère pince: raisonner au niveau du repère mobile oblige à ajouter ou à retrancher systématiquement la quantité Δz dans tous les

"indigestes" si les deux repères précédents étaient liés par une situation plus compliquées (composantes de translation plus composantes de rotation simultanées).

Pour représenter une structure arborescente comme celle des repères liés les uns aux autres, sans préjuger du nombre maximal de liaisons que peut avoir un même repère, il existe une solution informatique relativement simple:

L'arbre de l'exemple précédent peut être implanté de la manière suivante:

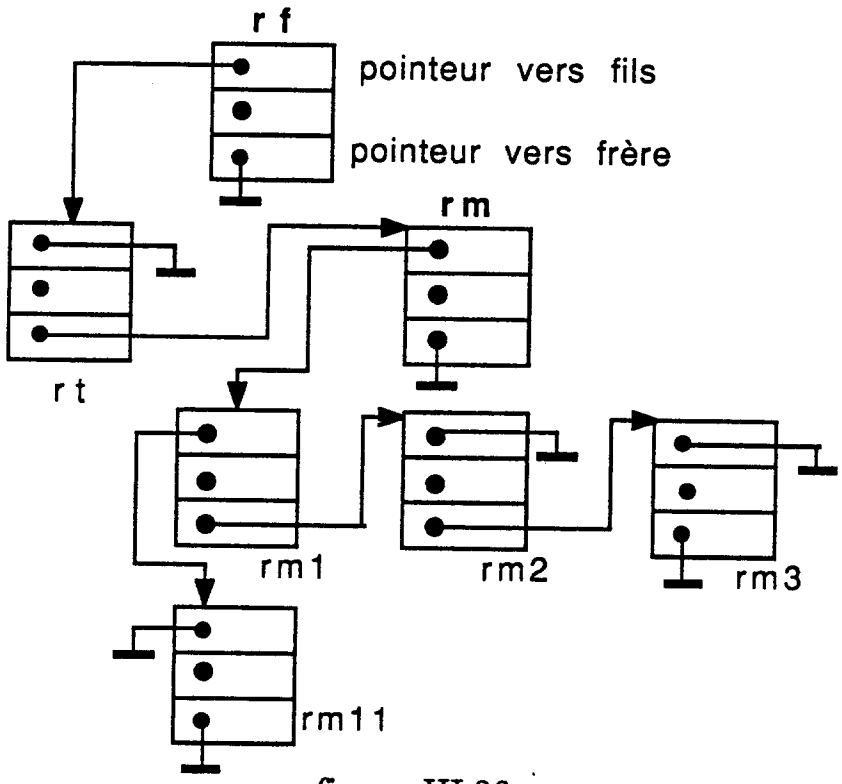


figure III.36

Notations: si *r2* est lié à *r1*, nous dirons que *r1* est le père de *r2* , ou que *r2* est le fils de *r1* . Si *r1* et *r2* sont les fils d'un même repère *r0*, nous dirons qu'il sont frères.

Chaque noeud de l'arbre possède un pointeur vers UN fils (ce pointeur pouvant être nul si le repère ne posséde aucun autre repère qui lui est lié) et un pointeur vers son frère suivant (si le repère n'a pas de frères, ce pointeur est nul).

Cependant, il est nécessaire de connaître le père de chaque noeud (notamment pour le calcul de la position absolue d'un repère, où il est nécessaire de remonter jusqu'au repère **rf**, racine de l'arborescence, en effectuant le produit de toutes les situations intermédiaires). De ce fait, chaque noeud doit posséder un pointeur vers son noeud père. Ce qui nous donne la structure finale suivante:

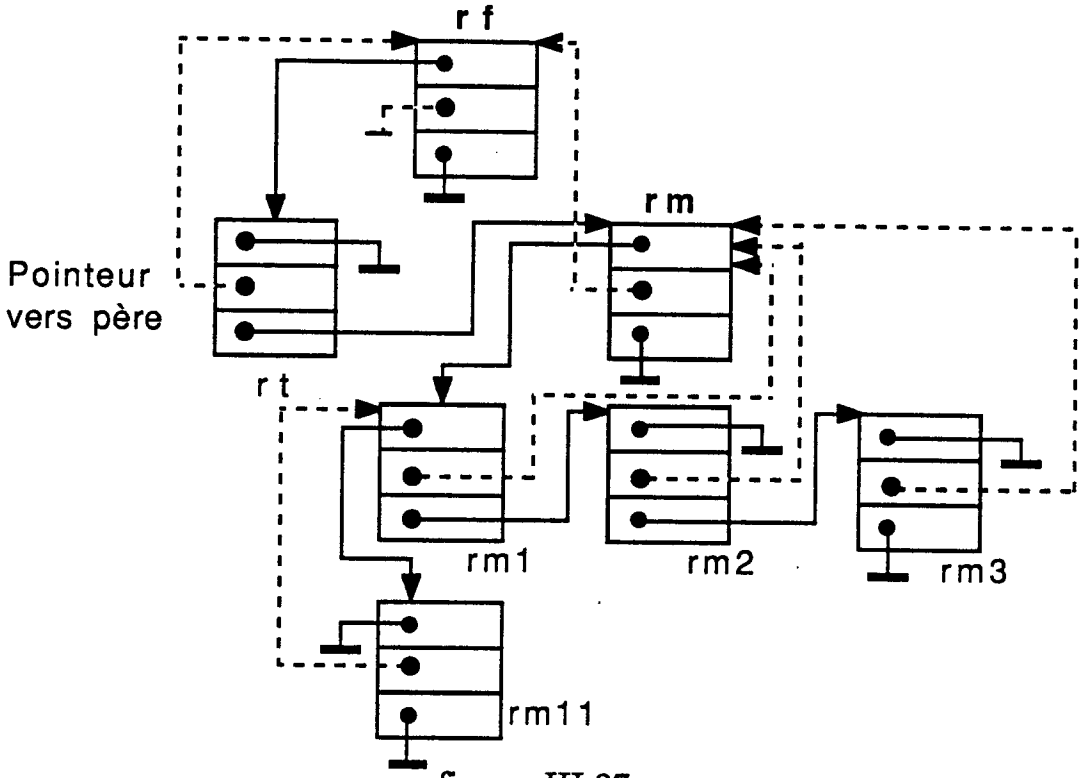


figure III.37

Ces trois liens sont contenus dans les trois champs que nous n'avions pas décrits dans le paragraphe concernant la structure des tables du gestionnaire des variables.

III.3.7.2.2 Stratégies d'exploration de l'arborescence:

III.3.5.2.2.a Destruction d'un repère

Lorsqu'un repère est détruit, tous les repères qui lui sont liés doivent l'être. Pour cela, il est nécessaire de suivre une exploration récursive de l'arbre schématisée par l'algorithme suivant:

```
Détruire(repère)
Debut
    Si repère à un fils Alors
        Détruire (repère->fils)
    FinSi
    Mémo = repère->frère
    Restituer l'espace mémoire occupé par repère
    Si Mémo existe Alors
        Détruire (Mémo)
    FinSi
Fin
```

figure III.38

L'interface entre l'analyseur syntaxique et le gestionnaire des variables est assurée par un groupe de fonctions que nous décrirons ci-dessous.

Le traitement des variables globales (dont la déclaration doit s'effectuer

global est suivi d'un nom de type puis d'une liste de variables; l'interpréteur explore cette liste et appelle pour chaque variable la fonction allog.

Syntaxe:
char *allog(char *type, char *nom)

Rôle:
Créer, si elle n'existe pas déjà, la variable globale de nom *nom* dans le type *type*.

Valeur rendue:
NULL si tout s'est bien passé.
Sinon, un pointeur sur le message d'erreur (en général, saturation de la table des symboles).

Nous avons déjà rencontré la fonction **clearniv**.

Syntaxe:
void clearniv(int niveau)

Rôle:
Détruire tous les variable du niveau passé en argument.

Valeur rendue:
Aucune

Pour le traitement proprement dit des expressions entières l'interface entre l'analyseur syntaxique et le gestionnaire des variables est réduite à deux fonctions: **recup** et **installe**.

Fonction **recup**:

Syntaxe:
void *recup(char *nom, int niveau)

Rôle:
La fonction recup explore tout d'abord la liste des variables du niveau spécifié à la recherche d'une variable de nom *nom*. Si cette recherche n'a pas réussi, la fonction recup la réitère au niveau global (ainsi, si deux variables de même nom existent en même temps au niveau global et au niveau actuel, seule la variable non globale sera visible).

Valeur rendue:
Cette fonction renvoie l'adresse à laquelle est implantée la variable cherchée, ou le pointeur NULL si la variable n'existe pas

Fonction **installe**

Syntaxe:
char *installe(char *nom, int valeur, int niveau)

Rôle:
La fonction installe exécute les deux premières étapes de la fonction recup (recherche d'une variable de nom *nom* au niveau désiré puis s'il y a lieu au niveau global). Si cette variable existe déjà à un de ces deux niveaux, sa valeur sera remplacée par celle passée en argument à la fonction installe. Si, au contraire, la variable n'existe pas, elle sera créée et recevra

Valeur rendue:
La fonction installe renvoie le pointeur NULL si tout s'est bien passé, ou un pointeur sur un message indiquant le type d'erreur rencontrée (en général, plus de place pour créer la variable).

La fonction recup est appelée par l'analyseur syntaxique chaque fois qu'une variable est rencontrée dans une expression entière. Après récupération de l'adresse de la variable, l'analyseur syntaxique détermine si l'opérateur d'adressage @ suivi d'un des modificateurs **b**, **w** ou **l** est présent après la variable; dans tous les cas l'analyseur syntaxique appelle la fonction **rdaddr**() en lui passant comme argument l'adresse de la variable suivie du modificateur d'adressage rencontré s'il est présent ou d'une valeur nulle s'il est absent. Dans le deuxième cas, la fonction rdaddr rendra tout simplement la valeur de la variable dont l'adresse lui a été passée; dans le premier cas, la fonction rdaddr effectue une indirection supplémentaire: la variable dont l'adresse lui a été passée est en fait un pointeur et rdaddr renverra alors l'octet, le mot ou le mot long ainsi pointé, suivant que le modificateur d'adressage était b, w ou l. (instructions **EXP8**).

Chaque fois que l'analyseur syntaxique rencontre une instruction d'affectation d'une variable entière (int ident '=' expression), il appelle, après évaluation de l'expression, la fonction installe() en lui passant comme arguments l'identificateur de la variable, la valeur de l'expression et le niveau actuel d'imbrication des programmes (clause else if... des instructions **EXP5**).

Si l'on reprend l'instruction précédente, mais que l'on fait suivre l'identificateur de variable de l'opérateur d'adressage, le traitement est différent puisque, dans ce cas, la variable est un pointeur et qu'il faut affecter le résultat de l'expression à la case mémoire dont l'adresse est contenue dans la variable. Pour cela, l'analyseur syntaxique appelle la fonction recup() pour obtenir la valeur de la variable (avec éventuellement génération d'une erreur de syntaxe si la variable n'est pas déclarée); une fois ceci fait, il apelle la fonction **wraddr**() chargée d'écrire le résultat de l'évaluation de l'expression dans la zone mémoire pointée par la variable (clause if($4)... des instructions **exp5**).

Une expression entière fait intervenir des opérandes et des opérateurs. Nous venons de voir comment les opérandes "atomiques" (constantes, identificateurs de variables, appels de primitives) étaient évalués séparément. Il nous reste à voir comment, après évaluation des atomes, est effectuée l'évaluation d'une expression contenant des opérateurs agissant sur plusieurs opérandes. Ce rôle est dévolu aux instructions annotées **exp7**: le fait que nous avons choisi la même syntaxe que **C** en ce qui concerne la représentation des opérateurs arithmétiques et logiques simplifie grandement la compréhension de ces quelques instructions. En bref, chaque opérateur di-adique est appliqué aux deux opérandes qui l'encadrent et chaque opérateur monadique est appliqué à l'opérande qui le suit. Au vu de ces instructions, il est possible de croire que les différentes règles sont sur un même pied d'égalité en ce qui concerne les priorités des opérateurs. Il n'en est rien, fort heureursement; l'affectatation de priorité aux différents opérateurs est effectuée par le biais des directives YACC

%left ETLOG OULOG à
%nonassoc '!' '~'

Ces directives spécifient pour chaque opérateur le sens d'évaluation si l'opérateur est associatif (Ex: %left '+' indique que l'opérateur + est associatif

et '~' sont déclarés non associatifs par la directive %nonassoc; cette restriction a été implantée pour rejeter des constructions du type *!!expression* (pourquoi faire simple, alors que l'on peut faire compliqué) strictement équivalentes à *expression* mais plus lourdes à analyser. Les opérateurs ainsi déclarés par une même directive sont de priorité identique; sinon, les opérateurs déclarés les derniers sont les plus prioritaires. Ainsi, l'ordre de priorité actuellement implanté sous CLIPS est le même que pour **C**, à savoir par ordre de priorité croissante:

&&	||		
|	^		
&			
!=	=		
<	>	>=	<=
+	-		
*	/	%	
!	~	-(unaire)	

figure III.33

Seul l'opérateur - unaire n'est pas déclaré explicitement par une des directives %left, %right ou %nonassoc; il est en fait implicitement déclaré dans la seule règle où il apparait:

'-' atome %prec '!' {$$ = -$2;}

ce qui veut dire que, quand il est employé comme opérateur unaire, '-' a la même priorité que '!' et non celle qu'il a d'habitude (la même que '+', soit nettement moins que lorsqu'il est employé comme opérateur unaire).

III.3.7.2.3 Expressions non entières

Chaque fois que l'analyseur syntaxique rencontre une instruction du type:

type ident '=' expression

où *type* est différent de int, toute l'instruction (depuis le premier caractère composant le type jusqu'au ';' qui la ferme) est copiée dans une zone mémoire tampon (appel à la fonction skiparg() déjà décrite), puis l'analyseur syntaxique appelle la fonction evalins() (pour EVAlue et INStalle) chargée de l'interfaçage entre l'interpréteur et le gestionnaire des variables . Cette fonction, appellera à son tour les routines standard du gestionnaire des variables chargées de l'évaluation des ordres du type :

set type variable '=' *expression*

envoyés à l'environnement LIPS.

III.3.7.2.4 Evaluation des paramètres d'appel d'un module

Un module peut avoir des paramètres d'entrée; ces derniers sont déclarés dans l'en-tête de module de la manière suivante:

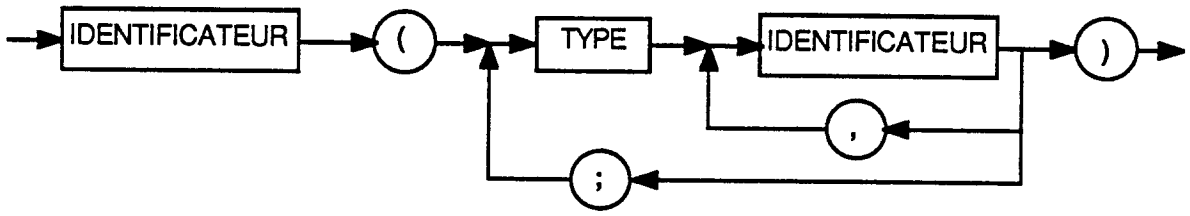


figure III.34

Le traitement des paramètres distingue deux cas:

- Le module en cours d'analyse est le programme principal de l'application, il a été appelé depuis le superviseur; dans ce cas, les éventuels paramètres à passer au module ont été définis dans la ligne de commande demandant l'exécution du module:

  call module '(' paramètres ')'

  Au moment de l'appel à l'interpréteur le superviseur transmet un pointeur sur la ligne de commande précédente. L'analyseur syntaxique mémorise alors dans la variable **parptr** la position du premier paramètre qu'il a reçu.

- Le module a été appelé depuis un autre programme. Nous avons vu plus haut (§ III.3.4.5) que, quand l'analyseur syntaxique détectait un appel à un S/P, il recopiait les paramètres transmis dans une zone tampon interne: **buffet** (rappel: ceci est effectué grâce à la fonction skiparg()).

Suivant le cas, la zone mémoire contenant les paramètres est différente. Donc, dès que l'analyseur syntaxique détecte dans le module appelé une déclaration de paramètre, il mémorise dans la variable **cptr** la position du premier paramètre: soit pmodndx est différent de 0 (cas d'un S/P), cptr pointe sur la zone mémoire interne **buffet**; soit pmodndx est nul, auquel cas cptr reçoit la valeur de parptr. Ce traitement est effectué par les instructions **PARAM1**.

Après quoi, l'analyseur syntaxique va travailler "en parallèle" sur la liste des paramètres formels du module (ceux déclarés dans l'en-tête du module appelé) et sur la liste de paramètres réels (ceux passés par le module appelant ou sur la ligne de commande LIPS): l'analyseur syntaxique explore la liste des paramètres formels, il connaît ainsi pour chaque paramètre et son type et son identificateur puis appelle la fonction **evalparam()** en lui passant comme paramètres l'identificateur du paramètre (*ident*), son type, le niveau d'imbrication actuel et l'adresse du pointeur cptr. La fonction evalparam() est chargée de décoder l'expression pointée par cptr en fonction du *type* et de l'affecter à la variable d'identificateur *ident* . En fin de décodage, la fonction evalparam() laisse le pointeur cptr positionné juste derrière le dernier caractère analysé. Après quoi, l'analyseur syntaxique recherche la virgule séparatrice de paramètres, afin de se préparer pour l'analyse de l'expression correspondant au prochain paramètre éventuel. Ainsi, l'analyseur syntaxique décode petit à petit la liste des expressions représentant les paramètres réels en fonction du type de chaque paramètre déclaré dans la liste des paramètres formels.

CONCLUSION

IV CONCLUSION

Le travail que nous avons présenté tout au long de ce mémoire a consisté en l'étude des problèmes de synchronisation rencontrés lors du développement d'une application sur un calculateur temps réel.

Après un exposé sur les concepts employés dans les systèmes temps réel, nous avons présenté le système multi-processeurs multi-tâches CESAR sur lequel nous avons réalisé l'implantation pratique des logiciels développés dans le cadre de notre étude.

Notre souci de retrouver au niveau logiciel la décomposition fonctionnelle d'un système de commande en deux parties (partie commande et partie opérative) nous a conduit à la réalisation de deux outils complémentaires permettant une approche structurée de la synchronisation de processus sur un calculateur temps réel: la "tâche généralisée" et le "compilateur d'expressions logiques".

Sur la base du concept de tâche généralisée M. Hervé POILVE a développé le superviseur et le synchroniseur de l'environnement LIPS auquel nous avons ajouté l'interpréteur CLIPS.

Avec LIPS, nous disposons d'un ensemble de développement très ouvert qui peut s'adresser à deux catégories de personnes:

- D'une part au concepteur d'une loi de commande, qui, à partir d'un noyau (ou niveau réflexe) non connu de LIPS, peut bâtir et tester facilement de nouvelles primitives qu'il déclarera à LIPS.
- D'autre part, au développeur d'une application, qui a à sa disposition les primitives mises au point par le concepteur de la loi de commande, et qui, écrit un module interprétable employant les primitives précédemment définies.

L'ensemble logiciel LIPS a désormais atteint sa phase d'industrialisation puisqu'il équipe plusieurs contrôleurs de robot dans des laboratoires de recerche; il de plus au centre d'applications industrielles telles que la cellule d'assemblage CARMEN (utilisant le poignet actif [ROBE 87] et une baie de commande LIPS quadri-processeurs) ou qu'un robot manipulateur de plaquettes de silicium en salle blanche. Ajoutons, que dans le cas de ces deux applications, la possibilité d'exécuter séparément des primitives ajoutée à la présence d'un interpréteur ont été très utiles en phase de mise au point: le test d'une nouvelle version d'un module se résumant à une session sous éditeur de texte suivie d'un chargement de la nouvelle version sur la baie de commande (pas de phase de compilation). Précisons qu'à l'heure ou nous écrivons ces lignes, l'environnement LIPS a été porté sur une baie de commande à base de MC68020 et de son coprocesseur arithmétique flottant MC68881. Dans cette version tous les types non entiers sont représentés en virgule flottante double précision au format IEEE (au lieu d'un format spécial sur 16 bits) et que toutes les fonctions de changement de répères et de transformation de coordonnées utilisent le coprocesseur arithmétique.

Notre étude demande cependant à être poursuivie:

D'une part, l'utilisation du "compilateur d'expressions logiques" et de la "tâche généralisé" implique la programmation en assembleur alors qu'à l'heure actuelle des langages de haut niveau tels que **C** ou bientôt **ADA** deviennent des standards de fait pour la programmation d'applications temps réel. Il serait donc souhaitable de rendre possible l'utilisation des ces deux outils depuis de tels langages.

D'autre part, l'interpréteur CLIPS souffre d'une certaine lenteur (due en particulier au choix du mécanisme d'analyse "à la volée"). Afin d'optimiser ses performances et faciliter la mise au point des modules LIPS sur le poste de développement plusieurs solutions sont envisageables:

- Porter l'interpréteur sur le poste de développement PC en supprimant la plus grande partie des actions associées aux différentes règles. De cette façon, il est possible de disposer sur le PC d'un vérificateur de syntaxe de manière à ne charger sur la baie de commande que des modules syntaxiquement valables. Ainsi, en phase de développement la baie de commande ne servirait qu'à la vérification de la logique des modules.
- Modifier sur le poste de développement la fonction de chargement/déchargement des modules ainsi que l'interpréteur de manière à ce que le code chargé sur la baie de commande soit "tokenisé". Cette solution permettrait un gain de vitesse appréciable lors de l'interprétation; mais nécessite une révision majeures de l'interface entre l'interpréteur et le superviseur.

Les deux dernières remarques constituent des objectif à court terme pour l'évolution de LIPS. A plus long terme, nous pensons qu'il serait intéressant que les extensions suivantes soient abordées et développées:

- L'ensemble interpréteur + synchroniseur rend des services inestimables pour la réalisation d'applications répétitives; mais il est peu adapté pour la prise en compte de certains évènements asynchrones par rapport au déroulement de l'application. Pour satisfaire ces besoins il serait utile d'associer au synchroniseur une tâche de type "automate programmable" capable de générer automatiquement des ordres d'activation de primitives en fonction de l'état de voies d'entrée/sortie logiques.
- D'autre part, nous assistons à "l'explosion" de l'informatique répartie. Dans un tel contexte, il serait souhaitable de généraliser le fonctionnement du synchroniseur à n machines réliées entre elles par un réseau.

ANNEXE A
COMPILATEUR
D'EXPRESSIONS LOGIQUES

Listing du compilateur

```
      IMPLICIT INTEGER*2 (A-Z)
      CHARACTER*80 LIGNE
      LOGICAL OKWAIT,OKIF,OKERR
      COMMON /LUFIC/LUIN,LUOUT,LUCON,LULST
      COMMON /INFO/LIGNE,NLIGN,NGEN,OKWAIT,OKIF,OKERR
      DATA NOWAIT,NOEND/-13,14/
C
C     EXPLORATION DU FICHIER SOURCE ET TRAITEMENT DE LA LIGNE LUE
C
      OKWAIT=.FALSE.
      OKIF=.FALSE.
      OKERR=.TRUE.
10    CONTINUE
        READ (LUIN,FMT='(A80)',END=9999,ERR=10000)LIGNE
        NLIGN=NLIGN+1
        IF (LIGNE(1:2).EQ.'*$') THEN
C
C         LIGNE DE SYNCHRO A ANALYSER
C
          CALL TRI (LIGNE(3:))
        ELSE
C
C       LIGNE D'ASSEMBLEUR NORMALE
C
          IF (OKWAIT) THEN
            IF (.NOT.OKIF) THEN
              CALL ERREUR(NOWAIT)
            ELSE
              CALL GENSYN
            ENDIF
          ENDIF
          IF (LIGNE(1:1).EQ.'*') THEN
            WRITE (LUOUT,FMT='(A80)',ERR=10000)LIGNE
            WRITE (LULST,FMT='(I6,5X,A80)',ERR=10000)NLIGN,LIGNE
          ELSE
            CALL CADRE(LIGNE)
          ENDIF
          OKWAIT=.FALSE.
          OKIF=.FALSE.
        ENDIF
      GOTO 10

10000 PRINT *,' Erreur de traitement de fichier '
9999  IF (INDEX(LIGNE,'END').EQ.0) CALL ERREUR(NOEND)
      CLOSE (LUIN)
      IF (OKERR) THEN
        CLOSE (LUOUT,STATUS='KEEP')
        WRITE (LUCON,FMT=100)
        WRITE (LULST,FMT=100)
100     FORMAT (/,' Pas d''erreurs detectees.')
      ELSE
        CLOSE (LUOUT,STATUS='DELETE')
        WRITE (LUCON,FMT=200)
        WRITE (LULST,FMT=200)
200     FORMAT (/,' Erreur(s) detectee(s) !!')
      ENDIF
      CLOSE (LULST)
      STOP
      END

      BLOCK DATA
```

```
      IMPLICIT INTEGER*2 (A-Z)
      CHARACTER*80 LIGNE
      LOGICAL OKWAIT,OKIF,OKERR
      COMMON /LUFIC/LUIN,LUOUT,LUCON,LULST
      COMMON /INFO/LIGNE,NLIGN,NGEN,OKWAIT,OKIF,OKERR
      DATA NOWAIT,NOEND/-13,14/
C
C     EXPLORATION DU FICHIER SOURCE ET TRAITEMENT DE LA LIGNE LUE
C
      OKWAIT=.FALSE.
      OKIF=.FALSE.
      OKERR=.TRUE.
10    CONTINUE
        READ (LUIN,FMT='(A80)',END=9999,ERR=10000)LIGNE
        NLIGN=NLIGN+1
        IF (LIGNE(1:2).EQ.'*$') THEN
C
C         LIGNE DE SYNCHRO A ANALYSER
C
          CALL TRI (LIGNE(3:))
        ELSE
C
C       LIGNE D'ASSEMBLEUR NORMALE
C
          IF (OKWAIT) THEN
            IF (.NOT.OKIF) THEN
              CALL ERREUR(NOWAIT)
            ELSE
              CALL GENSYN
            ENDIF
          ENDIF
          IF (LIGNE(1:1).EQ.'*') THEN
            WRITE (LUOUT,FMT='(A80)',ERR=10000)LIGNE
            WRITE (LULST,FMT='(I6,5X,A80)',ERR=10000)NLIGN,LIGNE
          ELSE
            CALL CADRE(LIGNE)
          ENDIF
          OKWAIT=.FALSE.
          OKIF=.FALSE.
        ENDIF
      GOTO 10

10000 PRINT *,' Erreur de traitement de fichier '
9999  IF (INDEX(LIGNE,'END').EQ.0) CALL ERREUR(NOEND)
      CLOSE (LUIN)
      IF (OKERR) THEN
        CLOSE (LUOUT,STATUS='KEEP')
        WRITE (LUCON,FMT=100)
        WRITE (LULST,FMT=100)
100     FORMAT (/,' Pas d''erreurs detectees.')
      ELSE
        CLOSE (LUOUT,STATUS='DELETE')
        WRITE (LUCON,FMT=200)
        WRITE (LULST,FMT=200)
200     FORMAT (/,' Erreur(s) detectee(s) !!')
      ENDIF
      CLOSE (LULST)
      STOP
```

```
      CHARACTER*(*) TAB,ELT
      DIMENSION TAB(NBELT)

      DEBUT=1
      FIN=NBELT
10    CONTINUE
        IF (DEBUT.GT.FIN) THEN
          POS=0
          RETURN
        ENDIF
        MILIEU=(DEBUT+FIN)/2
        IF (ELT.EQ.TAB(MILIEU)) THEN
          POS=MILIEU
          RETURN
        ENDIF
        IF (LLE(ELT,TAB(MILIEU))) THEN
          FIN=MILIEU-1
        ELSE
          DEBUT=MILIEU+1
        ENDIF
      GOTO 10
      END

      SUBROUTINE CADRE
C
C IMPRESSION D'UNE LIGNE EN FORMAT JOLI
C
      IMPLICIT INTEGER*2 (A-Z)
      CHARACTER*80 LIGNE,ETIQ,CODEOP,OPERAND,COMMENT
      CHARACTER*20 FORM
      LOGICAL OKWAIT,OKIF,OKERR
      COMMON /INFO/LIGNE,NLIGN,NGEN,OKWAIT,OKIF,OKERR
      COMMON /LUFIC/LUIN,LUOUT,LUCON,LULST

      POS=1
      CALL PARSE(LIGNE,POS,ETIQ,BIDON)
      CALL PARSE(LIGNE,POS,CODEOP,BIDON)
      CALL PARSE(LIGNE,POS,OPERAND,LONG)
      COMMENT=LIGNE(POS:80)
      LONG=INDEX(OPERAND,' ')-1
      IF (LONG.LT.18) LONG=18
      WRITE(LUOUT,FMT=100)ETIQ,CODEOP,OPERAND(1:LONG),COMMENT
100   FORMAT(A8,X,A8,X,A,X,A)
      WRITE(LULST,FMT=200)NLIGN,ETIQ,CODEOP,OPERAND(1:LONG),COMMENT
200   FORMAT(I6,5X,A8,X,A8,X,A,X,A)
      RETURN
      END

      SUBROUTINE PARSE(LIGNE,POS,ZONE,LONG)
C
C DECOMPOSITION D'UNE LIGNE EN ZONES
C
      IMPLICIT INTEGER*2 (A-Z)
      CHARACTER*80 LIGNE,ZONE,CAR*1
      LOGICAL*1 KOT
      DATA KOT/.TRUE./,LL/80/
      CHARACTER*80 BLANC
      DATA (BLANC(I:I),I=1,80)/80*' '/

      ZONE=BLANC
```

```
          CAR=LIGNE(POS:POS)
          IF ((CAR.EQ.' ').AND.KOT) GOTO 100
          ZONE(LONG:LONG)=CAR
          IF (CAR.EQ.'''') KOT=.NOT.KOT
          POS=POS+1
          LONG=LONG+1
        GOTO 10
100     CONTINUE
          POS=POS+1
          IF (POS.GT.LL) RETURN
          CAR=LIGNE(POS:POS)
        IF (CAR.EQ.' ') GOTO 100
        END

        SUBROUTINE TRI(EXPRES)
C
C TRAITEMENT DES ORDRES DE SYNCHRO: $WAIT ET $IF
C
        IMPLICIT INTEGER*2 (A-Z)
        CHARACTER*80 EXPRES,COMMAND,ZONARG,LIGNE,BLANC,CHBID
        CHARACTER*8 ARGUMT(32),CNGEN,CNARG,TEMP,STR,POPDE(32),OPDE
        CHARACTER TOPTR(4),CAR,OPTR,TBNODT(32)*8
        DIMENSION TABPRI(4),POPTR(32),PTRARG(32),TABVAR(100,2)
        DIMENSION TABNOD(100,2)
        LOGICAL OKWAIT,EVALTT,OKIF,OKERR
        COMMON /LUFIC/LUIN,LUOUT,LUCON,LULST
        COMMON /INFO/LIGNE,NLIGN,NGEN,OKWAIT,OKIF,OKERR
        COMMON /AS/TOPTR,POPTR,POPDE,PPOPTR,PPOPDE,PRIOPR,NBOPTR,NIVPAR,
       +TABPRI
        COMMON /GENCOD/ARGUMT,PTRARG,TABVAR,POSLIB,TABNOD,NUMNOD,NARG,
       +TBNODT
        DATA LL,NBMAX/78,32/,CONEPA,NOARG,MAXARG,DBLARG,SYNERR/1,-2,-3,-4
       +,5/
        DATA ARGINC,PAROUV,PARFER,NOOPDE,NOOPTR/-6,-7,-8,-9,-10/
        DATA NOTHEN,DBWAIT,NOWAIT,NOETIQ,DBETIQ/-11,-12,-13,-16,17/
        DATA (BLANC(I:I),I=1,80)/80*' '/

        POS=1
        CALL PARSE(EXPRES,POS,COMMAND,LONG)
C
C       TRAITEMENT DE L'ORDRE $WAIT
C
        IF (COMMAND(1:LONG).EQ.'WAIT') THEN
          IF (OKWAIT) THEN
            IF (OKIF) THEN
              CALL GENSYN
            ELSE
              WRITE(LULST,FMT='(I6,5X,A80)')NLIGN,LIGNE
              WRITE(LUOUT,FMT='(''**'',A80)')LIGNE
              CALL ERREUR(DBWAIT)
              RETURN
            ENDIF
          ENDIF
          WRITE(LUOUT,FMT='(''**'',A80)')LIGNE
          WRITE(LULST,FMT='(I6,5X,A80)')NLIGN,LIGNE
          NBNODT=1
          NUMNOD=0
          POSLIB=1
          DO 1,I=1,NBMAX
```

```
      OKIF=.FALSE.
      NARG=0
C
C     RECHERCHE DES ARGUMENTS
C
      IF (POS.GT.LL-2) THEN
        CALL ERREUR(NOARG)
        RETURN
      ENDIF
      ZONARG=EXPRES(POS:)
5     POS=INDEX(ZONARG,',')
      IF (POS.NE.0) THEN
        NARG=NARG+1
        IF (NARG.GT.(NBMAX-1)) THEN
          CALL ERREUR(MAXARG)
          RETURN
        ENDIF
        ARGUMT(NARG)=ZONARG(1:POS-1)
        ZONARG=ZONARG(POS+1:)
        GOTO 5
      ENDIF
      IF (ZONARG(1:1).NE.' ') THEN
        NARG=NARG+1
        ARGUMT(NARG)=ZONARG
      ELSE
        CALL ERREUR(SYNERR)
      ENDIF
C
C     TRI DES ARGUMENTS PAR ORDRE ALPHABETIQUE
C
      ECART=NARG
10    CONTINUE
        ECART=ECART/2
        IF (ECART.GE.1) THEN
          DO 20,J=1,NARG-ECART
            DO 30,I=J,1,-ECART
              M=I+ECART
              IF (LLE(ARGUMT(I),ARGUMT(M))) GOTO 25
              TEMP=ARGUMT(M)
              ARGUMT(M)=ARGUMT(I)
              ARGUMT(I)=TEMP
30          CONTINUE
25          IF (ARGUMT(I).EQ.ARGUMT(M)) THEN
              CALL ERREUR(DBLARG)
              RETURN
            ENDIF
20        CONTINUE
        GOTO 10
      ENDIF
      CNARG=STR(NARG)
      NGEN=NGEN+1
      CNGEN=STR(NGEN)
      WRITE(LUOUT,FMT=200)CNARG(1:INDEX(CNARG,' ')-1),CNGEN
200   FORMAT(9X,'WAIT&',T19,A,',',A)
      CNGEN=STR(NGEN)
      WRITE(LUOUT,FMT=300)(ARGUMT(I)(1:INDEX(ARGUMT(I),' ')-1),
     +  CNGEN,I=1,NARG)
300   FORMAT(9X,'TAB&',T19,A,',',A)
C
```

```
            OPDRAP=0
            WRITE(LUOUT,FMT='(''**'',A80)')LIGNE
            WRITE(LULST,FMT='(I6,5X,A80)')NLIGN,LIGNE
            IF (.NOT.OKWAIT) THEN
              CALL ERREUR(NOWAIT)
              RETURN
            ENDIF
            OKIF=.TRUE.
            PPOPDE=0
            PPOPTR=0
            PRIOPR=0
            NIVPAR=0
            POSFIN=INDEX(EXPRES,'THEN')
            IF (POSFIN.EQ.0) THEN
              CALL ERREUR(NOTHEN)
              RETURN
            ENDIF
            ZONARG=EXPRES(POS:POSFIN-1)
 60         POS=80
            DO 40,I=1,NBOPTR
              POS1=INDEX(ZONARG,TOPTR(I))
              IF (POS1.NE.0) POS=MIN(POS1,POS)
 40         CONTINUE
 C
 C          TRAITEMENT D'UN OPERATEUR
 C
            IF (POS.EQ.1) THEN
              OPTR=ZONARG(1:1)
              ZONARG=ZONARG(2:)
              DO 50,I=1,NBOPTR
                IF (TOPTR(I).EQ.OPTR) NUMOPE=I
 50           CONTINUE
              IF (OPTR.EQ.'(') THEN
 C
 C              OPERATEUR = (
 C
                NIVPAR=NIVPAR+1
              ELSE IF (OPTR.EQ.')') THEN
 C
 C              OPERATEUR = )
 C
                NIVPAR=NIVPAR-1
                IF (NIVPAR.LT.0) THEN
                  CALL ERREUR(PAROUV)
                  RETURN
                ENDIF
                IF (EVALTT()) RETURN
                PPOPTR=PPOPTR-1
                IF (PPOPTR.EQ.0) THEN
                  PRIOPR=0
                ELSE
                  PRIOPR=TABPRI(POPTR(PPOPTR))
                ENDIF
              ELSE
 C
 C            TOUT AUTRE OPERATEUR:
 C            SI L'OPERATEUR EST MOINS PRIORITAIRE QUE LE PRECEDENT ON EVALUE
 C            PUIS ON EMPILE LE DERNIER OPERATEUR
```

```
            ENDIF
            IF (OPTR.NE.')') THEN
              PPOPTR=PPOPTR+1
              POPTR(PPOPTR)=NUMOPE
              PRIOPR=TABPRI(NUMOPE)
            ENDIF
C
C         TRAITEMENT D'UN OPERANDE
C
          ELSE IF (POS.NE.80) THEN
            OPDRAP=OPDRAP+1
            POS1=1
            CALL PARSE(ZONARG,POS1,CHBID,LONG)
            IF (POS1.LT.POS) THEN
              CALL ERREUR(NOOPTR)
              RETURN
            ENDIF
            OPDE=ZONARG(1:POS-1)
            ZONARG=ZONARG(POS:)
            CALL DICOTO(OPDE,ARGUMT,NARG,NUMOPD)
            IF (NUMOPD.EQ.0) THEN
              CALL ERREUR(ARGINC)
              RETURN
            ENDIF
            PPOPDE=PPOPDE+1
            POPDE(PPOPDE)=OPDE
          ELSE
C
C           IL RESTE OU UN OPERANDE OU RIEN DU TOUT
C
            OPDE=ZONARG
C
C             IL RESTE UN OPERANDE DANS LA LIGNE
C
            IF (OPDE.NE.'          ') THEN
              OPDRAP=OPDRAP+1
              POS1=1
              CALL PARSE(OPDE,POS1,CHBID,LONG)
              IF (POS1.LT.LEN(OPDE)) THEN
                CALL ERREUR(NOOPTR)
                RETURN
              ENDIF
              CALL DICOTO(OPDE,ARGUMT,NARG,NUMOPD)
              IF (NUMOPD.EQ.0) THEN
                CALL ERREUR(ARGINC)
                RETURN
              ENDIF
              PPOPDE=PPOPDE+1
              POPDE(PPOPDE)=OPDE
            ENDIF
C
C           LA LIGNE EST TERMINEE, ON FINIT L'EVALUATION
C
            IF (NIVPAR) 70,80,90
80          IF (EVALTT()) RETURN
C
C           ON MARQUE QUE LA RACINE DE L'ARBRE EST UN NOEUD TERMINAL
C
            IF (OPDRAP.EQ.1) THEN
```

```
                PTRARG(NUMOPD)=POSLIB
              ELSE
                PTR=PTRARG(NUMOPD)
  100           CONTINUE
                PTR1=TABVAR(PTR,2)
                IF (PTR1.NE.0) THEN
                  PTR=PTR1
                  GOTO 100
                ENDIF
                TABVAR(PTR,2)=POSLIB
              ENDIF
              TABVAR(POSLIB,1)=NUMNOD
              TABVAR(POSLIB,2)=0
              POSLIB=POSLIB+1
            ELSE
              TABNOD(NUMNOD,2)=0
            ENDIF
            CALL PARSE(EXPRES,POSFIN,CHBID,LONG)
            IF (POSFIN.GT.LL) THEN
              CALL ERREUR(NOETIQ)
            ELSE
              CALL PARSE(EXPRES,POSFIN,CHBID,LONG)
              TBNODT(NBNODT)=CHBID(1:LONG)
              IF (POSFIN.LE.LL) CALL ERREUR (DBETIQ)
              NBNODT=NBNODT+1
            ENDIF
            RETURN
  90        CALL ERREUR(PARFER)
            RETURN
  70        CALL ERREUR(PAROUV)
            RETURN
          ENDIF
          GOTO 60
C
C     TRAITEMENT D'UN ORDRE INCONNU AU BATAILLON
C
        ELSE
          CALL ERREUR(CONEPA)
        ENDIF
        RETURN
        END

        LOGICAL FUNCTION EVALTT()
C
C EVALUATION DE L'EXPRESSION EMPILEE JUSQU'A DETECTION D'UNE (
C
        IMPLICIT INTEGER*2 (A-Z)
        CHARACTER*8 TOPTR(4)*1,POPDE(32),OPDE1,OPDE2
        DIMENSION TABPRI(4),POPTR(32)
        COMMON /AS/TOPTR,POPTR,POPDE,PPOPTR,PPOPDE,PRIOPR,NBOPTR,NIVPAR,
       +TABPRI
        DATA NOOPDE/-9/

        EVALTT=.FALSE.
  10    CONTINUE
C
C       SI LA PILE OPERATEUR EST VIDE, ON ARRETE L'EVALUATION
C
        IF (PPOPTR.NE.0) THEN
```

```
C
          IF (TABPRI(NUMOPE).NE.0) THEN
            PPOPTR=PPOPTR-1
C
C             DEPILAGE DU 2EME OPERANDE
C
              IF (PPOPDE.EQ.0) THEN
                CALL ERREUR(NOOPDE)
                EVALTT=.TRUE.
                RETURN
              ENDIF
              OPDE2=POPDE(PPOPDE)
              PPOPDE=PPOPDE-1
C
C             DEPILAGE DU 1ER OPERANDE
C
              IF (PPOPDE.EQ.0) THEN
                CALL ERREUR(NOOPDE)
                EVALTT=.TRUE.
                RETURN
              ENDIF
              OPDE1=POPDE(PPOPDE)
              PPOPDE=PPOPDE-1
C
C             EVALUATION DU TRIPLET OPERANDE1, OPERATEUR, OPERANDE2
C
              CALL EVAL(NUMOPE,OPDE1,OPDE2)
            GOTO 10
          ENDIF
        ENDIF
        PRIOPR=0
        RETURN
        END

        SUBROUTINE EVAL(NUMOPE,OPDE1,OPDE2)
C
C S/P D'EVALUATION DU TRIPLET OPERATEUR, OPERANDE1, OPERANDE2
C
        IMPLICIT INTEGER*2 (A-Z)
        CHARACTER*8 TOPTR(4)*1,POPDE(32),OPDE1,OPDE2,ARGUMT(32),STR
        DIMENSION TABPRI(4),POPTR(32),PTRARG(32),TABVAR(100,2)
        DIMENSION TABNOD(100,2)
        COMMON /AS/TOPTR,POPTR,POPDE,PPOPTR,PPOPDE,PRIOPR,NBOPTR,NIVPAR,
       +TABPRI
        COMMON /LUFIC/LUIN,LUOUT,LUCON,LULST
        COMMON /GENCOD/ARGUMT,PTRARG,TABVAR,POSLIB,TABNOD,NUMNOD,NARG,
       +TBNODT

        NUMNOD=NUMNOD+1
C
C       ON EMPILE UN NOEUD INTERMEDIAIRE DANS LA PILE OPERANDE
C
        PPOPDE=PPOPDE+1
        POPDE(PPOPDE)='@NOD.'//STR(NUMNOD)
        TABNOD(NUMNOD,1)=NUMOPE
C
C       TRAITEMENT DU 1ER OPERANDE
C
```

```
          CALL DICOTO(OPDE1,ARGUMT,NARG,POS)
          IF (PTRARG(POS).EQ.0) THEN
            PTRARG(POS)=POSLIB
          ELSE
            POS=PTRARG(POS)
 10         CONTINUE
              POS1=TABVAR(POS,2)
            IF (POS1.NE.0) THEN
              POS=POS1
              GOTO 10
            ENDIF
            TABVAR(POS,2)=POSLIB
          ENDIF
          TABVAR(POSLIB,1)=NUMNOD
          TABVAR(POSLIB,2)=0
          POSLIB=POSLIB+1
        ELSE
C
C         L'OPERANDE EST UN NOEUD DE L'ARBRE
C
          NODPRE=INVSTR(OPDE1(6:))
          TABNOD(NODPRE,2)=NUMNOD
        ENDIF
C
C       TRAITEMENT DU 2EME OPERANDE
C
        IF (OPDE2(1:5).NE.'@NOD.') THEN
C
C         L'OPERANDE EST UNE FEUILLE DE L'ARBRE
C
          CALL DICOTO(OPDE2,ARGUMT,NARG,POS)
          IF (PTRARG(POS).EQ.0) THEN
            PTRARG(POS)=POSLIB
          ELSE
            POS=PTRARG(POS)
 20         CONTINUE
              POS1=TABVAR(POS,2)
            IF (POS1.NE.0) THEN
              POS=POS1
              GOTO 20
            ENDIF
            TABVAR(POS,2)=POSLIB
          ENDIF
          TABVAR(POSLIB,1)=NUMNOD
          TABVAR(POSLIB,2)=0
          POSLIB=POSLIB+1
        ELSE
C
C       L'OPERANDE EST UN NOEUD DE L'ARBRE
C
          NODPRE=INVSTR(OPDE2(6:))
          TABNOD(NODPRE,2)=NUMNOD
        ENDIF
        RETURN
        END

        SUBROUTINE GENSYN
C
```

```
      CHARACTER*8 LIGNE*80,ETIQ,ZONOP,CNGEN,CNNOD,C1*1,C2*2,ARGUMT(32)
      CHARACTER*8 CODEOP,TBNODT(32),STR
      DIMENSION PTRARG(32),TABVAR(100,2),TABNOD(100,2)
      LOGICAL OKIF,OKWAIT,OKERR,UNEFOI
      COMMON /GENCOD/ARGUMT,PTRARG,TABVAR,POSLIB,TABNOD,NUMNOD,NARG,
     +TBNODT
      COMMON /INFO/LIGNE,NLIGN,NGEN,OKWAIT,OKIF,OKERR
      COMMON /LUFIC/LUIN,LUOUT,LUCON,LULST
      DATA VNOUTI/15/

      CNGEN=STR(NGEN)
C
C     REMISE A ZERO DES EVENEMENTS
C
      WRITE(LUOUT,FMT=400)CNGEN
400   FORMAT('@CLR.',A3,X,'EQU',T19,'*')
      DO 5,I=1,NUMNOD
        IF (TABNOD(I,2).GE.0) THEN
          CNNOD=STR(I)
          WRITE(LUOUT,FMT=500)CNGEN(1:INDEX(CNGEN,' ')-1),CNNOD(1:INDEX
     +    (CNNOD,' ')-1)
500       FORMAT(T10,'CLR.W',T19,'@N',A,'.',A,'-2')
        ENDIF
5     CONTINUE
      WRITE(LUOUT,FMT=200)'RTS'
C
C     GENERATION DU CODE CORRESPONDANT AUX FEUILLES DE L'ARBRE
C
      DO 10,I=1,NARG
        UNEFOI=.TRUE.
20      CONTINUE
          IF (UNEFOI) THEN
            POS=PTRARG(I)
            IF (POS.EQ.0) THEN
              CALL ERREUR(VNOUTI)
              GOTO 10
            ENDIF
            UNEFOI=.FALSE.
            ETIQ='@'//ARGUMT(I)(1:INDEX(ARGUMT(I),' ')-1)//'.'//CNGEN
            CNNOD=STR(TABVAR(POS,1))
            ZONOP='@N'//CNGEN(1:INDEX(CNGEN,' ')-1)//'.'//CNNOD
          ELSE
            ETIQ='        '
            CNNOD=STR(TABVAR(POS,1))
            ZONOP='@N'//CNGEN(1:INDEX(CNGEN,' ')-1)//'.'//CNNOD
          ENDIF
          POS=TABVAR(POS,2)
          WRITE(LUOUT,FMT=100)ETIQ,'BSR',ZONOP
100       FORMAT(A8,X,A3,T19,A)
        IF (POS.NE.0) GOTO 20
        WRITE(LUOUT,FMT=200)'RTS'
200     FORMAT(T10,A)
10    CONTINUE
C
C     GENERATION DU CODE CORRESPONDANT A UN NOEUD DE L'ARBRE
C
      NBNODT=1
      DO 30,I=1,NUMNOD
        CNNOD=STR(I)
```

```
110         C2='NO'
            C1='T'
            ZONOP=TBNODT(NBNODT)
            NBNODT=NBNODT+1
            GOTO 150
120       CONTINUE
            C1='T'
            ZONOP=TBNODT(NBNODT)
            NBNODT=NBNODT+1
            GOTO 140
130       CONTINUE
            C1='I'
            CNNOD=STR(NODSUI)
            ZONOP='@N'//CNGEN(1:INDEX(CNGEN,' ')-1)//'.'//CNNOD
140       CONTINUE
            IF (TABNOD(I,1).EQ.3) THEN
              C2='OU'
            ELSE
              C2='ET'
            ENDIF
150       CONTINUE
          CODEOP=C2//'.N'//C1//'&'
          WRITE(LUOUT,FMT=300)ETIQ,CODEOP,ZONOP
300       FORMAT(A8,X,A8,X,A)
30      CONTINUE
        RETURN
        END

        SUBROUTINE ERREUR(ERRNO)
C
C SOUS-PROGRAMME D'AFFICHAGE DES MESSAGES D'ERREUR
C
        IMPLICIT INTEGER*2 (A-Z)
        LOGICAL OKWAIT,OKIF,OKERR
        COMMON /LUFIC/LUIN,LUOUT,LUCON,LULST
        CHARACTER*80 LIGNE,MESS(17),TYPE*15
        COMMON /INFO/LIGNE,NLIGN,NGEN,OKWAIT,OKIF,OKERR
        DATA (MESS(I),I=1,17)/'Ordre inconnu','Pas d''argument',
     +'Trop d''arguments','Repetition d''un argument',
     *'Pas d''argument apres la virgule','Argument non declare',
     -'( Manquante',') Manquante','Operande manquant',
     ='Operateur manquant','THEN Manquant','WAIT sans IF apres',
     1'IF sans WAIT avant','Pas de END en fin de fichier',
     2'Variable declaree en WAIT non utilisee',
     3'Pas d''etiquette apres THEN','Plus d''une etiquette apres THEN'/

        IF (ERRNO.LT.0) THEN
          TYPE='Erreur         :'
          OKERR=.FALSE.
        ELSE
          TYPE='Avertissement:'
        ENDIF
        WRITE(LULST,FMT=100)TYPE,MESS(ABS(ERRNO))
100     FORMAT('******',5X,A15,X,A)
        WRITE(LUCON,FMT=200)TYPE,NLIGN,MESS(ABS(ERRNO))
200     FORMAT('* ',A15,' Ligne ',I4,': ',A)
        RETURN
        END
```

Exemple de fichier d'entrée

```
          ENTRY    TEST1
          EXTRN.W  NF01
          EXTRN.W  NF02
          EXTRN.W  NF03

A         DS.L     1
B         DS.L     1
C         DS.L     1

TEST1     MOVE.L   #$80808080,AIT.SY+12
          LFO@     #NF01,A
          LFO@     #NF02,B
          LFO@     #NF03,C
BOU       EQU      *
*$WAIT A,B,C
*$IF A.B THEN AL1
*$IF B.C THEN AL2
*$IF C.A THEN AL3
AL1       BCHG.B   #7,AIT.SY+15
          RENBLOC@ A
          RENBLOC@ B
          LFO@     #NF01,A
          LFO@     #NF02,B
          BRA      BOU
AL2       BCHG.B   #7,AIT.SY+14
          RENBLOC@ B
          RENBLOC@ C
          LFO@     #NF02,B
          LFO@     #NF03,C
          BRA      BOU
AL3       BCHG.B   #7,AIT.SY+13
          RENBLOC@ C
          RENBLOC@ A
          LFO@     #NF03,C
          LFO@     #NF01,A
          BRA      BOU
          END
```

Résultat de la compilation

```
            EXTRN     AIT.SY
            ENTRY     TEST1
            EXTRN.W   NFO1
            EXTRN.W   NFO2
            EXTRN.W   NFO3

A           DS.L      1
B           DS.L      1
C           DS.L      1

TEST1       MOVE.L    #%80808080,AIT.SY+12
            LFO@      #NFO1,A
            LFO@      #NFO2,B
            LFO@      #NFO3,C
BOU         EQU       *
****$WAIT   A,B,C
            WAIT&     3,1
            TAB&      A,1
            TAB&      B,1
            TAB&      C,1
****$IF A.B THEN AL1
****$IF B.C THEN AL2
****$IF C.A THEN AL3
@CLR.1      EQU       *
            CLR.W     @N1.1-2
            CLR.W     @N1.2-2
            CLR.W     @N1.3-2
            RTS
@A.1        BSR       @N1.1
            BSR       @N1.3
            RTS
@B.1        BSR       @N1.1
            BSR       @N1.2
            RTS
@C.1        BSR       @N1.2
            BSR       @N1.3
            RTS
@N1.1       ET.NT&    AL1
@N1.2       ET.NT&    AL2
@N1.3       ET.NT&    AL3
AL1         BCHG.B    #7,AIT.SY+15
            RENBLOC@  A
            RENBLOC@  B
            LFO@      #NFO1,A
            LFO@      #NFO2,B
            BRA       BOU
AL2         BCHG.B    #7,AIT.SY+14
            RENBLOC@  B
            RENBLOC@  C
            LFO@      #NFO2,B
            LFO@      #NFO3,C
            BRA       BOU
AL3         BCHG.B    #7,AIT.SY+13
            RENBLOC@  C
```

ANNEXE B
TACHE GÉNÉRALISÉE

Listing de la tâche généralisée

```
*************************************************************************
*                                                                       *
*                NOUVELLE TACHE GENERALISEE CESAR                       *
*                                                                       *
*                                                                       *
*                                                                       *
*************************************************************************


*  DEFINITION DE LA TABLE TACHE

CRIN      EQU     0                 TABLE DES COMPTE-RENDUS
CRGO      EQU     CRIN+1
CRFN      EQU     CRIN+2
CRSUSP    EQU     CRIN+3
CRCONT    EQU     CRIN+4
CRAF      EQU     CRIN+5
CRFP      EQU     CRIN+6
@IN       EQU     CRIN+8            TABLE DES ADRESSES DES ROUTINES
@GO       EQU     @IN+4
@FN       EQU     @IN+3
@SUSP     EQU     @IN+12
@CONT     EQU     @IN+16
@AF       EQU     @IN+20
@FP       EQU     @IN+24
@DEB      EQU     @IN+28
FNDRAP    EQU     @IN+32
NBSUSP    EQU     FNDRAP+1
FOE       EQU     NBSUSP+1
REQFN     EQU     FOE+2
@TABRE    EQU     REQFN+4           @ DE LA TABLE DES RESSOURCES
@TABDON   EQU     @TABRE+4
@TABFOS   EQU     @TABDON+4
@REQ      EQU     @TABFOS+4
          ENTRY   CONEPA
          EXTRN.W NRES.256

*-----------------------------------------------------------------------

* INITIALISATION DE L'ENVIRONNEMENT TACHE

CODSYNC   PROG
          PDTASK  0,A6              CHARGE @ TABLE
          MOVE.L  #PFN,@FN(A6)      CHARGE ADRESSE TRAITEMENT ORDRE FN?
          LEA     PS1,A4            CHARGE @ DE RETOUR
          TST.B   CRIN(A6)          ACCEPTE INIT ?
          BMI     PINIT1            NON: IL FAUT FAIRE L'INIT
          MOVE.L  #PINIT,@IN(A6)    ACCEPTE INIT

*-----------------------------------------------------------------------

* TRI DES ORDRES ET AIGUILLAGE VERS LA ROUTINE DE TRAITEMENT
PS1       JSR     LECFO             LECTURE SUR LA F.O. D'ENTREE
PS1.1     WAIT@   @REQ(A6),A5           ATTENTE EVT F.O.
          MOVE.L  16(A5),D7         RECUPERE L'ORDRE
          MOVE.L  @IN(A6,D7.W),A2   TRAITEMENT ORDRE
          JMP     (A2)

*-----------------------------------------------------------------------
```

```
* REPONSE A UN ORDRE NON RECONNU OU MAL A PROPOS

CONEPA   ASR.W    #2,D7              DIVISE PAR 4
         MOVE.B   CRIN(A6,D7.W),D0   RECUPERE LE COMPTE RENDU
         ECRICR@  @REQ(A6)               ENVOI DU C.R.
         CMP.L    #PS1,A4            POINT DE RETOUR = PS1
         BEQ      CONEP1             OUI: PAS DE LECTURE F.O.
         JSR      LECFO              LECTURE F.O.
CONEP1   JMP      (A4)               RETOUR A L'APPELANT

*-------------------------------------------------------------------------

* PROCEDURE DE TRAITEMENT DE L'ORDRE INIT

PINIT    MOVE.L   20(A5),@TABRE(A6)  INIT TABLE DES RESSOURCES
PINIT1   MOVE.L   @TABRE(A6),D7      D7 POINTE SUR LA TABLE RESSOURCES
         BEQ      PINIT2             SI NUL PAS DE RESSOURCES
         MOVE.L   D7,A1              A1 POINTE SUR LA TABLE RESSOURCES
         TST.B    (A1)+              EST CE UNE RESSOURCE DE CADENCEMENT ?
         BPL      PINIT2             NON: PAS DE CONNECTION
         CLR.W    D7
         MOVE.B   (A1)+,D7           NOMBRE DE RESSOURCE CADENCEMENT
         BEQ      PINIT2             SI NUL PAS DE CONNECTION
         SUB.W    #1,D7              PREPARATION POUR DBRA
PINIT3   MOVE.L   (A1)+,A2           REQUETE DE CONNECTION
         TRAP     4                  ENVOI REQUETE
         DBRA     D7,PINIT3          C'EST REPARTI POUR UN TOUR
PINIT2   TST.B    CRAF(A6)           ACCEPTE L'A.F.?
         BMI      PINIT4             NON: ON NE CHANGE PAS LA TABLE
         MOVE.L   #PAF,@AF(A6)       CHARGEMENT @ TRAITEMENT A.F.
PINIT4   MOVE.L   #CONEPA,@IN(A6)    ON ACCEPTE PLUS L'INIT
         MOVE.B   CRIN(A6),D0        C.R. D'INIT
         BMI      PINIT5             PAS DE C.R. SI INIT NON ACCEPTE
         ECRICR@  @REQ(A6)               ENVOI DU C.R.
PINIT5   TST.B    CRGO(A6)           ACCEPTE LE GO?
         BMI      PGO                NON: ON EFFECTUE LA PROCEDURE DE GO
         MOVE.L   #PGO,@GO(A6)       CHARGE @ TRAITEMENT DE GO
         JMP      (A4)               RETOUR A L'APPELANT

*-------------------------------------------------------------------------

* PROCEDURE DE TRAITEMENT DE L'ORDRE FN?

PFN      CMPA.L   #PS1,A4            EST'ON ACTIVABLE ?
         BEQ      OKFN               NON: ON ENVOIE LE C.R.
         ST       FNDRAP(A6)         MARQUE ARRIVEE DE FN?
         MOVE.L   @REQ(A6),REQFN(A6)     SAUVE LA REQUETE FN?
         JSR      LECFO              LECTURE F.O
         JMP      (A4)               RETOUR A L'APPELANT
OKFN     MOVE.B   CRFN(A6),D0        CHARGE LE C.R.
         ECRICR@  @REQ(A6)               ENVOI DU C.R.
         JMP      (A4)               RETOUR A L'APPELANT

*-------------------------------------------------------------------------

* PROCEDURE DE TRAITEMENT DE L'ORDRE SUSP

PSUSP    TST.B    NBSUSP(A6)         A T'ON RECU DES SUSPs ?
         BNE      PSUSP1             OUI: PAS DE CHANGEMENT
         MOVE.L   A4,D6              SAUVE @ DE RETOUR
         LEA      PS1,A4             PS1 EST LA Nelle @ DE RETOUR
         MOVE.L   #PCONT,@CONT(A6)   ON ACCEPTE LES CONT
PSUSP1   ADD.B    #1,NBSUSP(A6)      INCREMENTE COMPTEUR DE SUSP
         MOVE.L   @TABFOS(A6),D1     ADRESSE TABLE F.O. SORTIE
         BEQ      PSUSP2             SI NUL SAUT
         JSR      PROPAG             PROPAGATION DE L'ORDRE
```

```
PSUSP1   MOVE.B  CRSUSP(A6),D0
         ECRICR@ @REQ(A6)              ENVOI DU C.R.
         BRA     PS1

*-------------------------------------------------------------------

* PROCEDURE DE TRAITEMENT DE L'ORDRE CONT

PCONT    MOVE.L  @TABFOS(A6),D1   ADRESSE TABLE F.O. SORTIE
         BEQ     PCONT1           SI NUL SAUT
         JSR     PROPAG           PROPAGATION DE L'ORDRE
PCONT1   MOVE.B  CRCONT(A6),D0    CHARGE LE C.R.
         ECRICR@ @REQ(A6)             ENVOI DU C.R.
         SUB.B   #1,NBSUSP(A6)    DECREMENTE LE COMPTEUR DE SUSP
         BNE     PS1              S'IL RESTE DES SUSP, CA SUFFIT PAS
         JSR     LECFO            LECTURE F.O.
         MOVE.L  #CONEPA,@CONT(A6)  ON REFUSE LES CONT
         MOVE.L  D6,A4            RESTAURE L'@ DE RETOUR ...
*                                   ... SUBTILEMENT SAUVEGARDEE PLUS HAUT
         JMP     (A4)             ET ON Y VA ....

*-------------------------------------------------------------------

* PROCEDURE DE TRAITEMENT DE L'ORDRE AF

PAF      EQU     *
         MOVE.L  @TABRE(A6),D4    CHARGE L'@ DE LA TABLES DES RESS.
         BEQ     PAF1             SI NUL PAS DE DECONNECTION
         MOVE.L  D4,A0            CHARGE L'@ DE LA TABLE DES RESS.
         TST.B   (A0)+            RESSOURCE DE CADENCEMENT ?
         BPL     PAF1             NON: ON SAUTE
         CLR.W   D4               PREPARATION DBRA
         MOVE.B  (A0)+,D4         NOMBRE DE RESSOURCES
         SUB.W   #1,D4            PREPARATION DBRA
PAF2     MOVE.L  (A0)+,A2         BLOC REQUETE
         MOVE.B  #$C0,9(A2)       1 SEUL EVENEMENT
         MOVE.B  #$38,10(A2)      PAS D'EVT, JETTE LE BLOC
         DBRA    D4,PAF2          ONCE AGAIN ...
         JBUFST  #NRES.256,@TABRE(A6),#256 JETTE LE BUFFER D'INIT
PAF1     MOVE.L  #CONEPA,@AF(A6)
         MOVE.L  #CONEPA,@GO(A6)
         MOVE.L  #PINIT,@IN(A6)   ACCEPTE L'INIT
         MOVE.B  CRAF(A6),D0      CHARGE LE C.R.
         BRA     PCOMM            SAUT VERS LE CODE COMMUN A FP ET AF

*-------------------------------------------------------------------

* PROCEDURE DE TRAITEMENT DE L'ORDRE FP

PFP      EQU     *
         MOVE.L  @TABRE(A6),D4    ADRESSE DE LA TABLE DES RESSOURCES
         BEQ     PFP1             SI NUL PAS DE MASQUAGE DES EVENEMENTS
         MOVE.L  D4,A0            ADRESSE DE LA TABLE DES RESSOURCES
         TST.B   (A0)+            RESSOURCE DE CADENCEMENT ?
         BPL     PFP1             NON: ON SAUTE
         CLR.W   D4               PREPARE DBRA
         MOVE.B  (A0)+,D4         NOMBRE DE RESSOURCES
         SUB.W   #1,D4            PREPARE DBRA
PFP2     MOVE.L  (A0)+,A2         ADRESSE REQUETE
         AND.B   #$BF,10(A2)      PAS DE GENERATION D'EVT
         DBRA    D4,PFP2          C'EST REPARTI POUR UN TOUR
PFP1     MOVE.B  CRFP(A6),D0      CHARGE LE C.R.
         MOVE.L  #PGO,@GO(A6)     ACCEPTE LE GO
PCOMM    EQU     *
* Libere les ressource de donnees
         MOVE.L  #CONEPA,@CONT(A6)
```

```
          MOVE.L   #CONEPA,@CONI(A6)
          MOVE.L   #CONEPA,@SUSP(A6)
          MOVE.L   #CONEPA,@FP(A6)
          CLR.B    NBSUSP(A6)          RAZ DU NOMBRE DE SUSP -
          LEA      PS1,A4              CHARGE @ DE RETOUR
          MOVE.L   @TABFOS(A6),D1      ADRESSE TABLE F.O. SORTIE
          BEQ      PCOMM1              SI NUL SAUT
          JSR      PROPAG              PROPAGATION DE L'ORDRE
PCOMM1    BCLR.B   #7,FNDRAP(A6)       ORDRE FN? ?
          BEQ      PCOMM10             SI NUL SAUT
          RENBLOC@ REQFN(A6)           REND REQ LEC ORDRE FN
PCOMM10   MOVE.L   @TABDON(A6),D1      ADRESSE BUFFER DE DONNEE
          BEQ      PCOMM2              SI NULLE ON SAUTE
          JBUFST   #NRES.256,D1,#256 RENVOI DU BUFFER
PCOMM2    ECRICR@ @REQ(A6)             ENVOI DU C.R.
          BRA      PS1

*-------------------------------------------------------------------------

* ROUTINES DIVERSES

EVTFO     MOVE.L   16(A5),D7               RECUPERE L'ORDRE
          MOVE.L   @IN(A6,D7.W),A2  TRAITEMENT DE L'ORDRE
          JMP      (A2)

LECFO     TRAP     8                   DEMANDE BLOC REQUETE
          MOVE.L   #$1F006800,8(A2) LECTURE FILE D'ORDRE
* SANS ATTENTE,GEN EVEN,CHAINAGE,GARDE LE BLOC
          MOVE.W   FOE(A6),12(A2)      NUMERO RESSOURCE
          TRAP     4                   APPEL SYSTEME
          MOVE.L   A2,@REQ(A6)         STOCKE ADRESSE REQUETE
          RTS

PROPAG    MOVE.L D1,A0
          MOVE.W (A0)+,D3              NBR F.O TOTAL
          SUB.W  #1,D3                 INIT DBRA
PROPAG1   TST.W  (A0)+                 TEST SI FILE LIBRE
          BEQ    PROPAG2               NON: SAUT
          TRAP   8                     DEMANDE BLOC REQUETE
          MOVE.L #$1FFF6800,8(A2)      ECRITURE FILE D'ORDRE
          MOVE.W (A0)+,12(A2)          NUMERO FILE D'ORDRE
          MOVE.B D7,19(A2)             ORDRE A PROPAGER
          MOVE.L A2,(A0)+              STOCKE ADRESSE REQUETE
          TRAP   4                     APPEL SYSTEME
          BRA    PROPAG3
PROPAG2   ADD.L  #6,A0                 RECALAGE DE A0
PROPAG3   DBRA   D3,PROPAG1            O.M.T
*
          MOVE.L @TABFOS(A6),A0
          MOVE.W (A0)+,D3
          SUB.W  #1,D3
PROPAG4   TST.W  (A0)+                 DOIT-ON L'ATTENDRE ?
          BEQ    PROPAG5               NON: SAUT
PROPAG6   MOVE.L 2(A0),A2              ADRESSE REQUETE
          TST.B  11(A2)                REQUETE TRAITEE ?
          BEQ    PROPAG7               OUI: ON SORT
          TRAP   8                     DEMANDE BLOC REQUETE
          MOVE.L #$00009000,8(A2)      ATTENTE D'EVENEMENT
          TRAP   4                     APPEL SYSTEME
          BRA    PROPAG6
PROPAG7   TRAP   9                     JETTE BLOC D'ECRITURE
PROPAG5   ADD.L  #6,A0                 POINTE SUR LA F.O SUIVANTE
          DBRA   D3,PROPAG4
          RTS
*-------------------------------------------------------------------------

* PROCEDURE DE TRAITEMENT DE L'ORDRE
```

```
* PROCEDURE DE TRAITEMENT DE L'ORDRE GO

PGO     MOVE.L  20(A5),@TABDON(A6) CHARGE ADRESSE BUFFER DE DONNEE
        MOVE.L  @TABRE(A6),D7      POINTE SUR LA TABLE DES RESSOURCES
        BEQ     PGO5               SI NUL PAS D'ATTENTE
        MOVE.L  D7,A1              ADRESSE TABLE DES RESSOURCES
        TST.B   (A1)+              RESSOURCE DE CADENCEMENT ?
        BPL     PGO1               NON: PAS DE DEMASQUAGE
        CLR.W   D7                 RAZ D7 POUR DBRA
        MOVE.B  (A1)+,D7           NOMBRE DE RESSOURCES
        SUB.W   #1,D7              PREPARATION DBRA
PGO2    MOVE.L  (A1)+,A2           RECUPERE REQUETE
        MOVE.W  14(A2),28(A2)      REACTUALISE COMPTEUR D'EVEN
        BSET.B  #6,10(A2)          GENERATION EVENEMENT
        DBRA    D7,PGO2
PGO1    EQU     *
* Lect ressources
PGO5    TST.B   CRFP(A6)           FP ACCEPTE ?
        BMI     PGO3               NON: PAS DE MODIF DES TABLES
        MOVE.L  #PFP,@FP(A6)       CHARGE @ TRAITEMENT FP
PGO3    TST.B   CRSUSP(A6)         SUSP ACCEPTE ?
        BMI     PGO4
        MOVE.L  #PSUSP,@SUSP(A6)   CHARGE @ TRAITEMENT SUSP
PGO4    MOVE.B  CRGO(A6),D0        CHARGE C.R.
        BMI     PS2                SI GO NON ACCEPTE, PAS DE C.R.
        ECRICR@ @REQ(A6)               ENVOI C.R.

*-------------------------------------------------------------------------

* ATTENTE DE TOUTES LES CONDITIONS D'ACTIVATION DE LA TACHE

PS2     MOVE.L  #CONEPA,@GO(A6)    N'ACCEPTE PLUS GO
PS2.0   JSR     LECFO              LECTURE F.O. ENTREE
        LEA     PS2.1,A4           CHARGE @ DE RETOUR
PS2.1   EQU     *
        MOVE.L  @TABRE(A6),D5      ADRESSE DE LA TABLES DES RESSOURCES
        BEQ     PS2.6              SI NUL PAS D'ATTENTE
        MOVE.L  D5,A3              ADRESSE DE LA TABLE DES RESSOURCES
        TST.B   (A3)+              RESSOURCE DE CADENCEMENT ?
        BPL     PS2.2              NON: PAS D'ATTENTE
        CLR.W   D5                 PREPARATION DBRA
        MOVE.B  (A3)+,D5           NOMBRE DE RESSOURCES
        SUB.W   #1,D5              INIT DBRA
        LEA     PS2.5,A4           CHARGE @ DE RETOUR
PS2.4   MOVE.L  (A3)+,A1           POINTE SUR LA REQUETE
PS2.5   WPRIOC@ @REQ(A6),A1,EVTFO,A5
        DBRA    D5,PS2.4           ONE MORE TIME
        LEA     PS2.2,A4           CHARGE @ DE RETOUR
PS2.2   EQU     *
* faire pour toutes les ressources de donnees
*       WPRIO@  NREQ,REQDON,EVTFO,A5
* sauver A5 pointeur de donnees
        BRA     DEBCOD
PS2.6   EQU     *
        LEA     DEBCOD,A4          ADRESSE DE RETOUR
        MOVE.L  @REQ(A6),A1         REQUETE LECTURE F.O
        TST.B   11(A1)             EVENEMENT ARRIVE ?
        BNE     DEBCOD             NON: ON EXECUTE LE CODE
        MOVE.L  16(A1),A5          RECUPERE LA REQUETE D'ECRITURE
        BRA     EVTFO              TRAITEMENT DE L'ORDRE

DEBCOD  MOVEM.L A4-A6,-(A7)        SAVE SOME IMPORTANT REGISTERS
        MOVE.L  @DEB(A6),A3        ADRESSE DE DEBUT DU CODE
        MOVE.L  @TABDON(A6),A0     CHARGE ADRESSE BUFFER DE DONNEE
```

```
        JBUFST  #NRES.256,D0,#256 RENVOI BUFFER DE DONNEE
SUIT2   MOVE.L  @TABRE(A6),D7     ADRESSE DE LA TABLE DES RESSOURCE
        BEQ     SUIT0             SI NUL ON NE MASQUE PAS LES EVENE
        MOVE.L  D7,A0             ADRESSE DE LA TABLE DES RESSOURCE
        TST.B   (A0)+             RESSOURCE DE CADENCEMENT ?
        BPL     SUIT0             NON: ON SAUTE
        CLR.W   D7                PREPARE DBRA
        MOVE.B  (A0)+,D7          NOMBRE DE RESSOURCES
        SUB.W   #1,D7             PREPARE DBRA
SUIT1   MOVE.L  (A0)+,A2          POINTE SUR LA REQUETE
        AND.B   #$BF,10(A2)       MASQUE LES EVENEMENTS
        DBRA    D7,SUIT1          ON CONTINUE CE PETIT JEU ...
SUIT0   MOVE.L  #CONEPA,@FP(A6)
        MOVE.L  #CONEPA,@SUSP(A6)
        MOVE.L  #PGO,@GO(A6)      CHARGE @ TRAITEMENT DE GO
        BCLR.B  #7,FNDRAP(A6)     A T'ON RECU UN FN? ?
        BEQ     NOFN              NON: PAS DE C.R.
        MOVE.B  CRFN(A6),D0       CHARGE LE C.R.
        ECRICR@ REQFN(A6)         ENVOI C.R.
NOFN    LEA     PS1,A4            CHARGE @ RETOUR
        BRA     PS1.1
```

Macro-instructions utilisées

```
%NAM00    TIMER@    %NRTIM,%CADENCE,%REQUETE
*  CONNECTION A UNE RESSOURCE DE CADENCEMENT TIMER
%NAM00    TRAP      8                    DEMANDE BLOC REQUETE
          MOVE.L    #$14406800,8(A2)     CONNECTION TIMER
*       SANS ATTENTE,GENERATION D'EVENENEMENT,CHAINAGE
          MOVE.W    %NRTIM,12(A2)     NOM DE LA RESSOURCE
          MOVE.L    #1,16(A2)            DELAI UNITE
          MOVE.L    %CADENCE,20(A2)    CADENCE
          CLR       28(A2)
          TRAP      4                     APPEL SYSTEME
          MOVE.L    A2,%REQUETE
          MEND


          MACRO
%NAM00    EVEN@     %NREVEN,%REQUETE
* CONNECTION A UNE RESSOURCE DE CADENCEMENT PAR EVENEMENT SOFT
%NAM00    TRAP      8                    DEMANDE BLOC REQUETE
          MOVE.L    #$14006800,8(A2)     CONNECTION RES EVEN
*       SANS ATTENTE,GENERATION D'EVENEMENT,CHAINAGE
          MOVE.W    %NREVEN,12(A2)     NOM DE LA RESSOURCE
          CLR       28(A2)
          TRAP      4                     APPEL SYSTEME
          MOVE.L    A2,%REQUETE
          MEND



          MACRO
%NAM00    IT@       %NRIT,%REQUETE
* CONNECTION A UNE RESSOURCE DE CADENCEMENT PAR IT HARD
%NAM00    TRAP      8                    DEMANDE BLOC REQUETE
          MOVE.L    #$14006800,8(A2)     CONNECTION RES IT
*       SANS ATTENTE,GENERATION D'EVENEMENT,CHAINAGE
          MOVE.W    %NRIT,12(A2)       NOM DE LA RESSOURCE
          CLR       28(A2)
          TRAP      4                     APPEL SYSTEME
          MOVE.L    A2,%REQUETE
          MEND



          MACRO
%NAM00    WAITC@    %REQUETE
* ATTENTE D'EVENEMENT SUR UNE RESSOURCE DE CADENCEMENT
%NAM00    MOVE.L    %REQUETE,A0            ADRESSE REQUETE
          MOVE      28(A0),D0
WAITC0%:  CMP       14(A0),D0              EVEN GENERE ?
          BNE       WAITC1%:               SI OUI ON SORT
          TRAP      8                      SI NON DEMANDE BLOC REQUETE
          MOVE.L    #$00009000,8(A2)       ATTENTE EVEN,JETTE BLOC
          TRAP      4                      APPEL SYSTEME
          BRA       WAITC0%:               BRANCHEMENT TEST
WAITC1%:  MOVE      14(A0),28(A0)          REACTULISE COMPTEUR EVEN
          MEND
```

```
* ECRITURE SUR FILE D'ORDRE AVEC DEMANDE DE COMPTE-RENDU
%NAM00    TRAP     8                        DEMANDE BLOC REQUETE
          MOVE.L   #$1FFF6800,8(A2)         ECRITURE FILE D'ORDRE
* SANS ATTENTE,AVEC GEN EVEN,CHAINAGE,GARDE LE BLOC
          MOVE.W   %NRFILO,12(A2)       NUMERO DE RESSOURCE
          MOVE.L   %EA1,16(A2)           ADRESSE REQUETE
          MOVE.L   %EA2,20(A2)           2eme PARAMETRE
          TRAP     4                     APPEL SYSTEME
          MOVE.L   A2,%REQUETE
          MEND


          MACRO
%NAM00    EFONCR@  %NRFILO,%EA1,%EA2
* ECRITURE SUR FILE D'ORDRE SANS DEMANDE DE COMPTE-RENDU
%NAM00    TRAP     8                        DEMANDE BLOC REQUETE
          MOVE.L   #$1FFF3800,8(A2)         ECRITURE FILE D'ORDRE
* SANS ATTENTE,SANS GEN EVEN,CHAINAGE,JETTE LE BLOC
          MOVE.W   %NRFILO,12(A2)       NUMERO DE RESSOURCE
          MOVE.L   %EA1,16(A2)           ADRESSE REQUETE
          MOVE.L   %EA2,20(A2)           2eme PARAMETRE
          TRAP     4
          MEND


          MACRO
%NAM00    LFO@     %NRFILO,%REQUETE
* LECTURE SUR FILE D'ORDRE
%NAM00    TRAP     8                        DEMANDE BLOC REQUETE
          MOVE.L   #$1F006800,8(A2)         LECTURE FILE D'ORDRE
* SANS ATTENTE,GEN EVEN,CHAINAGE,GARDE LE BLOC
          MOVE.W   %NRFILO,12(A2)       NUMERO DE RESSOURCE
          TRAP     4                     APPEL SYSTEME
          MOVE.L   A2,%REQUETE
          MEND


          MACRO
%NAM00    WAIT@    %REQUETE,%EA
* ATTENTE D'EVEN SUR UNE RESSOURCE AUTRE QUE CADENCEMENT
%NAM00    MOVE.L   %REQUETE,A0               ADRESSE REQUETE
WAIT0%:   TST.B    11(A0)                  TEST CODE DE RETOUR
          BEQ      WAIT1%:                 SI NUL ON SORT
          TRAP     8                       SINON DEMANDE BLOC REQUETE
          MOVE.L   #$00009000,8(A2)        POUR MISE EN ATTENTE D'EVEN
          TRAP     4                       APPEL SYSTEME
          BRA      WAIT0%:                 RETOUR
WAIT1%:   MOVE.L   16(A0),%EA
          MEND


          MACRO
%NAM00    ECRICR@  %REQUETE
* RENVOIE UN COMPTE-RENDU SUR UNE FILE D'ORDRE.
* LE COMPTE-RENDU EST CONTENU DANS RQ  REQUEQ ADRE
```

```
        CLR.B     D0                    CODE RETOUR = OK
        TRAP      2                     REND REQUETE D'ECRITURE
        MOVE.L    A0,A2                 A2 POINTE SUR REQUETE DE LECTUR
        TRAP      9                     REND LE BLOC REQUETE DE LECTURE
        MEND


        MACRO
%NAM00  RENBLOC@ %REQUETE
* REND UN BLOC REQUETE ASSOCIE A UNE RESSOURCE
%NAM00  MOVE.L    %REQUETE,A2             RECUPERE L'ADRESSE DE LA REQUET
        TRAP      9                     REND LE BLOC AU SYSTEME
        MEND


        MACRO
%NAM00  WPRIO@    %REQFILO,%REQRES,%ADR,%EA
* ATTENTE D'EVENEMENT SUR DEUX RESSOURCES
%NAM00  MOVE.L    %REQFILO,A0           BLOC REQUETE
        MOVE.L    %REQRES,A1             IDEM
WPRI00%: TST.B    11(A0)                TEST SI REQUETE FILO TRAITEE
        BEQ       WPRI01%:              SI OUI ON SORT
        TST.B     11(A1)                SI NON TEST SI REQUETE RES TRAI
        BEQ       WPRI02%:              SI OUI ON SORT
        TRAP      8                     DEMANDE BLOC REQUETE
        MOVE.L    #$00009000,8(A2)      ATTENTE EVENEMENT
        TRAP      4                     APPEL SYSTEME
        BRA       WPRI00%:              RETOUR
WPRI01%: MOVE.L   16(A0),%EA            ADRESSE PARAMETRE
        BRA       %ADR
WPRI02%: MOVE.L   16(A1),%EA            IDEM
        MEND


        MACRO
%NAM00  WPRIOC@  %REQFILO,%REQCAD,%ADR,%EA
* ATTENTE D'EVENEMENT SUR DEUX RESSOURCES DONT UNE DE CADENCEMENT
%NAM00  MOVE.L    %REQFILO,A0           BLOC REQUETE
        MOVE.L    %REQCAD,A1            IDEM
        MOVE      28(A1),D0             COMPTE D'EVEN
WPRIC0%: TST.B    11(A0)                TEST SI REQUETE FILO TRAITEE
        BEQ       WPRIC1%:              SI OUI ON SORT
        CMP       14(A1),D0             EVEN CAD GENERE ?
        BNE       WPRIC2%:              SI OUI ON SORT
        TRAP      8                     SI NON DEMANDE BLOC REQUETE
        MOVE.L    #$00009000,8(A2)      ET MISE EN ATTENTE D'EVEN
        TRAP      4                     APPEL SYSTEME
        BRA       WPRIC0%:              RETOUR
WPRIC1%: MOVE.L   16(A0),%EA            ADRESSE PARAMETRE
        BRA       %ADR
WPRIC2%: MOVE     14(A1),28(A1)         REACTUALISE COMPTE D'EVEN
        MEND
```

```
*  CONNECTION A UNE RESSOURCE DE CADENCEMENT TIMER
%NAMOO    TRAP      8                        DEMANDE BLOC REQUETE
          MOVE.L    #$14406800,8(A2)         CONNECTION TIMER
*       SANS ATTENTE,GENERATION D'EVENENENEMENT,CHAINAGE
          MOVE.W    #%NRTIMER,12(A2)         NOM DE LA RESSOURCE
          MOVE.L    #1,16(A2)                DELAI UNITE
          MOVE.L    #%CADENCE,20(A2)         CADENCE
          CLR       28(A2)
          TRAP      4                        APPEL SYSTEME
          BRA       TIMF%:
@%NRTIMER DC.L      1
TIMF%:    MOVE.L    A2,@%NRTIMER
          MEND


          MACRO
%NAMOO    EVEN      %NREVEN
          EXTRN.W   %NREVEN
* CONNECTION A UNE RESSOURCE DE CADENCEMENT PAR EVENEMENT SOFT
%NAMOO    TRAP      8                        DEMANDE BLOC REQUETE
          MOVE.L    #$14006800,8(A2)         CONNECTION RES EVEN
*       SANS ATTENTE,GENERATION D'EVENEMENT,CHAINAGE
          MOVE.W    #%NREVEN,12(A2)          NOM DE LA RESSOURCE
          CLR       28(A2)
          TRAP      4                        APPEL SYSTEME
          BRA       EVEN%:
@%NREVEN  DC.L      1
EVEN%:    MOVE.L    A2,@%NREVEN
          MEND



          MACRO
%NAMOO    IT        %NRIT
          EXTRN.W   %NRIT
* CONNECTION A UNE RESSOURCE DE CADENCEMENT PAR IT HARD
%NAMOO    TRAP      8                        DEMANDE BLOC REQUETE
          MOVE.L    #$14006800,8(A2)         CONNECTION RES IT
*       SANS ATTENTE,GENERATION D'EVENEMENT,CHAINAGE
          MOVE.W    #%NRIT,12(A2)            NOM DE LA RESSOURCE
          CLR       28(A2)
          TRAP      4                        APPEL SYSTEME
          BRA       IT%:
@%NRIT    DC.L      1
IT%:      MOVE.L    A2,@%NRIT
          MEND



          MACRO
%NAMOO    GENEVEN   %NREVEN
          EXTRN.W   %NREVEN
* GENERATION D'EVENEMENT A UNE RESSOURCE D'EVENEMENT
%NAMOO    TRAP      8                        DEMANDE BLOC REQUETE
          MOVE.L    #$1A003800,8(A2)         GENE D'EVEN
*       SANS ATTENTE,SANS GENERATION D'EVENEMENT,CHAINAGE,BLOC RENDU
          MOVE.W    %NREVEN,12(A2)           NOM DE LA RESSOURCE
          TRAP      4                        APPEL SYSTEME
          MEND
```

```
          MACRO
%NAM00    WAITC     %NOM
          EXTRN.W   %NOM
* ATTENTE D'EVENEMENT SUR UNE RESSOURCE DE CADENCEMENT
%NAM00    MOVE.L    @%NOM,A0            ADRESSE REQUETE
          MOVE      28(A0),D0
WAITC0%:  CMP       14(A0),D0           EVEN GENERE ?
          BNE       WAITC1%:            SI OUI ON SORT
          TRAP      8                   SI NON DEMANDE BLOC REQUETE
          MOVE.L    #$00009000,8(A2)    ATTENTE EVEN,JETTE BLOC
          TRAP      4                   APPEL SYSTEME
          BRA       WAITC0%:            BRANCHEMENT TEST
WAITC1%:  MOVE      14(A0),28(A0)       REACTULISE COMPTEUR EVEN
          MEND



          MACRO
%NAM00    DBUFST    %NRSPACE,%EA
          EXTRN.W   %NRSPACE
* DEMANDE DE BUFFER A UNE RESSOURCE D'ESPACE
%NAM00    TRAP      8                   DEMANDE BLOC REQUETE
          MOVE.L    #$20000800,8(A2)    DEMANDE BUFFER
* LECTURE STANDART,SANS ATTENTE,SANS GEN EVEN,SANS CHAINAGE
          MOVE      %NRSPACE,12(A2)     NUMERO RESSOURCE
          TRAP      4                   APPEL SYSTEME
          MOVE.L    16(A2),%EA          ADRESSE BUFFER
          TRAP      9                   RENVOI BLOC REQUETE A LA BANQUE
          MEND



          MACRO
%NAM00    JBUFST    %NRSPACE,%EA
          EXTRN.W   %NRSPACE
* RENVOI DE BUFFER DANS UNE RESSOURCE D'ESPACE
%NAM00    TRAP      8
          MOVE.L    #$20021800,8(A2)    ECRITURE BUFFER
* ECRITURE STANDART,SANS ATTENTE,SANS GEN EVEN,SANS CHAINAGE,REND BL
          MOVE      %NRSPACE,12(A2)     NUMERO RESSOURCE
          MOVE.L    %EA,16(A2)          ADRESSE BUFFER
          TRAP      4
          MEND



          MACRO
%NAM00    EFOCR     %NRFILO,%EA1,%EA2
          EXTRN.W   %NRFILO
* ECRITURE SUR FILE D'ORDRE AVEC DEMANDE DE COMPTE-RENDU
%NAM00    TRAP      8                   DEMANDE BLOC REQUETE
          MOVE.L    #$1FFF6800,8(A2)    ECRITURE FILE D'ORDRE
* SANS ATTENTE,GEN EVEN,CHAINAGE,GARDE LE BLOC
          MOVE.W    #%NRFILO,12(A2)     NUMERO DE RESSOURCE
          MOVE.L    %EA1,16(A2)         ADRESSE REQUETE
          MOVE.L    %EA2,20(A2)         2eme PARAMETRE
          TRAP      4                   APPEL SYSTEME
          BRA       EFOCR%:
```

```
          MACRO
%NAM00    EFONCR    %NRFILO,%EA1,%EA2
          EXTRN.W   %NRFILO
* ECRITURE SUR FILE D'ORDRE SANS DEMANDE DE COMPTE-RENDU
%NAM00    TRAP      8                        DEMANDE BLOC REQUETE
          MOVE.L    #$1FFF3800,8(A2)         ECRITURE FILE D'ORDRE
* SANS ATTENTE,SANS GEN EVEN,CHAINAGE,JETTE LE BLOC
          MOVE.W    #%NRFILO,12(A2)          NUMERO DE RESSOURCE
          MOVE.L    %EA1,16(A2)              ADRESSE REQUETE
          MOVE.L    %EA2,20(A2)              IDEM
          TRAP      4
          MEND



          MACRO
%NAM00    LFO       %NRFILO
          EXTRN.W   %NRFILO
* LECTURE SUR FILE D'ORDRE
%NAM00    TRAP      8                        DEMANDE BLOC REQUETE
          MOVE.L    #$1F006800,8(A2)         LECTURE FILE D'ORDRE
* SANS ATTENTE,GEN EVEN,CHAINAGE,GARDE LE BLOC
          MOVE.W    #%NRFILO,12(A2)          NUMERO DE RESSOURCE
          TRAP      4                        APPEL SYSTEME
          BRA       LFO%:
@%NRFILO  DC.L      1
LFO%:     MOVE.L    A2,@%NRFILO
          MEND



          MACRO
%NAM00    WAIT      %NRES,%EA
          EXTRN.W   %NRES
* ATTENTE D'EVEN SUR UNE RESSOURCE AUTRE QUE CADENCEMENT
%NAM00    MOVE.L    @%NRES,A0                ADRESSE REQUETE
WAIT0%:   TST.B     11(A0)                   TEST CODE DE RETOUR
          BEQ       WAIT1%:                  SI NUL ON SORT
          TRAP      8                        SINON DEMANDE BLOC REQUETE
          MOVE.L    #$00009000,8(A2)         POUR MISE EN ATTENTE D'EVEN
          TRAP      4                        APPEL SYSTEME
          BRA       WAIT0%:                  RETOUR
WAIT1%:   MOVE.L    16(A0),%EA
          MEND



          MACRO
%NAM00    GETBUF    %NRESES,%EA
          EXTRN.W   %NRESES
* DEMANDE D'UN BUFFER A LA RESSOURCE ESPACE MEMOIRE ASSOCIEE A UNE E
%NAM00    TRAP      8                        DEMANDE D'UN BLOC REQUETE
          MOVE.L    #$20C0E800,8(A2)         DEMANDE D'UN BUFFER
          MOVE.W    %NRESES,12(A2)           NUMERO DE LA RESSOURCE E/S
          TRAP      4                        ENVOI REQUETE
          MOVE.L    16(A2),%EA               ADRESSE DU BUFFER
          TRAP      8
```

```
%NAM00    RENBUF    %NRESES,%EA
          EXTRN.W   %NRESES
* REND UN BUFFER A LA RESSOURCE ESPACE MEMOIRE ASSOCIEE A UNE E/S
%NAM00    TRAP      8                    DEMANDE D'UN BLOC REQUETE
          MOVE.L    #$20E01800,8(A2)     Rede Caesaris ...
          MOVE.L    %EA,16(A2)           ... bufferum quod Caesaris est.
          MOVE.W    %NRESES,12(A2)       NUMERO DE RESSOURCE E/S
          TRAP      4                    ENVOI REQUETE
          MEND


          MACRO
%NAM00    ECRICR    %NRFILO
          EXTRN.W   %NRFILO
* RENVOIE UN COMPTE-RENDU SUR UNE FILE D'ORDRE.
* LE COMPTE-RENDU EST CONTENU DANS D0. DESTROY REGISTER A0 !!
%NAM00    MOVE.L    @%NRFILO,A0          ADRESSE DE LA REQUETE DE LECTUR
          MOVE.L    16(A0),A2            ADRESSE DE LA REQUETE D'ECRITUR
          MOVE.L    D0,16(A2)            ECRITURE DU COMPTE-RENDU
          CLR.B     D0                   CODE RETOUR = OK
          TRAP      2                    REND REQUETE D'ECRITURE
          MOVE.L    A0,A2                A2 POINTE SUR REQUETE DE LECTUR
          TRAP      9                    REND LE BLOC REQUETE DE LECTURE
          MEND


          MACRO
%NAM00    RENBLOC   %NRES
          EXTRN.W   %NRES
* REND UN BLOC REQUETE ASSOCIE A UNE RESSOURCE
%NAM00    MOVE.L    @%NRES,A2            RECUPERE L'ADRESSE DE LA REQUET
          TRAP      9                    REND LE BLOC AU SYSTEME
          MEND


          MACRO
%NAM00    WPRIO     %NRFILO,%NRES,%ADR,%EA
          EXTRN.W   %NRFILO
          EXTRN.W   %NRES
* ATTENTE D'EVENEMENT SUR DEUX RESSOURCES
%NAM00    MOVE.L    @%NRFILO,A0          BLOC REQUETE
          MOVE.L    @%NRES,A1            IDEM
WPRI00%:  TST.B     11(A0)               TEST SI REQUETE FILO TRAITEE
          BEQ       WPRI01%:             SI OUI ON SORT
          TST.B     11(A1)               SI NON TEST SI REQUETE RES TRAI
          BEQ       WPRI02%:             SI OUI ON SORT
          TRAP      8                    DEMANDE BLOC REQUETE
          MOVE.L    #$00009000,8(A2)     ATTENTE EVENEMENT
          TRAP      4                    APPEL SYSTEME
          BRA       WPRI00%:             RETOUR
WPRI01%:  MOVE.L    16(A0),%EA           ADRESSE PARAMETRE
          BRA       %ADR
WPRI02%:  MOVE.L    16(A1),%EA           IDEM
```

```
          EXTRN.W  %NRFILO
          EXTRN.W  %NRCAD
* ATTENTE D'EVENEMENT SUR DEUX RESSOURCES DONT UNE DE CADENCEMENT
%NAM00    MOVE.L   @%NRFILO,A0          BLOC REQUETE
          MOVE.L   @%NRCAD,A1           IDEM
          MOVE     28(A1),D0            COMPTE D'EVEN
WPRIC0%:  TST.B    11(A0)               TEST SI REQUETE FILO TRAITEE
          BEQ      WPRIC1%:             SI OUI ON SORT
          CMP      14(A1),D0            EVEN CAD GENERE ?
          BNE      WPRIC2%:             SI OUI ON SORT
          TRAP     8                    SI NON DEMANDE BLOC REQUETE
          MOVE.L   #$00009000,8(A2)     ET MISE EN ATTENTE D'EVEN
          TRAP     4                    APPEL SYSTEME
          BRA      WPRIC0%:             RETOUR
WPRIC1%:  MOVE.L   16(A0),%EA           ADRESSE PARAMETRE
          BRA      %ADR
WPRIC2%:  MOVE     14(A1),28(A1)        REACTUALISE COMPTE D'EVEN
          MEND


          MACRO
%NAM00    PDTASK   %NRTASK,%EA
          EXTRN.W  %NRTASK
* DEMANDE D'INFORMATION SUR UNE TACHE: POINTEUR DE DONNEES
%NAM00    TRAP     8                    DEMANDE BLOC REQUETE
          MOVE.L   #$1C00E800,8(A2)
          MOVE.W   #%NRTASK,12(A2)
          TRAP     4                    APPEL SYSTEME
          MOVE.L   20(A2),%EA           POINTEUR DE DONNEES
          TRAP     9                    JETTE BLOC
          MEND


          MACRO
%ETIQ     ENDTSK   %DEBTSK,%TSKTAB
%ETIQ     RTS
          EXTRN    CONEPA
          ENTRY    %TSKTAB

* CREATION DES VARIABLES ET TABLES DE LA TACHE

%TSKTAB   DC.B     $80,$80,$08,$80      RESERVE PLACE POUR C.R.
          DC.B     $80,$80,$80,0
          DC.L     CONEPA,CONEPA,CONEPA,CONEPA
          DC.L     CONEPA,CONEPA,CONEPA,%DEBTSK
          DC.L     0,0,0,0,0,0
          MEND


          MACRO
%ETIQ     GETFO    %Dn,%An,%RETOUR
          EXTRN    TBPRES
%ETIQ     LEA      TBPRES,A2
          MOVE.L   56(%An),A0
          ASL.W    #3,%Dn
```

```
        MACRO
:TIQ    LIBFO   %Dn,%An
        EXTRN   TBPRES
:TIQ    LEA     TBPRES,A2
        MOVE.L  56(%An),A0
        ASL.W   #3,%Dn
        MOVE.W  -4(A0,%Dn),D0
        CLR.B   0(A2,D0.W)
        CLR.B   -6(A0,%Dn)
        MEND
```

ANNEXE C
ANALYSEUR LEXICOGRAPHIQUE

```
%C

/* Definition des classes de caracteres */

C       [0-9]                                   /* Chiffres */
L       [A-Za-z_]                               /* Lettres */
A       [A-Za-z0-9_]                            /* Symboles alphanumeriques */
H       [0-9A-Fa-f]                             /* Caracteres Hexadecimaux */
B       [ \r\t\f\032]                           /* Blancs */
P       [^ \r\t\f\032A-Za-z0-9_;{}]             /* Signes de ponctuation */
S       [;{}]                                   /* Separateurs d'instruction */
W       [blwBLW]                                /* Specificateurs d'adresse */

%{
#include <stdio.h>
#include "config.h"
#include "ytab.h"
#include "extern.h"

static char c = 0, *pointeur = NULL;
static unsigned short base = 0, offset = 0;
static unsigned long valeur = 0;
%}

%%
{B}     ;

\n      {if ( (pmodndx == 0) && evflag ) {      /* Step permis que sur le P/P
            if ( (nligne >= ligne1) && (nligne <= ligne2))
               wtresum();
         }
        }

"<="    return(INFEG);                          /* Operateurs */
">="    return(SUPEG);
"!="    return(NONEG);
"&&"    return(ETLOG);
"||"    return(OULOG);

break   return(BREAK);                          /* Mots cles */
call    return(CALL);
case    return(CASE);
continue        return(CONTINUE);
default return(DEFAULT);
do      return(DO);
else    return(ELSE);
exit    return(EXIT);
global  return(GLOBAL);
int     return(INT);
if      return(IF);
pause   return(PAUSE);
return  return(RETURN);
wait    return(WAIT);
while   return(WHILE);

{L}{A}* {strcpy(tmpbuf, yytext);                /* Identificateur */
         tmpbuf[IDSIZ-1] = '\0';
         return(IDENT);
        }

\${H}+  {base = 16;                             /* Nombre Hexadecimal */
         pointeur = yytext + 1;
         goto conversion;
        }
```

```
(C)+    {base = 10;                          /* Nombre decimal */
         pointeur = yytext;
        /*
         *  Covnèrsion chaine de caracteres Hexa ou Decimale en entier 32 bit
         */
        conversion:
         valeur = 0;
         while ( c = *pointeur++ ) {
            offset = ( c <= '9' ? '0' : ( c <= 'F' ? 'A' : 'a' ) - 10 );
            valeur = valeur * base + c - offset;
         }
         yylval.valent = valeur;
         return(NOMBRE);
        }

@(W)    {yylval.sval = *(yytext + 1);    /* Specificateur d'adresse */
         return(ADR_SPE);
        }

(S)     {if (ykscrute()) {               /* Separateurs d'instructions */
            errflag = OK;
            RecoveryFlag = 0;
            return(0);
         }
         return(yytext[0]);
        }

"\\"    while ( (c = input()) != '\\' ) /* Commentaires */
            if (c == '\0') return(0);

(P)     {return(yytext[0]);}             /* Signe de ponctuation */
```

ANNEXE D
ANALYSEUR SYNTAXIQUE

```
%{
/*
 * @(#) Interpreteur CLIPS V2.02g  18/05/88  par P. Bacquet
 */
#define CESAR 1
#include <stdio.h>
#include <ctype.h>
#ifdef CESAR
#define stderr &LOC
#define stdout &LOC
#include <cesar.h>
#endif
#include "config.h"
#define PARENT 0
#define VIRG 1

extern signed char input(), *installe(), *allog(),
        *evalins(), *evalparam();
extern short BUFFERS, DIMBUF, LOC, TIMER;
extern PROGID chkmod();

extern int *recup();

/*
 *  Attention la plupart des initialisations suivantes sont bidons,
 *  tout ca a cause du compilo WhiteSmith
 */

short nligne = 0, ligne1 = 0, ligne2 = 0, pmodndx = 0;
static short parcpt = 0, bcompte = 0, ContCompte = 0;
static int casval = 0, crexec = 0, valeur = 0;
static RID a1 = NULL;
int berrctx[16] = {0};

static union {
   signed char *b;
   short *w;
   int *l;
} allptr = NULL;

signed char tmpbuf[IDSIZ] = {0}, *nextchar = NULL, ltrflag = FAUX, evflag = F
   errflag = 0, RecoveryFlag = 0, *lastchar = NULL;
static const signed char bidbuf[] = {ERRMARK, 3, 2, 2};

#define TMBFSZ 256
static signed char pilef[2 * STACKSIZ] = {0}, buffet[TMBFSZ] = {0},
   *pf_ptr = NULL, idflag = 0, LastFlag = 0, TypBuf[IDSIZ] = {0},
   IdBuf[IDSIZ] = {0}, *cptr = NULL, *adrbuf = NULL, *parptr = NULL;

static LINE pilep[STACKSIZ] = {NULL}, linerec = {NULL, 0};
static LINEPTR pp_ptr = NULL;

static PROGID pilemod[MODSIZ] = {0};
static PROGID a2 = NULL;

%}

%union {
   signed char *chrptr;
   int valent;
   short sval;
}

%token <chrptr> IDENT WHILE INT
```

```
%token <valent> NOMBRE
%token <sval>   INFEG SUPEG NONEG ADR_SPE
/*  %token FOR BY TO   */
%token BREAK CALL CASE CONTINUE DEFAULT DO ELSE EXIT GLOBAL IF PAUSE RETURN
%token WAIT

%left ETLOG OULOG
%left '|' '^'
%left '&'
%left NONEG '='
%left '<' '>' INFEG SUPEG
%left '+' '-'
%left '*' '/' '%'
%nonassoc '!' '~'

%start Module

%type <valent> app_fct atome const consts expr exp_unr ins_sim valeur
%type <sval>   adr_spe signe
%type <chrptr> type
%%

/* Debut des regles Yacc */

Module  : Ent_mod corps                 /* Regle utilise uniquement pour un P
                {clearniv(0);}          /* Init. table des sysmboles */
        ;

module  : ent_mod dcl_glos '{' ordres '}'
        | '\002'
                {nextchar = linerec.pos;
                 goto EndModule;
                }
        | ent_mod dcl_glos '{' ordres error '\002'
                {nextchar = pp_ptr->pos;
                EndModule:
                 yyclearin;
                 yyerrok;
                 RecoveryFlag = 0;
                }
        ;

corps   : dcl_glos '{' ordres '}'
        ;

Ent_mod : IDENT
                {/*
                  *  Recupere et enregistre le pointeur sur le descripteur de
                  *  module (P/P uniquement)
                  */
                  pilemod[++pmodndx] = chkmod(tmpbuf);
                }
          lst_pars
        ;

ent_mod : IDENT lst_pars
        ;

lst_pars: '('
                {idflag = PARAM;
                /*
                 * Suivant le niveau (PP ou SP) le pointeur sur les parametre
                 * est different, dans un cas la ligne provient du superviseu
                 * dans l'autre, elle est interne.
```

```
                */
                cptr = (pmodndx ? buffet : parptr);
               }
          dcl_pars ')' ';'
        ;

dcl_pars:    /* Rien */
        | dcl_par
        | dcl_pars ';' dcl_par
        ;

dcl_par : type
                {/* Memorise le type de variable dans une zone adquate */
                 strcpy(TypBuf, tmpbuf);}
          idents
        | INT id_exps
        ;

id_exps : id_exp
        | id_exps ',' id_exp
        ;

id_exp  : IDENT
                {$<chrptr>$ = nextchar;
                 strcpy(IdBuf, tmpbuf);
                 nextchar = cptr;
                 --pmodndx;
                }
          expr
                {if ( allptr.b = installe(IdBuf, $3, pmodndx + 1) ) {
                    if ( pmodndx >= 0 )
                        nligne = pp_ptr->lineno;
                    goto NonInterpreterError;
                 }
                 ++pmodndx;
                 yyclearin;
                 cptr = nextchar;
                 nextchar = $<chrptr>2;
                }
        ;

idents  : IDENT {goto Traitid;}
        | idents ',' IDENT
                {Traitid:
                 if ( idflag == GLOB ) {
                   /*
                    * Declaration globale
                    */
                    if ( allptr.b = allog(TypBuf, tmpbuf) ) {
                     NonInterpreterError:
                       errflag = BEBERR;
                       YYABORT;
                    }
                 }
                 else {
                   /*
                    * Parametres
                    */
                    if (allptr.b = evalparam(tmpbuf, TypBuf, pmodndx, &cptr))
                       if (--pmodndx >= 0)
                          nligne = pp_ptr->lineno;
                       goto NonInterpreterError;
                    }
                    do {
```

```
                        if (*cptr == '\0') break;
                    }
                    while ( *cptr++ != ',' );
                }
            }
        ;

dcl_glos: /* RIEN */
        | dcl_glo
        | dcl_glos dcl_glo
        ;

dcl_glo : GLOBAL
                {idflag = GLOB;}
          Type idents ';'
        ;

ordres  : ordre
        | ordres ordre
        ;

ordre_sp: ';'
        | ordre
        ;

ordre   : ins_sim ';'
        | blc_cnt
        | ins_spe ';'
        | '(' ordres ')'
        ;

blc_cnt: IF expr
              {fempile(evflag);
               if (evflag)
                  evflag = $2;
              }
         ordre go_else
              {evflag = fdepile();}
/*      | FOR IDENT '=' expr TO expr go_by DO ordre */
        | wblocs        /* Une boucle WHILE */
        | CASE expr '{'
              {$<valent>$ = casval;
               fempile(evflag);
               if (evflag) {
                  casval = $2;
               }
              }
          selects default '}'
              {evflag = fdepile();
               casval = $<valent>4;
              }
        ;

wblocs  : wbloc
        | wblocs '\001'
              {LoopAgain:
               nextchar = linerec.pos;
              }
          wbloc
        ;

wbloc   : WHILE expr DO
              {TraitBoucl:
               fempile(evflag);
```

IF 1

IF 3

CASE 1

CASE 2

WHILE 1

```
                if (evflag) {
                  /*
                   *  Si la boucle est executable, on empile les parametres
                   *  de retour en debut de boucle
                   */
                   evflag = $2;
                   bcompte++;
                   linerec.savpt = pf_ptr;
                   linerec.pos = $1;
                   pempile();
                }
               }
        ordre_sp
               {LastFlag = evflag;
                evflag = fdepile();
                if (evflag) {
                  /*
                   *  Si la boucle tait executable, on restaure les parametr
                   *  precedement empiles pour revenir en debut de boucle
                   */
                   pdepile();
                   bcompte--;
                   if ( LastFlag ) {
                      nextchar = bidbuf;
                      nligne = linerec.lineno;
                   }
                }
               }
        | WHILE expr DO error '\003'
               {WBlocErr:
                yyclearin;
                if (--ContCompte <= 0) {
                   yyerrok;
                   RecoveryFlag = 0;
                   nextchar = bidbuf;
                }
                else {
                   nextchar = bidbuf + 1;
                   YYERROR;
                }
               }
        ;

ins_spe : BREAK valeur
               {if (evflag) {
                   if (brkhandle(FAUX, $2)) YYABORT;
                   evflag = FAUX;
                }
               }
        | CONTINUE valeur
               {if (evflag) {
                  /*
                   *  Prepare l'execution de l'ordre continue grace a
                   *  l'utilisation du mecanisme de recueration d'erreurs
                   */
                   if (brkhandle(VRAI, $2)) YYABORT;
                   bcompte -= $2;
                   ContCompte = $2;
                   yyclearin;
                   nligne = linerec.lineno;
                   nextchar = bidbuf + 1;
                  /*
                   *  Generation de la "fausse" erreur qui permettra de depi
                   *  toutes les structures correspondant aux boucles d'ou l
```

WHILE 2

WHILE 3

CONT 2

CONT 3

```
                    *  doit sortir
                    */
                    errflag = NOERROR;
                    YYERROR;
                  }
                 }
        ;
```

CONT3

```
valeur  : /* RIEN */
                {$$ = 1;}
        | NOMBRE
        ;

go_else : /* RIEN */
        | ELSE
                {if ( *pf_ptr)
                    evflag = !evflag;
                }
          ordre
        ;

/*
go_by   :
        | BY expr
                {$$ = $2);}
        ;
*/
selects : select
        | selects  select
        ;

select  : consts ';'
                {fempile(evflag);
                 if ( evflag ) {
                    evflag = $1;
                 }
                }

          ordre
                {LastFlag = evflag;
                 evflag = fdepile();
                 if (LastFlag) evflag = FAUX;
                }
        ;

consts  : const
                {$$ = ( casval == $1);}
        | consts ',' const
                {$$ = ( $1 ? $1 : casval == $3);}
        ;

const   : signe NOMBRE {$$ = ( $1 == PLUS ? $2 : -$2);}
        ;

signe   : /* RIEN */ {$$ = PLUS;}
        | '+' {$$ = PLUS;}
        | '-' {$$ = MOINS;}
        ;

default : /* RIEN */
        | DEFAULT ':' ordre
                {if ( *pf_ptr && evflag )
```

IF3

CASE3

CASE4

CASE5

```
                    evflag = VRAI;
                }
        ;

ins_sim : INT IDENT
                {strcpy(IdBuf, tmpbuf);}
          adr_spe '=' expr
                {if (evflag) {
                    if ($4) {
                       if ( (allptr.l = recup(IdBuf, pmodndx)) == NULL) {
                          strcpy(tmpbuf, IdBuf);        /* Pour erreur */
                          errflag = NODECL;
                          YYABORT;
                       }
                       wraddr(*allptr.l, $4, $6);
                    }
                    else if ( allptr.b = installe(IdBuf, $6, pmodndx))
                       goto NonInterpreterError;
                }
                }
        | type IDENT
                {if (skiparg($1, VIRG))
                    YYABORT;
                 if (evflag) {
                     if (allptr.b = evalins(buffet, pmodndx))
                       goto NonInterpreterError;
                 }
                }
        | NOMBRE ADR_SPE '=' expr
                {if (evflag)
                    wraddr($1, $2, $4);
                }
        | app_fct
                {if (evflag)
                    if ($1) {
                      SsErr:
                       errflag = SSERR;
                       YYABORT;
                    }
                }
        | EXIT
                {if (evflag) {
                    killprog();
                    YYACCEPT;
                 }
                }
        | PAUSE
                {if (evflag)
                    wtresum();          /* On attend le start */
                }
        | WAIT '(' expr ')'
                {if (evflag) {
                   /*  Attente delai specifie */
                    rqwait(a1 = rqtim(&TIMER, $3));
                    rqtrash(a1);
                 }
                }
        | CALL IDENT '('
                {$<sval>$ = bcompte;
                 if (skiparg(nextchar, PARENT))
                    YYABORT;
                 if (evflag) {          /* Si saut au S/P executable */
                   /* Verifie que le module appele existe, sinon erreur */
                    if ( (a2 = chkmod(tmpbuf)) == NULL ) {
```

```
                errflag = NOMODULE;
                YYABORT;
            }
            /*
             * Verifie que le nombre max. de S/P imbriques n'est pas
             * trop grand; sinon, erreur
             */
            if (++pmodndx >= MODSIZ) {
                --pmodndx;          /* Pour restaurer le niveau courant *
                errflag = DPMOD;
                YYABORT;
            }
            /*
             *  Empile les parametres de retour a restaurer en fin de
             */
            pilemod[pmodndx] = a2;
            linerec.pos = nextchar;
            linerec.lineno = nligne;
            linerec.savpt = pf_ptr;
            pempile();
            nligne = 1;
            bcompte = 0;
            /*
             *  Decale le pointeur de decodage pour interpretation du
             */
            nextchar = a2->xeqadr;
        }
        else {
            /*
             * On saute le module "inexistant" en
             * generant une fausse erreur de syntaxe
             */
            linerec.pos = nextchar;
            yyclearin;
            nextchar = bidbuf + 2;
        }
    }
module
    {if (evflag) {
        /*
         *  Restaure les parametres de retour uniquement si le S/P
         *  etait executable
         */
        pdepile();
        clearniv(pmodndx--);        /* Libere les variables local
        pf_ptr = linerec.savpt;
        nligne = linerec.lineno;
        bcompte = $<sval>4;
    }
    nextchar = linerec.pos;         /* Pointe sur le ; */
    }
| RETURN
    {if (evflag) {
        if ( pmodndx == 0 ) {
            /*
             *  RETURN sur P/P equivalent a un ordre EXIT => fin
             *  de l'interpretation
             */
            killprog();
            YYACCEPT;
        }
        else {
            pp_ptr += bcompte;      /* On "sort" des boucles */
            yyclearin;
```

CALL 1

CALL 2

CALL 3

CALL 4

CALL 5

RETURN 1

```
                        /*
                         *  Et on tremine le S/P en cours
                         */
                         nextchar = bidbuf + 2;
                         errflag = NOERROR;
                         YYERROR;
                      }
                   }
                }
        ;

app_fct : IDENT '('
                {/*
                  *  Recopie de l'appel pour transmission aux routines
                  *  du superviseur.
                  */
                 if ( skiparg($1, PARENT) )
                    YYABORT;
                 if (evflag) {
                   /*
                    *  Pose un null en fin de buff(au cas ou) et appel au sup
                    */
                    buffet[127] = 0;
                    if ( (crexec = ykexec(buffet, &allptr)) < 0) {
                       goto SsErr;
                    }
                    else {
                       $$ = crexec;
                    }
                 }
                }
        ;

/* Calcul des expressions entieres. No comment !! */

expr    : exp_unr
        | expr '+' expr { $$ = $1 + $3;}
        | expr '-' expr { $$ = $1 - $3;}
        | expr '/' expr { $$ = $1 / $3;}
        | expr '*' expr { $$ = $1 * $3;}
        | expr '%' expr { $$ = $1 % $3;}
        | expr ETLOG expr { $$ = $1 && $3;}
        | expr OULOG expr { $$ = $1 || $3;}
        | expr '&' expr { $$ = $1 & $3;}
        | expr '|' expr { $$ = $1 | $3;}
        | expr '^' expr { $$ = $1 ^ $3;}
        | expr '<' expr { $$ = $1 < $3;}
        | expr '>' expr { $$ = $1 > $3;}
        | expr INFEG expr { $$ = $1 <= $3;}
        | expr SUPEG expr { $$ = $1 >= $3;}
        | expr NONEG expr { $$ = $1 != $3;}
        | expr '=' expr { $$ = $1 == $3;}
        | atome
        ;

exp_unr : '!' atome {$$ = ! $2;}
        | '~' atome {$$ = ~ $2;}
        | '-' atome %prec '!' {$$ = - $2;}
        ;

atome   : NOMBRE adr_spe
                {if (evflag)
                    $$ = rdaddr($1, $2);
                }
```

RETURN2

EXP7

```
        | IDENT adr_spe
                {/* Recuperation de la valeur d'une variable entiere */
                 if (evflag){
                    if ( (allptr.l = recup(tmpbuf, pmodndx)) == NULL) {
                       errflag = NODECL;
                       YYABORT;
                    }
                    else
                       $$ = rdaddr(*allptr.l, $2);
                 }
                }
        | app_fct
                {$$ = $1;
                 if ( evflag && ($1 > 0) )
                   /*
                    * Cas d'un echec digere, il faut rendre le buffer
                    * contenant le compte-rendu de la primitive
                    */
                    rqbbuf(&BUFFERS, allptr.b, &DIMBUF);
                }
        | '(' expr ')' {$$ = $2;}
        ;

Type    : type
        | INT {strcpy(TypBuf, "int");}
        ;

type    : IDENT
                {strcpy(TypBuf, tmpbuf);
                 $$ = $1;
                }
        ;

adr_spe : /* Rien */ {$$ = FAUX;}
        | ADR_SPE {$$ = $1;}
        ;

/* Fin des regles Yacc */

%%
static brkhandle(mode, nombre)
short mode;
register int nombre;
{
   register unsigned char *ptr;
   register LINEPTR lptr;
  /*
   * Le BREAK ou CONTINUE est il a l'interieur d'une boucle ?
   */
   if ( nombre > bcompte ) {
      errflag = NOWHILE;
      return(ERROR);
   }
   /*
    * On empeche l'evaluation des blocs contenus dans la boucle
    */
    ptr = (unsigned char *) pf_ptr;
    lptr = pp_ptr;
    while (nombre--) {
       while ( ptr < (unsigned char *) lptr->savpt )
          *ptr++ = FAUX;
       lptr++;
    }
    if (mode) {
```

```
        pf_ptr = ((signed char *) ptr) + 1;
        pp_ptr = lptr;
        cpybuf(&linerec, lptr - 1, sizeof(linerec));
     }
     return(OK);
}

cpybuf(dst, src, compte)
register char *src, *dst;
register short compte;
{
   while(compte--) *dst++ = *src++;
}

char fdepile()
{
   return( *pf_ptr++ );
}

fempile(flag)
signed char flag;
{
  if ( --pf_ptr < pilef ) {
     errflag = DPFLAG;
     killprog();
  }
  *pf_ptr = flag;
}

extern int yydebug, yyprevious;

signed char *interp(MainId, ptbuff)
PROGID MainId;
signed char *ptbuff;
{

   nligne = 1;
   ligne2 = ( (ligne1 = MainId->n1) ? MainId->n2 : 0 );
  /*
   *  Init. des varaibles de l'interpreteur
   */
   nextchar = MainId->xeqadr;
   adrbuf = ptbuff;
   do {                  /* Recherche des eventuels parametres d'appel */
      if ( *ptbuff++ == '(' ) break;
   } while (*ptbuff);
   parptr = ptbuff;
   bcompte = 0;
   evflag = VRAI;
   errflag = SYNERR;
   ltrflag = FAUX;
   RecoveryFlag = 0;
   yyprevious = 10;
   pf_ptr = pilef + 2 * STACKSIZ;       /* Pointeurs de piles */
   pp_ptr = pilep + STACKSIZ;
   pmodndx = -1;

   berron();                            /* Pour digerer une eventuelle erreur

   if (yyparse() == 0 ) {               /* Lance l'interpretation */
      errflag = OK;
   }

   berroff();
```

```
    if (errflag != SYNERR) yyerror();    /* Genere message de sortie */
    return(adrbuf);
}

/*
 *  Fonctions de getion des piles flags et structure de boucle
 */

static signed char *pdepile()
{
   cpybuf(&linerec, pp_ptr++, sizeof(linerec));
}

static pempile()
{
  if ( --pp_ptr < pilep ) {
     errflag = DPPTR;
     killprog();
  }
  linerec.lineno = nligne;
  cpybuf(pp_ptr, &linerec, sizeof(linerec));
}

/*
 *  Lecture d'un octet, ou mot ou mot long a une adresse donnee
 */

static rdaddr(adresse, taille)
int adresse;
short taille;
{
   allptr.b = (signed char *) adresse;
   switch (taille) {
      case FAUX:
         return(adresse);
      case 'b': case 'B':
         return( *(allptr.b) );
      case 'w': case 'W':
         return( *(allptr.w) );
      case 'l': case 'L':
         return( *(allptr.l) );
   }
}

/*
 * Cette fonction saute les arguments d'une fonction ou d'un sous-programme
 * et les recopie dans buffet, jusqu'a un ';' si mode = VIRG ou au dernier
 * ')' fermant si mode = PARENT
 */

static skiparg(ptr, mode)
signed char *ptr;
short mode;
{
   register signed char *cptr, c;
   short par_cpt, spflag = 0, tlbuf = 0;

   par_cpt = 0;
   nextchar = ptr;
   cptr = buffet;
   while ( ++tlbuf < TMBFSZ) {
      switch ( (*cptr++ = c = input()) {
         case '\0':
            errflag = EOFERR;
```

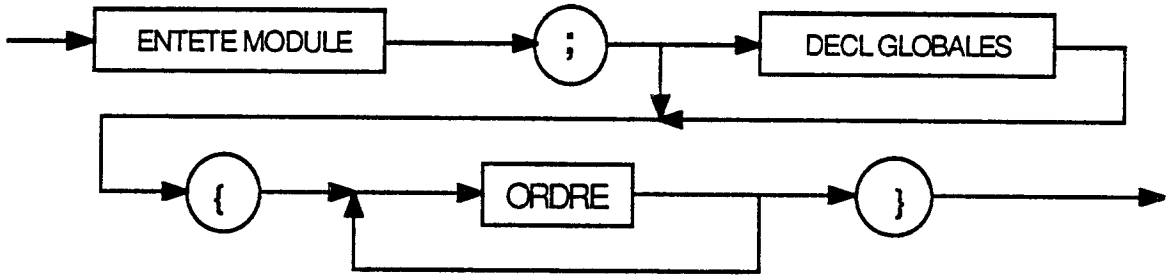
```
            return(!OK);
         case '\n':
         case  ' ':
         case '\t':
         case '\f':              /* Compacte les blancs */
            if (!spflag)
               *(cptr - 1) = ' ';
            else
               cptr--;
            break;
         case  ';':
            unput();             /* Restaure le ; */
            goto Enough;
         case  '(': par_cpt++; break;
         case  ')':
            if ( --par_cpt == 0 )
               if (mode == PARENT)
                  goto Enough;
      }
      spflag = isspace(c);       /* Memorise la "blancheur" du dernier car */
   }
  /*
   *  On n'arrive ici qu'en cas de debordement buffer
   */
   return(errflag = TLEX);

   Enough:
   *cptr = '\0';          /* Pose un NULL en fin de buffer */
   return(OK);
}

/*
 *  Idem rdaddr mais en ecriture
 */

static wraddr(adresse, taille, valeur)
int adresse, valeur;
short taille;
{
   allptr.b = (signed char *) adresse;
   switch (taille) {
      case 'b': case 'B':
         *(allptr.b) = valeur;
         break;
      case 'w': case 'W':
         *(allptr.w) = valeur;
         break;
      case 'l': case 'L':
         *(allptr.l) = valeur;
         break;
   }
}

yyerror()
{
   static const char *const errmess[] = {
      "0000  Ok %s",
      "Identifier %s undeclared",
      "Unknown module %s",
      "Fatal error: module stack overflow",
      "Fatal error: flag stack overflow",
      "Fatal error: pointer stack overflow",
      "Fatal error: yacc's internal stack overflow",
      "Syntax error",
```

```
        "Too large break or continue range",
        "Attempt to call an unloading subprogram",
        "Unexpected end of file",
        "Bus error",
        "Too long expression"
    };

    register char *ptr;
    register short flagerr;

    flagerr = errflag;
    /*
    * Digere les appels parasites et les "fausses" erreurs
    */
    if ( (RecoveryFlag++ > 0 ) || (flagerr < STOP) ) {
       errflag = SYNERR;
       return;
    }
    if ( pmodndx < 0 ) pmodndx = 0;
    if (flagerr == SSERR) {                          /* Si erreur Superviseur ...
       rqbbuf(&BUFFERS, adrbuf, &DIMBUF);            /* ... Echange les buffers */
       adrbuf = allptr.b;
    }
    switch (flagerr) {
       case SSERR:
          cptr = adrbuf;
          while ( *cptr++);
          sprintf(--cptr, ", module %s line %d",
             pilemod[pmodndx]->nom, nligne);
          goto GenErr;
       case BEBERR:
          sprintf(adrbuf, "%s, module %s line %d", allptr.b,
             pilemod[pmodndx]->nom, nligne);
          goto GenErr;
       case OK:
          sprintf(adrbuf, errmess[0], pilemod[0]->nom);
          goto Exit;
       case NODECL:
       case NOMODULE:
          sprintf(buffet, errmess[flagerr], tmpbuf);
          break;
       default:
          strcpy(buffet, errmess[flagerr]);
    }
    sprintf(adrbuf, "%04x  %s, module %s line %d",
       flagerr + 0xBFFF, buffet, pilemod[pmodndx]->nom, nligne);
  GenErr:
    errflag = SYNERR;
  Exit:
    killprog();
}

/* Fonction d'arret immediat de l'interpretation */

killprog()
{
   for (pmodndx = MODSIZ; pmodndx >= 0; pmodndx--)
      clearniv(pmodndx);
   nextchar = bidbuf + 4;          /* Pour sortir sur EOF */
}

/* Fonction de mise en attente (PAUSE) */

wtresum()
```

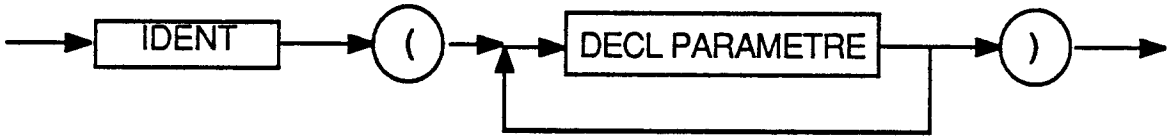
```
{
   char *buffer;

   buffer = rqgbuf(&BUFFERS);
   sprintf(buffer, "Program %s paused line %d.", pilemod[pmodndx]->nom ,nlign
   if (ykpause(buffer)) {
      killprog();
      errflag = OK;
   }
}
```

ANNEXE E
DIAGRAMMES SYNTAXIQUES DE LIPS

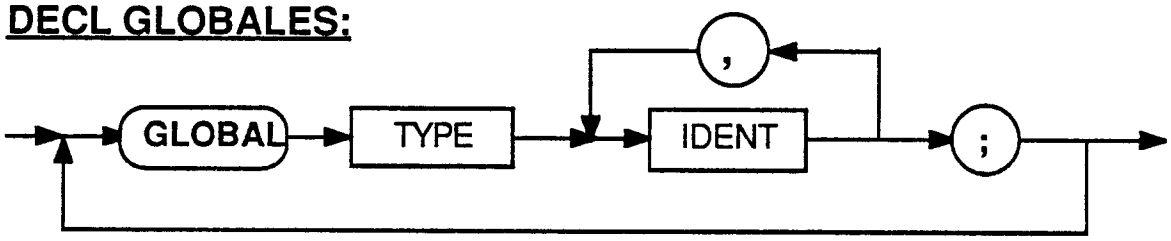
MODULE:



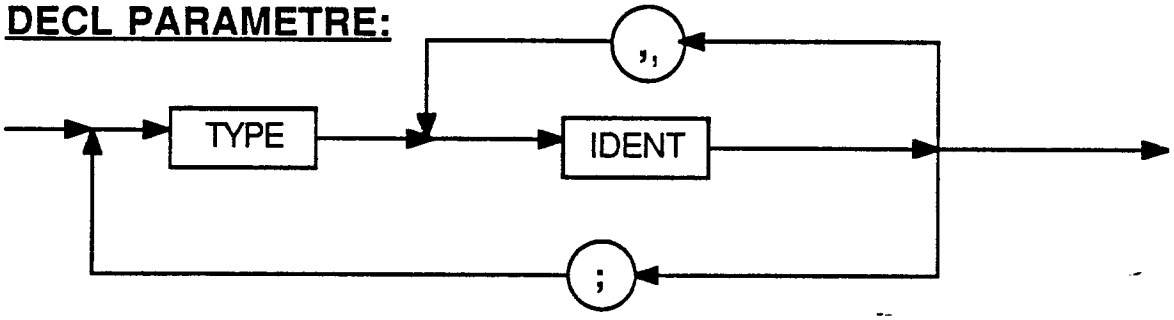
ENTETE MODULE:



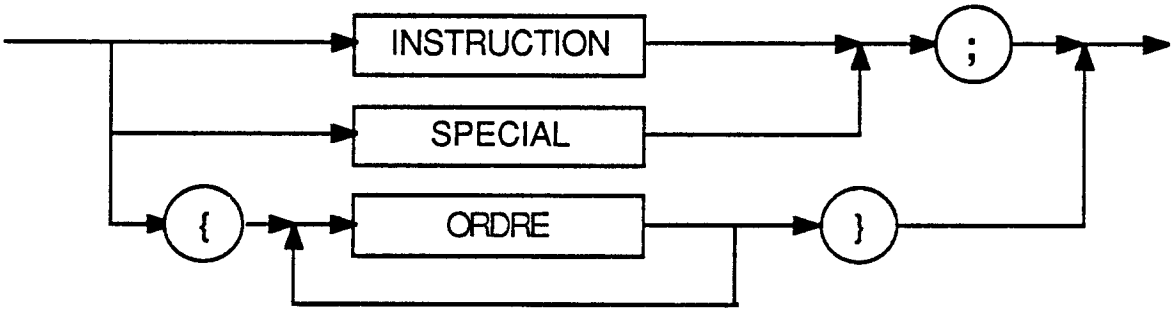
DECL GLOBALES:



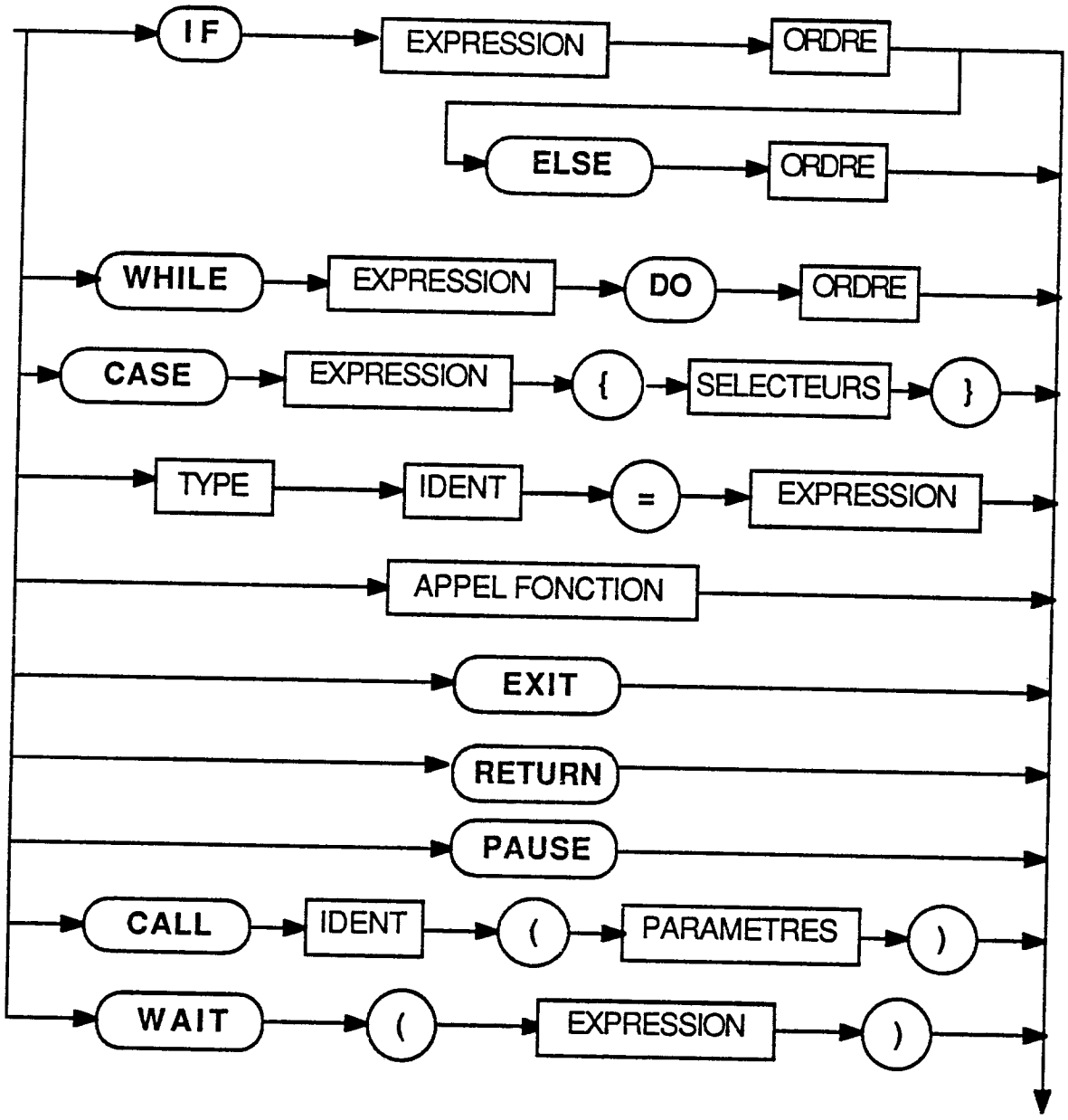
DECL PARAMETRE:



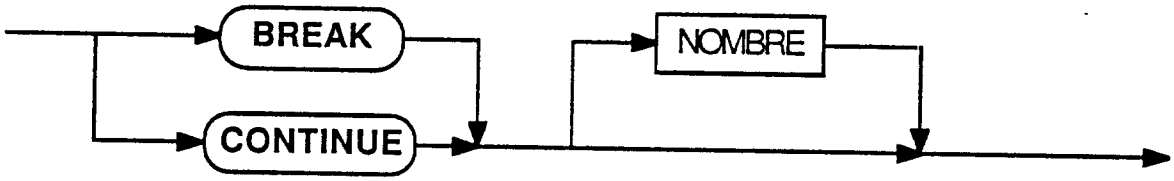
ORDRE:



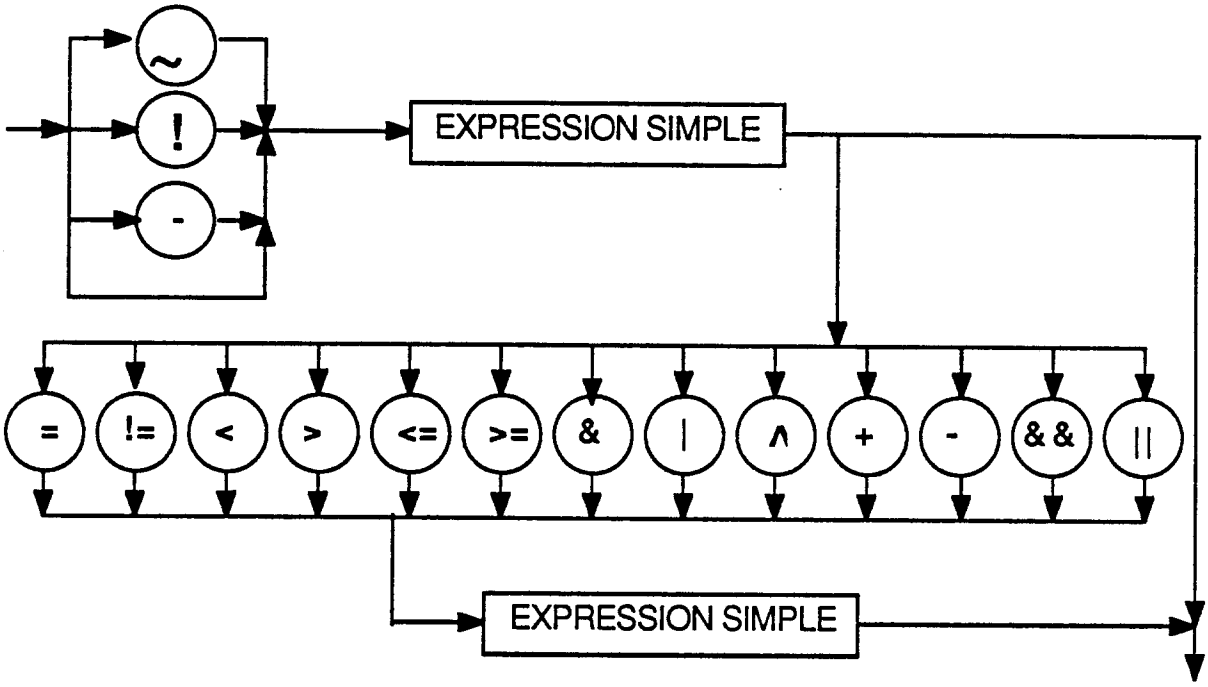
INSTRUCTION:



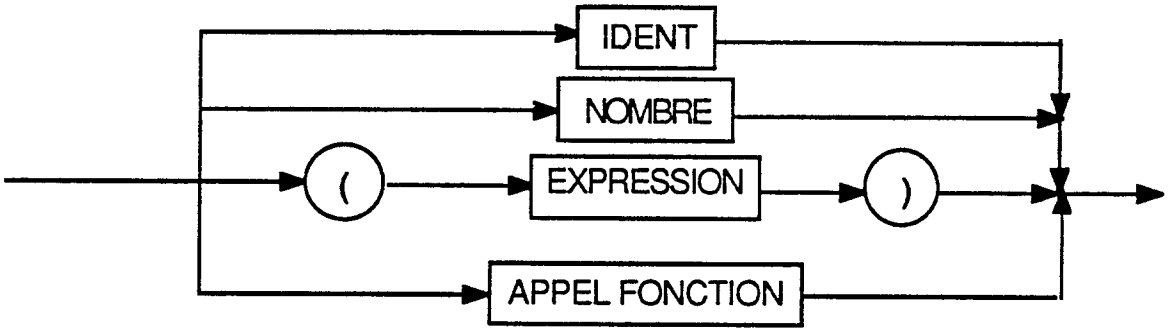
SPECIAL:



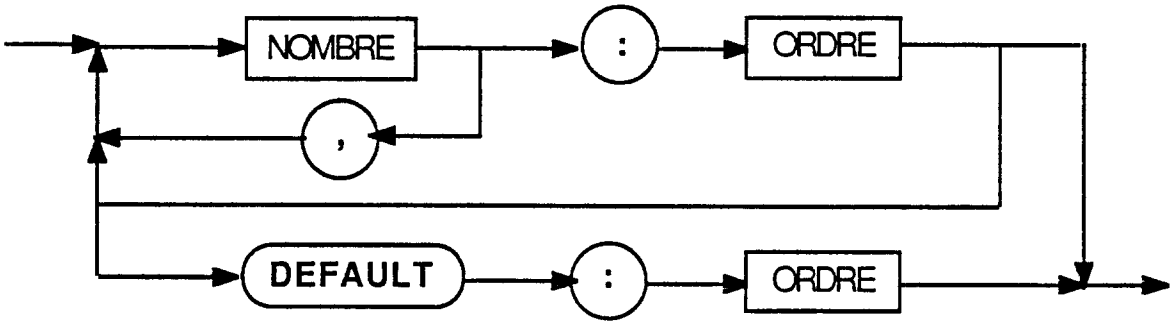
EXPRESSION:



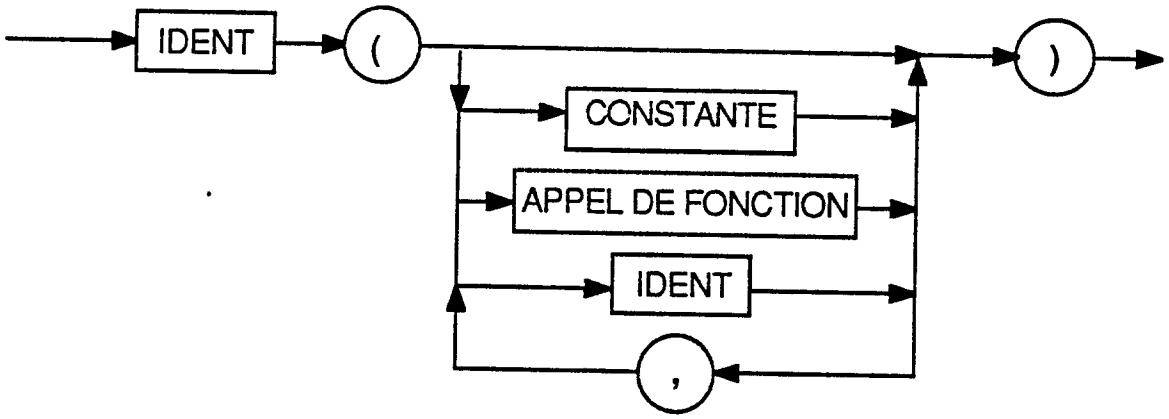
EXPRESSION SIMPLE:



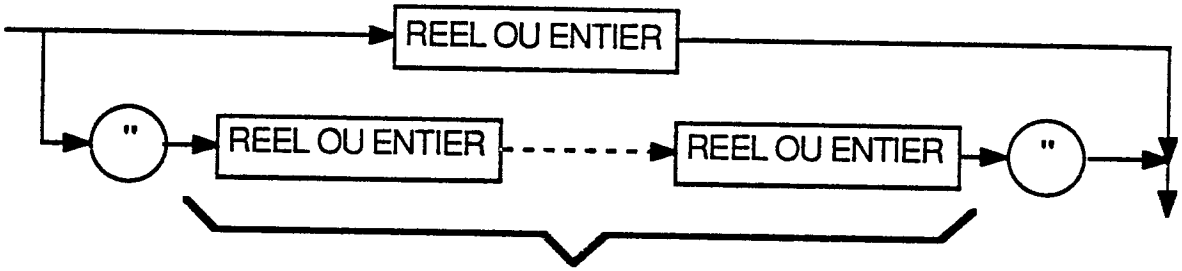
SELECTEURS:



APPEL DE FONCTION:

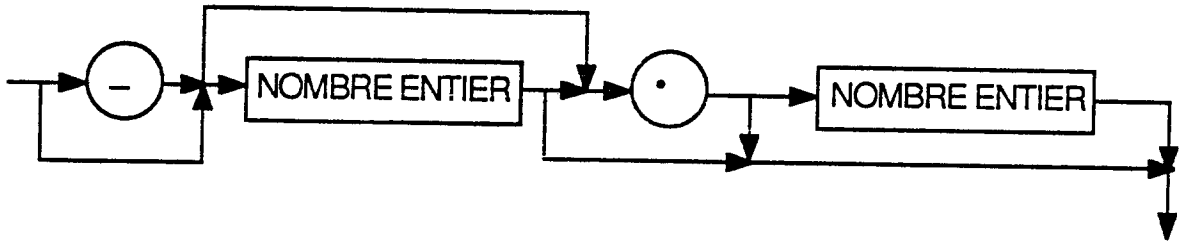


CONSTANTE:

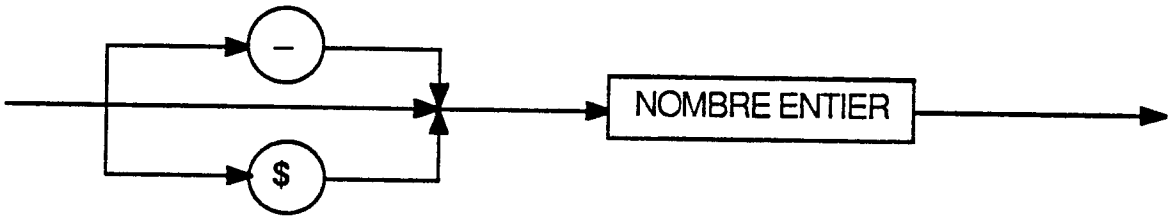


6 fois

REEL OU ENTIER:

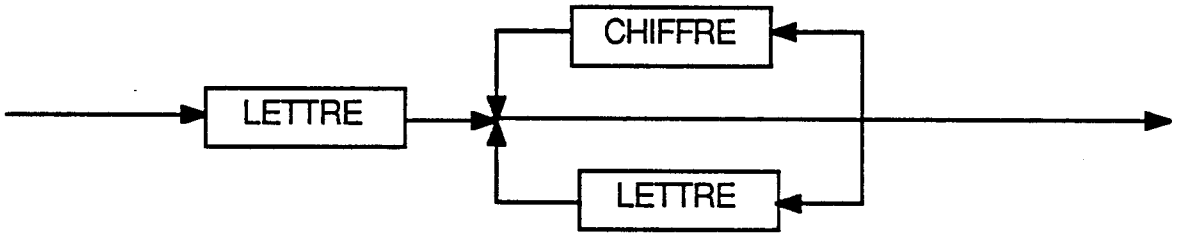


NOMBRE:

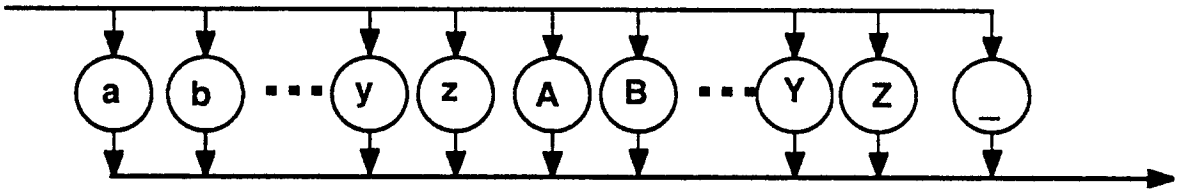


NOMBRE ENTIER:

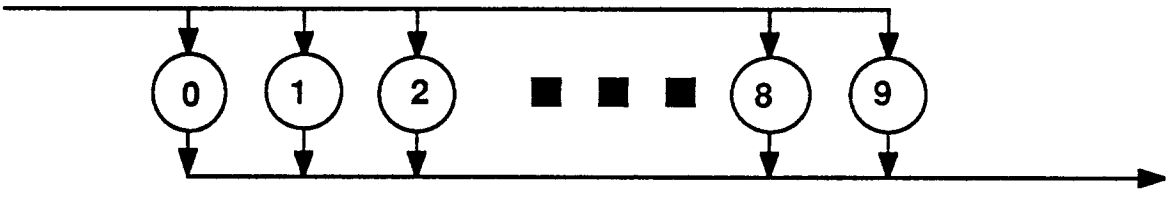
IDENT:



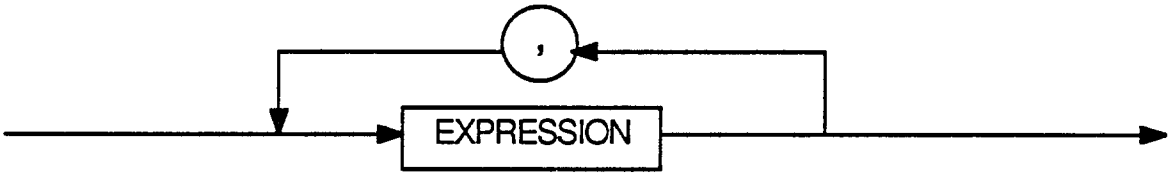
LETTRE:



CHIFFRE:



PARAMETRES:



ANNEXE F
BIBLIOGRAPHIE

REFERENCES BIBLIOGRAPHIQUES

[BACQ 87] Pierre BACQUET, Hervé POILVE et Alain ROBERT
Manuel d'utilisation de LIPS
AICO

[BRAM 83] G.W. BRAMS
Les réseaux de Petri
MASSON

[COUR 80] Marc COURVOISIER, Robert VALETTE
SYSTEMES DE COMMANDE EN TEMPS REEL
Description, Analyse et Réalisation
SCM

[GALL 77] Hervé GALLAIRE
TECHNIQUES DE COMPILATION
Méthodes d'Analyses syntaxiques
Edition CEPADUES / Collection Sup' Aéro

[JOHN 78] Stephen C. JOHNSON
Yet Another Compiler-Compiler
Bell Laboratories.

[KERN 78] Brian W. KERNINGHAM et Dennis M. RITCHIE
The C programming language
PRENTICE-HALL

[KRUT 79] KRUTCHEN
Le langage PASCAL
Eyrolles

[LESK] M.E. LESK & E. SCHMIDT
A Lexical Analyser Generator
Bell Laboratories.

[MEIN 71] Jean-Pierre MEINADIER
Structure et fonctionnement des ordinateurs
Larousse

[ROBE 87] Alain ROBERT
Thèse de Docteur Ingénieur
E.N.S.A.E

[SYRE 83] Jean-Claude SYRE, D. COMTE
Les supercalculateurs et leurs applications
Cours Sup'Aéro

BIBLIOGRAPHIE

Hamid DEMMOU
Sur la mise en œuvre directe d'une spécification fonctionnelle des systèmes temps réel
Thèse de Docteur de 3e cycle.
Université Paul Sabatier Toulouse

Eric THURIOT
Le Synchroniseur coloré: une approche pour la mise en œuvre des systèmes multi-tâches.
Thèse de Docteur Ingénieur
Université Paul Sabatier Toulouse