N° d'ordre : 763

50376
1979
15

THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNIQUES DE LILLE

pour obtenir le titre de

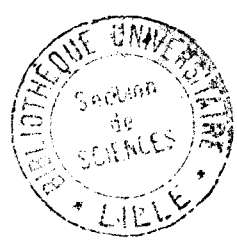
DOCTEUR DE 3ème CYCLE

(INFORMATIQUE)

par

Dimitri PLEMENOS

ETUDE ET RÉALISATION D'UN SYSTÈME DE CONCEPTION ASSISTÉE PAR ORDINATEUR APPLIQUÉ À LA MICRO-INFORMATIQUE



Thèse soutenue le 11 juin 1979, devant la Commission d'Examen

MEMBRES DU JURY	M.P. POUZET	Président
	M.V. CORDONNIER	Rapporteur
	M.C. CARREZ	Examinateur
	Mme D. BORRIONE	Examinateur

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE

01.02.1976

DOYENS HONORAIRES de l'Ancienne Faculté des Sciences

MM. R. DEFRETIN, H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES des Anciennes Facultés de Droit et Sciences Economiques, des Sciences et des Lettres

M. ARNOULT, Mme BEAUJEU, MM. BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, CORSIN, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, P. GERMAIN, HEIM DE BALSAC, HOCQUETTE, KAMPE DE FERIET, KOUGANOFF, LAMOTTE, LASSERRE, LELONG, Mme LELONG, MM. LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, NORMANT, PEREZ, ROIG, ROSEAU, ROUBINE, ROUELLE, SAVART, WATERLOT, WIEMAN, ZAMANSKI.

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

MM. R. DEFRETIN, M. PARREAU.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

M. J. LOMBARD.

PROFESSEURS TITULAIRES

M.	BACCHUS Pierre	Astronomie
M.	BEAUFILS Jean-Pierre	Chimie Physique
M.	BECART Maurice	Physique Atomique et Moléculaire
M.	BILLARD Jean	Physique du Solide
M.	BIAYS Pierre	Géographie
M.	BONNEMAN Pierre	Chimie Appliquée
M.	BONNOT Ernest	Biologie Végétale
M.	BONTE Antoine	Géologie Appliquée
M.	BOUGHON Pierre	Algèbre
M.	BOURIQUET Robert	Biologie Végétale
M.	CELET Paul	Géologie Générale
M.	CONSTANT Eugène	Electronique
M.	DECUYPER Marcel	Géométrie
M.	DELATTRE Charles	Géologie Générale
M.	DELHAYE Michel	Chimie Physique
M.	DERCOURT Michel	Géologie Générale
M.	DURCHON Maurice	Biologie Expérimentale
M.	FAURE Robert	Mécanique
M.	FOURET René	Physique du Solide
M.	GABILLARD Robert	Electronique
M.	GLACET Charles	Chimie Organique
M.	GONTIER Gérard	Mécanique
M.	GRUSON Laurent	Algèbre
M.	GUILLAUME Jean	Microbiologie
M.	HEUBEL Joseph	Chimie Minérale
M.	LABLACHE-COMBIER Alain	Chimie Organique
M.	LANSRAUX Guy	Physique Atomique et Moléculaire
M.	LAVEINE Jean-Pierre	Paléontologie
M.	LEBRUN André	Electronique
M.	LEHMANN Daniel	Géométrie

.../...

Mme	LENOBLE Jacqueline	Physique Atomique et Moléculaire
M.	LINDER Robert	Biologie et Physiologie Végétales
M.	LOMBARD Jacques	Sociologie
M.	LOUCHEUX Claude	Chimie Physique
M.	LUCQUIN Michel	Chimie Physique
M.	MAILLET Pierre	Sciences Economiques
M.	MONTARIOL Frédéric	Chimie Appliquée
M.	MONTREUIL Jean	Biochimie
M.	PARREAU Michel	Analyse
M.	POUZET Pierre	Analyse Numérique
M.	PROUVOST Jean	Minéralogie
M.	SALMER Georges	Electronique
M.	SCHILTZ René	Physique Atomique et Moléculaire
Mme	SCHWARTZ Marie-Hélène	Géométrie
M.	SEGUIER Guy	Electrotechnique
M.	TILLIEU Jacques	Physique Théorique
M.	TRIDOT Gabriel	Chimie Appliquée
M.	VIDAL Pierre	Automatique
M.	VIVIER Emile	Biologie Cellulaire
M.	WERTHEIMER Raymond	Physique Atomique et Moléculaire
M.	ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS SANS CHAIRE

M.	BELLET Jean	Physique Atomique et Moléculaire
M.	BODARD Marcel	Biologie Végétale
M.	BOILLET Pierre	Physique Atomique et Moléculaire
M.	BOILLY Bénoni	Biologie Animale
M.	BRIDOUX Michel	Chimie Physique
M.	CAPURON Alfred	Biologie Animale
M.	CORTOIS Jean	Physique Nucléaire et Corpusculaire
M.	DEBOURSE Jean-Pierre	Gestion des entreprises
M.	DEPREZ Gilbert	Physique Théorique
M.	DEVRAINNE Pierre	Chimie Minérale
M.	GOUDMAND Pierre	Chimie Physique
M.	GUILBAULT Pierre	Physiologie Animale
M.	LACOSTE Louis	Biologie Végétale
Mme	LEHMANN Josiane	Analyse
M.	LENTACKER Firmin	Géographie
M.	LOUAGE Francis	Electronique
Mle	MARQUET Simone	Probabilités
M.	MIGEON Michel	Chimie Physique
M.	MONTEL Marc	Physique du Solide
M.	PANET Marius	Electrotechnique
M.	RACZY Ladislas	Electronique
M.	ROUSSEAU Jean-Paul	Physiologie Animale
M.	SLIWA Henri	Chimie Organique

MAITRES DE CONFERENCES (et chargés d'Enseignement)

M.	ADAM Michel	Sciences Economiques
M.	ANTOINE Philippe	Analyse
M.	BART André	Biologie Animale
M.	BEGUIN Paul	Mécanique
M.	BKOUCHE Rudolphe	Algèbre
M.	BONNELLE Jean-Pierre	Chimie
M.	BONNEMAIN Jean-Louis	Biologie Végétale
M.	BOSCQ Denis	Probabilités
M.	BREZINSKI Claude	Analyse Numérique
M.	BRUYELLE Pierre	Géographie

.../...

M. CARREZ Christian	Informatique
M. CORDONNIER Vincent	Informatique
M. COQUERY Jean-Marie	Psycho-Physiologie
Mle DACHARRY Monique	Géographie
M. DEBENEST Jean	Sciences Economiques
M. DEBRABANT Pierre	Géologie Appliquée
M. DE PARIS Jean-Claude	Mathématiques
M. DHAINAUT André	Biologie Animale
M. DELAUNAY Jean-Claude	Sciences Economiques
M. DERIEUX Jean-Claude	Microbiologie
M. DOUKHAN Jean-Claude	Physique du Solide
M. DUBOIS Henri	Physique
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
Me EVRARD Micheline	Chimie Appliquée
M. FONTAINE Jacques-Marie	Electronique
M. FOURNET Bernard	Biochimie
M. FORELICH Daniel	Chimie Physique
M. GAMBLIN André	Géographie
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GUILLAUME Henri	Sciences Economiques
M. HECTOR Joseph	Géométrie
M. JACOB Gérard	Informatique
M. JOURNEL Gérard	Physique Atomique et Moléculaire
Mle KOSMAN Yvette	Géométrie
M. KREMBEL Jean	Biochimie
M. LAURENT François	Automatique
Mle LEGRAND Denise	Algèbre
Mle LEGRAND Solange	Algèbre
M. LEROY Jean-Marie	Chimie Appliquée
M. LEROY Yves	Electronique
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique Théorique
M. LOUCHET Pierre	Sciences de l'Education
M. MACKE Bruno	Physique
M. MAHIEU Jean-Marie	Physique Atomique et Moléculaire
Me N'GUYEN VAN CHI Régine	Géographie
M. MAIZIERES Christian	Automatique
M. MALAUSSENA Jean-Louis	Sciences Economiques
M. MESSELYN Jean	Physique Atomique et Moléculaire
M. MONTUELLE Bernard	Biologique Appliquée
M. NICOLE Jacques	Chimie Appliquée
M. PAQUET Jacques	Géologie Générale
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie Physique
M. PERROT Pierre	Chimie Appliquée
M. PERTUZON Emile	Physiologie Animale
M. PONSOLLE Louis	Chimie Physique
M. POVY Lucien	Automatique
M. RICHARD Alain	Biologie
M. ROGALSKI Marc	Analyse
M. ROY Jean-Claude	Psycho-Physiologie
M. SIMON Michel	Sociologie
M. SOMME Jean	Géographie
Mle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. STEEN Jean-Pierre	Informatique

.../...

M.	THERY Pierre	Electronique
M.	TOULOTTE Jean-Marc	Automatique
M.	TREANTON Jean-René	Sociologie
M.	VANDORPE Bernard	Chimie Minérale
M.	VILLETTE Michel	Mécanique
M.	WALLART Francis	Chimie
M.	WERNIER Georges	Informatique
M.	WATERLOT Michel	Géologie Générale
Mme	ZINN-JUSTIN Nicole	Algèbre

Je tiens à remercier :

Monsieur P. POUZET, Professeur à l'INSTITUT UNIVERSITAIRE DE TECHNOLOGIE de l'UNIVERSITE DE LILLE-I, qui m'a fait l'honneur de présider le Jury,

Monsieur V. CORDONNIER, Professeur à l'UNIVERSITE DE LILLE-I, qui m'a accueilli dans son équipe et, par ses conseils, m'a aidé à mener à bien ce travail,

Monsieur C. CARREZ, Professeur à l'UNIVERSITE DE LILLE-I, et Madame D. BORRIONE, C.N.R.S. GRENOBLE, qui ont accepté de faire partie du Jury et qui, par leurs critiques pertinentes et constructives, ont activement participé à l'amélioration de cet ouvrage.

Je tiens également à remercier

Mes collègues de l'I.U.T. qui, par leur solidarité et leur compréhension, ont grandement facilité mon travail,

Je tiens enfin à remercier toutes les personnes qui ont contribué à la réalisation matérielle de la thèse et, en particulier, Madame J. DESCARPENTRIES qui, malgré son emploi du temps surchargé, a assuré, avec sa rapidité et son perfectionnisme habituels, la dactylographie du manuscrit,

Madame H. DEBOCK qui, avec beaucoup de gentillesse, a accepté d'assurer les travaux de tirage dans des délais très rapides,

Mademoiselle M. DRIESSENS qui s'est occupé, avec efficacité, de toutes les démarches administratives.

à CHOO et NATACHA,

à mes parents.

TABLE DES MATIERES

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

Le concepteur de systèmes informatiques avait, jusqu'à un passé récent, des problèmes de langage et de méthodologie car il ne disposait d'aucun moyen pour exprimer clairement et de manière concise la logique d'un ordinateur. Les seuls moyens dont il disposait, à savoir les diagrammes logiques et l'algèble de Boole, ne permettaient que des descriptions trop détaillées pour être communiquées d'une part, et pour servir d'outil de conception, d'autre part.

YAOHAN CHU [16] compare cette situation à celle existant naguère en programmation, où les programmes, écrits en langage-machine ou en langage d'assemblage, étaient trop détaillés pour être compris, ce qui a donné naissance aux différents langages de programmation de haut niveau.

Ce besoin de langage de conception, ou d'aide à la conception, a donné lieu, depuis une quinzaine d'années, à des recherches ayant abouti à la définition d'un certain nombre de langages de description de matériel informatique. Ces langages permettent des descriptions, à divers niveaux, des composants d'un ordinateur. Ce choix de niveau de description est nécessaire compte tenu de la complexité du problème général de description d'un ordinateur complet à partir de ses composants les plus élémentaires.

Un ordinateur comporte de 10.000 à 1.000.000 de circuits logiques de base, sans compter les bascules qui composent sa mémoire. La description la plus fine, en ignorant le niveau technologique, de chacun de ces composants, nécessite un certain nombre d'éléments de définition, tels que

- Entrées
- Sorties
- Fonctions booléennes
- etc...

ce qui rend pratiquement impossible, et, en tout cas, illisible, la description d'un ordinateur complet, même très simple.

Pour éviter cela, on doit autoriser des regroupements de cellules binaires pour former des registres, des unités arithmétiques et logiques, des mémoires, etc... Ces regroupements doivent se faire suivant un certain nombre de critères. En dehors des regroupements itératifs (par exemple n fois bascule = registre, n fois registre = mémoire) qui sont évidents, on doit, par exemple :

- choisir les niveaux qui font perdre le moins d'information sur les cellules,
- choisir des niveaux homogènes. Ainsi, une unité centrale et un registre sont des composants hétérogènes et ne doivent pas figurer au même niveau,
- éventuellement, autoriser l'outil de description à faire lui-même des regroupements.

La figure INT-1 représente un exemple des différents niveaux de regroupement possibles dans une description d'ordinateur.

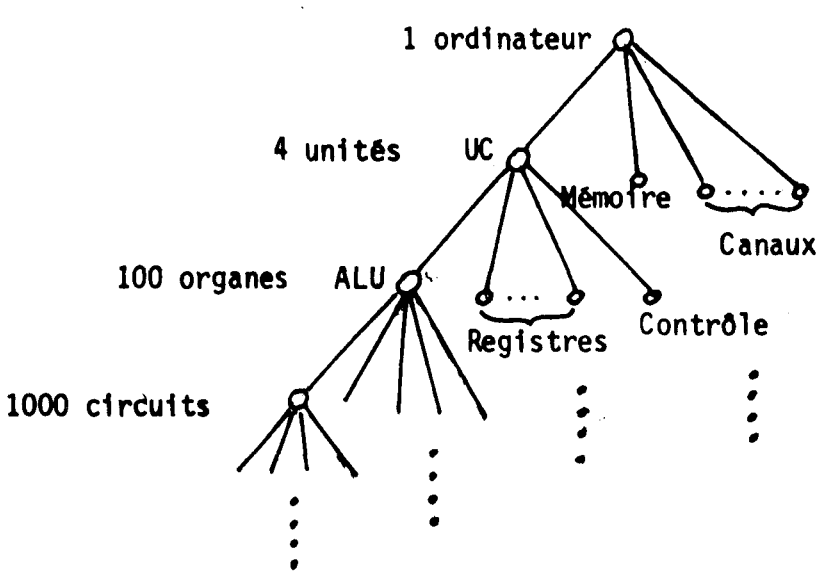


figure INT-I

Le choix d'une description peut se faire sur 1 ou 2 niveaux de synthèse et permet, à partir de modules du niveau immédiatement inférieur, de construire les modules pour le niveau immédiatement supérieur. Ainsi, à partir des modules ALU , accumulateur, registres, etc..., on construit les modules UC , mémoire, etc... Comme nous l'avons déjà dit, il serait illusoire de tenter un passage direct des feuilles à la racine de l'arbre. C'est la raison pour laquelle on s'oriente de plus en plus, en matière de langages de description, vers des langages modulaires et hiérarchisés qui permettent, par leur conception structurée, d'exprimer les différents niveaux de synthèse.

Le but de ce travail est de proposer un système de description répondant à des objectifs précis. Mais, avant d'entreprendre la définition de notre système, nous allons procéder à une analyse de l'existant. Ainsi, la présente thèse, est-elle composée de 4 chapitres dont nous présentons ici brièvement le contenu.

Dans le chapitre I, nous faisons une présentation succincte de quelques uns des langages de description existant actuellement, suivie d'une étude critique faisant ressortir les avantages et les limites de ces langages.

Le chapitre II est consacré à la définition du système proposé. Cette définition est précédée d'un cahier des charges définissant les objectifs à atteindre.

Dans le chapitre III, nous présentons l'un des composants du système, le langage de description LEDA. Il s'agit, dans un premier temps, d'une présentation informelle basée sur des exemples qui permettent de bien voir l'utilisation du langage dans des cas précis. Ce n'est qu'après cette première approche que nous présentons la définition complète du langage.

Les problèmes d'implémentation du système sont présentés au chapitre IV,

en particulier l'interpréteur LEDA et les bibliothèques. On y présente également les primitives de manipulation des objets du système qui forment un langage de commande, ainsi que le processus de simulation.

C H A P I T R E I

-=-=-=-=-=-=-=-=-=-=-

LES LANGAGES DE DESCRIPTION

CHAPITRE I

LES LANGAGES DE DESCRIPTION

Nous avons vu, dans l'introduction, que les langages de description de matériel sont nés du besoin d'un outil (ou d'un ensemble d'outils). Ce besoin concerne :

1) LE CONCEPTEUR, qui souhaite pouvoir décrire son système de manière lisible et, si possible, pouvoir par la suite vérifier la conformité du système décrit, soit par modélisation, soit par simulation...

2) L'UTILISATEUR, qui doit trouver dans cet outil un moyen de connaître la configuration proposée et de tester ses propres choix :

   - matériel complémentaire
   - système
   - micro-programmation
   - programmes d'application.

Pour que l'outil souhaité par les concepteurs et les utilisateurs soit universel, il doit se prêter à la description aussi bien de petits systèmes à base de micro-processeurs que de gros systèmes à base d'ordinateurs les plus sophistiqués. Il doit, par ailleurs, permettre à l'utilisateur de choisir le niveau de description de son système et de descendre jusqu'à un niveau de description très fin, sans nuire pour autant à la clarté de description de l'ensemble. Il doit, enfin, permettre la description aussi bien du fonctionnement que de la structure physique et logique du système. Il serait d'autre part souhaitable que cet outil puisse déboucher sur une simulation permettant soit de vérifier la conformité fonctionnelle du système décrit, soit d'en évaluer les performances.

A notre connaissance, il n'existe pas, à l'heure actuelle, d'outil vérifiant tous les critères que nous venons d'énoncer. Nous pensons d'ailleurs qu'il serait illusoire de croire qu'un tel outil puisse exister un jour car la recherche de la finesse dans la description se fait toujours au détriment de la clarté. Il y a donc un compromis à établir mais, dans la mesure du possible, le choix de l'équilibre entre les deux aspects doit être fait par l'utilisateur de l'outil et non par celui qui le conçoit.

Les langages de description de matériel apportent des réponses partielles au problème de la description des ordinateurs, suivant des critères qui diffèrent selon le langage. Pour les uns, le critère qui prévaut est celui de la clarté de description, pour les autres, celui de la finesse. Un autre critère, dont les langages de description tiennent de plus en plus compte, est le critère de modularité.

L'objectif d'une partie de ces langages se limite à la définition d'un outil de description (algorithmique ou non suivant le niveau) et le traitement sur ordinateur de cette description n'y est pas prévu. Ces langages ne sont donc pas compilables et leur définition syntaxique n'est pas nécessairement rigoureuse. Les premières versions de I.S.P. et P.M.S. constituent des exemples types de cette famille de langages.

Pour les autres, la description doit déboucher sur une simulation permettant de tester le système décrit. Cette famille comporte des langages tels que CDL, CASSANDRE et LASCAR définis pour des niveaux divers de description, ainsi que les versions compilables de I.S.P. et P.M.S.. On pourrait, enfin, mentionner des langages de simulation tels que SIMULA ou GPSS qui couvrent en fait un domaine beaucoup plus vaste que la simulation de systèmes informatiques, ainsi que les langages de programmation les plus évolués tels que PL/I (voir à ce propos une tentative de description d'un SIEMENS 4004 en PL/I [9]), ALGOL 68 et, surtout APL.

Dans les pages qui suivent, nous allons essayer de faire une présentation succincte de quelques langages représentatifs de description et une analyse comparative de leurs caractéristiques essentielles.

I.1.- LE LANGAGE ISP.

Le langage ISP (Instruction Set Processor) [8] a été conçu pour décrire surtout le niveau de langage-machine d'un ordinateur.

La description d'un ordinateur en ISP se fait par la présentation de ses composants matériels et logiciels : Registres, mémoire, moyens de communication, jeu d'instructions, actions des instructions sur les données. Cette présentation se fait, habituellement, dans un ordre fixe.

Notre intention n'est pas de faire ici une présentation complète du langage mais plutôt d'en donner un aperçu. Ainsi, nous présenterons brièvement les différentes parties de la description.

On peut distinguer deux parties dans une description ISP :

1) Une partie statique comportant :

   a) Description de la mémoire et de l'unité centrale.

      Exemple :

      $M \subset [0:2^k] < 0:7 >$

      Mémoire de 2^k mots de 8 bits. Les adresses des mots sont comprises entre 0 et 2^k-1 . Les bits à l'intérieur des mots sont numérotés de 0 à 7 .

      La description de l'unité centrale se fait par la présentation de tous ses registres.

Exemple :

R[0:3]<0:7>

4 registres de 8 bits

On a la possibilité de donner un nom à un registre ou à une partie de registre.

Exemple :

CO := R[3]

On utilise le nom CO pour désigner le registre n° 3 de l'exemple précédent.

b) Description du format des instructions

Cette description se fait par assimilation au registre qui contient les instructions.

Exemple :

INST<0:15>	Instruction de 16 bits
OP:=INST<0:7>	Code opération
ADA:=INST<8:15>	Adresse absolue

c) Description des types des données

Exemple :

WBV<0:15>
Vecteur (V) Booléen (B) sur un mot (W)

d) Description des opérations sur les types des données

Exemple :

A ← A+B{f}
Addition en virgule flottante.

2) Une partie dynamique comportant :

a) Description de la fonction d'adressage.

Exemple :

AD<0:7>:=(¬ BAI → AD';BAI → M[AD'])

Si le bit d'adressage est à 0 , l'adresse de l'opérande est la valeur de AD ; s'il est à 1 , l'adresse de l'opérande est la valeur contenue dans le mot de mémoire d'adresse AD.

b) Description du processus d'enchaînement et d'interprétation des instructions

Exemple :

```
RUN ∧ ¬(IRQ ∧ EI) → (RI ← M[PC];
                     PC ← PC+1;next
                     EXECUTION);
RUN ∧ IRQ ∧ EI → (M[0] ← PC;EI ← 0;PC ← 1)
```

Si la machine est dans l'état de marche et qu'il n'y a pas de demande d'interruption lorsqu'elle se trouve dans l'état interruptible, l'instruction en mémoire est placée dans le registre d'instruction, le compteur ordinal est incrémenté de 1 et l'instruction est exécutée. Si la machine est dans l'état de marche et qu'il y a une demande d'interruption alors qu'elle se trouve dans l'état interruptible, l'adresse de l'instruction est placée en mémoire 0, la machine passe à l'état non interruptible et l'adresse de traitement de l'interruption est placée dans le compteur ordinal.

c) Description du jeu d'instructions de la machine décrite et du processus d'exécution.

.../...

Exemple :

DCA(:=OP=3) → (M[Z] ← AC;AC ← 0);

Rangement de l'accumulateur en mémoire et remise de l'accumulateur à zéro. Les deux instructions de la description ISP M[Z] ← AC et AC ← 0 peuvent s'exécuter en parallèle, ce qui est indiqué par le séparateur ";" .
Si l'on veut indiquer que deux instructions doivent s'exécuter séquentiellement, on fait suivre le séparateur ";" du mot next.

Exemple :

JMS(:=OP=4) → (M[Z] ← PC;next PC ← Z+1);

Appel du sous-programme d'adresse Z . Le compteur ordinal est placé dans le mot d'adresse Z (adresse de retour), puis, il est chargé de l'adresse Z+1.

Il est à noter que la description d'une instruction de la machine décrite comporte 2 parties :

1.- la partie codage : Assimilation à des registres formatés,

2.- la partie action : transferts et tests.

ISP n'est pas un langage au sens strict du terme dans la mesure où sa définition n'est pas suffisamment précise et rigoureuse pour permettre une exploitation par ordinateur (interprétation, compilation) de l'ensemble décrit. En effet, l'utilisateur peut ajouter, à sa guise, des éléments nouveaux au langage. Il n'en reste pas moins que la notation ISP permet des descriptions claires et concises de n'importe quel ordinateur ce qui en fait non seulement un outil au service du concepteur mais aussi un outil pédagogique fort intéressant. Le manque de rigueur de sa définition n'est pas forcément un inconvénient dans la mesure où il permet des descriptions très souples. Il apparaît souvent que l'introduction de la rigueur dans la définition d'un langage ne peut se faire qu'au

détriment de la souplesse et de la clarté de description.

Notons, cependant, qu'un sous-ensemble du langage a donné lieu à l'écriture d'un compilateur à l'Université de Carnegie-Mellon, aux Etats-Unis [2]. La dernière version de ce compilateur, nommée ISPS, étend la notation ISP d'origine en y introduisant certaines constructions classiques des langages de programmation telles que la structure de blocs, les actions conditionnelles (IF,DECODE), les actions répétitives et les appels de sous-programmes.

I.2.- LE LANGAGE PMS

Le langage PMS (Processors, Memories, Switches) [8] a été conçu pour décrire la structure physique d'un ordinateur ou d'un réseau d'ordinateurs. Pour ce faire, PMS utilise un petit nombre de composants élémentaires définis d'après la fonction qu'ils réalisent. Ces composants sont les suivants :

- LA MEMOIRE M est un composant qui contient et stocke de l'information. Ce composant réalise essentiellement deux opérations :
  lecture d'unités d'information se trouvant en mémoire et
  écriture d'unités d'information en mémoire.
  Une mémoire pouvant contenir plus d'une unité d'information est dotée d'un système d'adressage permettant la sélection des unités d'information. Une mémoire peut être considérée comme un aiguillage vers un ensemble de mémoires composantes, ce qui permet une définition récursive de la mémoire.

- LE LIEN L est un composant qui sert à transférer l'information d'un endroit à un autre de l'ordinateur. Ses extrémités sont fixes. Sa fonction élémentaire est le transfert d'une ou plusieurs unités d'information du composant relié à l'une des extrémités, au composant relié à l'autre.

- LE CONTROLE K est un composant qui commande les opérations d'autres composants du système. En effet, en dehors du processeur P (voir plus loin), tous les autres composants d'un système sont supposés passifs et ont besoin d'être activés par le contrôle K pour réaliser leur fonction.

- LE COMMUTATEUR S est un composant qui établit dynamiquement des liens entre composants. Chaque commutateur est associé à un ensemble de liens possibles et sa fonction consiste à établir des liens et à en supprimer d'autres.

- LE TRADUCTEUR T est un composant qui transforme le codage d'une information en entrée et en codage de sortie. Cette opération conserve la signification de l'information qu'elle traite.

- L'OPERATEUR DE TRAITEMENT DES DONNEES D est un composant qui produit des unités d'information à partir de celles qui lui sont fournies en entrée en fonction des commandes lui précisant le type d'opération à effectuer. C'est le composant qui effectue toutes les opérations portant sur les données (arithmétiques, logiques, décalage, etc...)

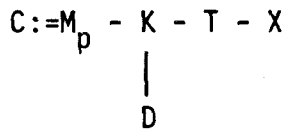
- LE PROCESSEUR P est le composant actif du système. Son rôle est d'interpréter un programme et d'exécuter une suite d'opérations. Ces opérations sont celles des types déjà mentionnés (M, L, K, S, T, D) plus celle qui permet d'obtenir des instructions placées en mémoire et de les interpréter comme opérations à exécuter.

Ainsi, une configuration classique d'ordinateur sera décrite en PMS par le schéma :

$$C := M_p - P_c - T - X$$

où M_p désigne la mémoire principale, P_c le processeur central, T le traducteur et X le milieu externe. Le symbole ":=" attribue le nom C à la configuration.

Si l'on désire un niveau de description plus fin, on peut décomposer le processeur central P_c en une unité de contrôle K et un opérateur de traitement des données D. On obtient ainsi :



ou bien :

où les lignes continues indiquent les liaisons de transfert des instructions et de leurs données et les lignes pointillées indiquent les liaisons de contrôle.

A chéque composant, on peut associer des spécifications supplémentaires permettant de mieux le définir. Ainsi, on peut écrire

$$A(a_1:v_1 \ ; \ a_2:v_2 \ ; \ \ldots \ ; \ a_n:v_n)$$

pour indiquer que le composant A est caractérisé par les attributs $a_1, a_2, \ldots, a_n$ ayant pour valeurs $v_1, v_2, \ldots, v_n$ respectivement.

Exemple :

M(fonction : primaire ; temps d'opération : 1,5 µs ;
taille : 4096 mots ; mot : (12+1)bits)

On voit que ces attributs s'expriment en langue naturelle, ce qui exclut tout emploi algorithmique.

Un des aspects les plus intéressants de PMS est la possibilité de décomposition d'un composant en d'autres composants, ce qui permet une définition récursive d'un composant. Ainsi, une mémoire peut être définie récursivement comme soit une mémoire, soit un aiguillage vers d'autres mémoires.

La description :

$$C := M - P - T - X$$

peut ainss être remplacée par la description :

$$C := \begin{matrix} M \\ M \\ \vdots \\ M \end{matrix} \Big| \; - S - P - T - X$$

et ainsi de suite, permettant une description de plus en plus fine de la mémoire.

$$M_p \left[\begin{array}{l} \text{core; 1 µs/w; 8192} \sim \text{32768 w;} \\ \text{4 by/w; (8+1 parity) b/by} \end{array} \right] - S_{tm} \left\{ \begin{array}{l} T \text{ console} \\ P_c \\ P_{io}^{\,1} - St_m - K(\# \; 0{:}63) - \\ P_{io}^{\,2} - S_{fx} - K(\# \; 0{:}1) - \\ P_{io}^{\,2} - S_{fx} - K(\# \; 0{:}1) - \end{array} \right.$$

$^1P_{io}$ (multiplexor Channel)

$^2P_{io}$ (# 1:4; High Speed Multiplexor Channel)

FIGURE I.1

DESCRIPTION EN PMS DE LA STRUCTURE D'UN ORDINATEUR I.B.M. 360; modèle 44. Cette description figure dans [8]:

Grâce à son petit nombre de composants de base et à la possibilité de définitions récursives, PMS permet de décrire la structure d'un ordinateur ou d'un réseau d'ordinateurs à des niveaus de détail variés. On peut regretter que l'emploi de PMS ne soit pas généralisé car ses descriptions sont concises et faciles à comprendre même par des non-spécialistes et pourraient être utilisées dans les notices de constructeurs afin de donner une vue globale du matériel proposé.

La généralisation de l'emploi de PMS permet d'une part d'expliquer, sur un schéma simple, la structure et le fonctionnement des ordinateurs, d'autre part de comparer deux ou plusieurs systèmes décrits en PMS.

Du fait de sa présentation semi-graphique, PMS n'est pas un langage au sens informatique du terme et n'a pas été conçu pour simuler le fonctionnement des systèmes décrits. Il semble cependant qu'un compilateur d'un sous-ensemble de PMS existe actuellement à l'Université de Carnegie-Mellon.

I.3.- LES LANGAGES CASSANDRE ET LASCAR.

Le langage CASSANDRE est un langage de description et de simulation de systèmes logiques, ce qui est une notion plus générale que celle de machines digitales. Il s'agit d'un langage évolué permettant des descriptions modulaires à un niveau fin (niveau des circuits séquentiels et combinatoires).

Le module de base de CASSANDRE est l'unité qui correspond à la notion de "boîte noire" et a les propriétés suivantes :

- chaque unité peut contenir un réseau d'unités interconnectées
- il existe une unité qui contient toutes les autres.
- une description en CASSANDRE peut être considérée comme un arbre dont chaque noeud est un réseau de boîtes.

Un programme en CASSANDRE est la description d'une unité et contient principalement deux parties de natures différentes :

1 - Une partie structurelle comprenant :
    - l'en-tête de l'unité avec ses entrées et sorties
    - des déclarations propres à l'unité
    - des connexions permanentes et inconditionnelles

2 - Une partie fonctionnelle qui est la description des fonctions logiques que l'unité est censée effectuer, description qui n'implique aucune réalisation particulière. Réaliser une unité, c'est remplacer la partie fonctionnelle par un réseau, donc une liste de connexions permanentes et inconditionnelles, donc une partie structurelle [32] .

L'en-tête de l'unité décrit son interface avec l'extérieur. Les déclarations, qui suivent immédiatement l'en-tête, concernent des variables associées au matériel employé et utilisées dans la description de l'unité.

La description d'une unité peut faire appel à des variables de 6 types différents :

1 - Eléments de mémorisation (Registres, Mémoires, Bascules, ...)
2 - Eléments booléens de connexion (fils, bornes, circuits, combinatoires,...)
3 - Impulsions de synchronisation (horloges, signaux dérivés,...)
4 - Variables d'état automates
5 - Autres unités CASSANDRE
6 - Variables arithmétiques utilisées pour décrire des algorithmes.

L'exemple suivant, pris dans la thèse de J. MERMET [32] , illustre la manière d'écrire une description en CASSANDRE et l'utilisation de notions vues dans ce paragraphe.

```
'UNITE'DECG(OP1(1:4),CO1(1:2) ; SG(1:4));
   'SIGNAL' OP(0:6),C(0:3);
   'EXTERNE' VALN2(2,4);
             VALN2(CO1,C);
             OP:=OP1 & 000 ;
          'POUR' I=0 'A' 3
            'DEBUT'
               'SI' C(1) 'ALORS'
                  SG:=OP(1:3+I);
               'FIN';
```

L'unité DECG décrite ici a deux entrées, OP1 et CO1 de quatre et deux positions binaires respectivement et une sortie SG de quatre positions binaires.

Suivant les valeurs des entrées OP1 et CO1 et des signaux C , l'unité calcule la valeur de la sortie SG . La valeur du signal C est fournie par l'unité VALN2.

Le mot-clé 'EXTERNE' précède la déclaration de toutes les unités utilisées dans une unité donnée. Ainsi la ligne

```
'EXTERNE' VALN2(2,4)
```

indique que VALN2 est une unité externe utilisée dans DECG , et la ligne :

```
VALN2(CO1,C),
```

indique que l'unité VALN2 doit être activée avec CO1 comme entrée et C comme sortie.

La partie fonctionnelle d'une unité est décrite en CASSANDRE par des instructions. Les instructions de CASSANDRE sont de deux types :

- Les instructions de branchement,
- Les instructions d'affectation.

Ces instructions peuvent être :

1 - Des instructions toujours vraies, c'est-à-dire exécutées inconditionnellement.

Elles figurent, dans une description d'unité, immédiatement après les déclarations.

2 - Des instructions sous état.

Ces instructions sont toujours étiquetées. Une instruction sous état n'est exécutée que si la variable d'état interne

de l'unité a pour valeur l'étiquette correspondant à cette instruction.

Toutes les instructions de CASSANDRE sont exécutables en parallèle et le résultat de leur exécution est indépendant de leur ordre lexicographique.

Le langage CASSANDRE et le système associé couvrent le domaine des réseaux de portes logiques et celui des ensembles logiques et microprogrammes. D'après leur auteur, ils ne peuvent pas s'étendre aux domaines du langage-machine et du système en raison du temps considérable que demanderait alors la simulation. Selon J. MERMET [32] , d'autres outils sont nécessaires pour couvrir ces deux derniers domaines.

C'est probablement pour occuper ce créneau que le langage et le système LASCAR ont été développés [10] . LASCAR est une extension de CASSANDRE qui conserve la structure de ce dernier et notamment sa modularité. Pour des raisons de performance à la simulation, y est introduite la notion de séquentialité qui n'existe pas dans CASSANDRE. D'autre part, des objets nouveaux ont été rajoutés à ceux manipulés par CASSANDRE et les opérateurs associés, ainsi que des instructions plus proches de celles des langages algorithmiques classiques.

Ces objets nouveaux sont :

- les entiers,
- les tableaux entiers,
- les compteurs,
- les procédures.

Les nouvelles instructions de LASCAR par rapport à CASSANDRE sont les suivantes :

- l'affectation d'entier
- l'appel de procédure

- l'instruction conditionnelle
- l'instruction POUR.

Le gros avantage des langages CASSANDRE et LASCAR est leur modularité. D'autre part, ils permettent la simulation du système décrit, donc la vérification de son fonctionnement. Ils se situent enfin à un niveau très fin de description, ce qui ne vas pas sans inconvénient au niveau de la lisibilité lorsqu'il s'agit de décrire un système complet.

I.4.- LE LANGAGE CDL.

Le langage CDL (Computer Design Language) [16] a été conçu pour décrire le niveau des micro-programmes d'un ordinateur et peut s'étendre à la description du niveau du langage-machine.

Une description en CDL est un ensemble de séquences. Une séquence est composée d'une suite de déclarations suivie d'une suite d'instructions utilisant des opérateurs et des expressions. Les objets manipulés par le langage sont :

1 - Les identificateurs,

2 - Les registres.
Le nom d'un registre est un identificateur qui doit être déclaré.

Exemple :

register REG1

3 - Les sous-registres.
Les sous-registres sont des parties de registres. Une déclaration de sous-registre permet de donner un nom à une partie de registre.

Exemple :

subregister R[ADR] = R[10-17]

4 - Les registres dispersés.
La concaténation de plusieurs registres ou sous-registres forme un registre dispersé.

Exemple :

R & REG1 & R[ADR] est un registre dispersé

5 - Les mémoires
Une mémoire est formée d'un ensemble de registres adressables d'un mot chacun. Elle doit être déclarée.

Exemple :

memory MEM[A] = MEM[0-117,0-27]

indique que la mémoire MEM est composée de 80 (120_8) mots de 24 bits et que A est son registre d'adressage.

6 - Les nombres et les valeurs.
Le système octal est adopté pour la représentation des nombres. Il existe deux valeurs qui sont de type booléen : true et false.

7 - Les variables de connexion (terminals)
Les variables de connexion sont des identificateurs spéciaux utilisés pour désigner des signaux.

Exemple :

terminal TERM1[1-10],TERM2[1]

Déclaration du groupe de 8 connexions TERM1 et de la connexion TERM2.

8 - Les constantes.
Des identificateurs peuvent désigner des constantes.

Exemple :

constant TROIS=3

CDL est un langage non procédural, ce qui veut dire que l'exécution des instructions d'une séquence dépend de la valeur de l'étiquette dont chaque instruction (ou groupe d'instructions) est précédée. Une instruction est exécutée si et seulement si l'étiquette correspondante prend la valeur true.

Les types d'instructions que l'on peut rencontrer dans une séquence sont les suivants :

1 - L'instruction de transfert.

Exemple :

A ← A add 1

Addition de 1 au contenu de A et transfert du résultat dans A .

2 - L'instruction de connexion.

Une instruction de connexion sert à définir l'état d'une connexion.

Exemple :

TERM1[3] = R[2] ∧ A[2] ∨ B[1]

L'expression booléenne en partie droite détermine l'état de la variable de connexion TERM1[3] .

3 - L'instruction d'échange.

Elle permet d'exprimer en une seule instruction un cas particulier de double transfert.

Exemple :

R ↔ MEM[A] exprime l'échange des valeurs de R et MEM[A].

4 - L'instruction conditionnelle.

5 - L'instruction FAIRE.

Cette instruction permet l'exécution d'une opération décrite quelque part dans le programme. Cette opération doit être déclarée.

Exemple :

opération COMPTEUR

⋮

R[1] ∧ T : do COMPTEUR

⋮

COMPTEUR : R1 ← R1+1

6 - L'instruction ALLER A.

C'est une instruction qui permet d'activer une des connexions d'un aiguillage.

Exemple :

switch AIG[TO,T1,T2]

⋮

go to AIG[2] Activation de la connexion T1

Notons enfin qu'une description en CDL peut comporter des commentaires.

Une description en CDL est composée d'une partie statique et d'une partie opératoire. La partie statique est composée de déclarations décrivant la structure de la mémoire et la structure de contrôle. La partie opératoire comporte un ensemble de micro-instructions précédées d'étiquettes de contrôle et décrit le fonctionnement du système et la partie dynamique de la structure de contrôle. La figure I-2 montre la description en CDL d'un système qui réalise la complémentation du chaque bit d'un registre de 5 bits. Cet exemple est pris dans [18] .

.../...

```
Comment        storage structure
Register       A(1-5),C(0-2)
Light          FINI(ON,OFF)
Switch         START(ON)
Comment        Control structure
Terminal       T(1-3),
Clock          P
Comment        Processor Structure
/START(ON)/    T ← 100₂,C ← 0,FINI ← OFF,
/T(1) ∧ P/     T ← 010₂,A ← A(1)' & A(1-4),C ← countup C,
/T(2) ∧ P/     IF(C=5) THEN(T ← 001₂) ELSE(T ← 100₂),
/T(3) ∧ P/     T ← 0, FINI ← ON,
               END
```

FIGURE I-2

CDL est intéressant pour son organisation non procédurale qui permet de séparer la structure de contrôle de la structure des données. Il permet des descriptions claires des microprogrammes mais il est moins adapté à la description du langage-machine. Le manque de modularité constitue un de ses principaux inconvénients.

I-5.- QUELQUES REMARQUES SUR LES LANGAGES DE DESCRIPTION.

Les langages de description que nous venons de voir sont loin d'être les seuls existants. Ils sont cependant assez représentatifs des différentes tendances en matière de langages de description.

On pourrait encore mentionner des langages tels que APDL qui est une extension d'ALGOL, AHPL [26] qui est une extention d'APL, le langage MAS proposé par M. ZAGHARIADES [44] et qui sépare complétement la partie contrôle de la partie opérative, APL * DS [24] qui est basé sur APL et bien d'autres encore, définis ces dernières années, notamment aux Etats-Unis.

Les langages que nous avons vus dans les paragraphes précédents ne visent pas tous le même niveau de description. Pour arriver à classer ces langages suivant les niveaux qu'ils décrivent, il nous faut voir auparavant les différents niveaux de description.

Il est généralement admis [1] que l'on peut distinguer 5 niveaux dans la description du matériel des systèmes informatiques :

1 - Le niveau système. C'est le niveau le plus général de description. Les objets traités à ce niveau sont des processeurs, des mémoires, des périphériques, etc...

2 - Le niveau langage-machine. Le cadre de travail est ici le processeur. Les objets traités sont les instructions de la machine qui sont interprétées suivant des règles de construction de la machine et qui agissent sur des états internes de celle-ci.

3 - Le niveau des micro-programmes. Les objets traités sont ici les registres. Le contenu d'un registre peut être transformé ou transféré dans un autre registre.

4 - Le niveau des réseaux de portes logiques. Les objets traités sont ici des bascules et les opérateurs booléens de base. Le Le fonctionnement du système est décrit par des équations booléennes.

5 - Le niveau des circuits électriques. C'est le niveau le plus fin de description et le plus proche de la réalité matérielle. C'est un niveau important qui se décrit par d'autres méthodes car les objets traités ici n'ont plus les propriétés discrètes qui caractérisent les autres niveaux. Cette description fait nécessairement appel à des lois physiques caractérisant la nature des supports mis en oeuvre.

Le langage PMS est le seul à couvrir entièrement le premier niveau. ISP couvre le niveau du langage-machine mais peut difficilement descendre au niveau des micro-programmes. CDL couvre le niveau des micro-

programmes et peut difficilement s'étendre au niveau du langage-machine. CASSANDRE couvre les niveaux des réseaux des portes logiques et des micro-programmes. LASCAR, enfin, couvre le niveau du langage-machine et, partiellement, celui du système. La figure I-3 résume la classification des différents langages de description suivant le niveau décrit.

NIVEAU / LANGAGE	1	2	3	4	5
PMS	O	N	N	N	N
ISP	N	O	N	N	N
CDL	N	P	O	N	N
CASSANDRE	N	N	O	O	N
LASCAR	P	O	N	N	N

O : OUI
N : NON
P : PARTIELLEMENT

FIGURE I-3

Il existe un autre critère de classification des langages de description : l'ordre d'exécution des actions décrites par le langage. Ce critère concerne les langages couvrant les niveaux 2, 3 ou 4 . On distingue ainsi deux types de langages :

1 - Les langages non procéduraux, où l'ordre d'exécution des actions est indépendant de leur place dans le programme. Cet ordre est déterminé par la valeur d'une expression

.../...

booléenne qui précède chaque action. Cette catégorie de langages est très proche des classes Data-Driven où l'exécution d'une instruction dépend uniquement de ses données. CASSANDRE et CDL sont des langages non procéduraux, alors que LASCAR l'est partiellement.

2 - Les langages procéduraux, où l'ordre d'exécution des actions est séquentiel. ISP est un langage procédural.

TYPE / LANGAGE	PROCEDURAL	NON PROCEDURAL
ISP	O	N
CASSANDRE	N	O
LASCAR	P	P
CDL	N	O

FIGURE I-4

Les langages de description existant sont-ils satisfaisants ? Quelles sont les propriétés que doit posséder un bon langage de description ? M. BARBACCI [1] propose les propriétés suivantes :

1) Lisibilité. Une description est destinée à être utilisée par des personnes qui, a priori, n'ont pas toutes la même expérience. Aussi, la description doit-elle être précise et concise. PMS est un langage très lisible. ISP aussi. CDL l'est également pour les descriptions du niveau des microprogrammes.

2) Familiarité (familiarity). Les objets du langages doivent être nommés et utilisés de manière proche de la réalité. Tous les langages vus au paragraphe précédent possèdent cette propriété.

3) Généralité. La description doit pouvoir se faire à des niveaux de détails différents. ISP, PMS, CASSANDRE et LASCAR sont des langages possédant cette propriété. Ceci est moins vrai pour CDL.

4) Simplicité. Le langage doit se contenter de peu de concepts de base généraux et les utiliser de manière conséquente dans la description. PMS est de loin le langage de description le plus simple. ISP et CDL le sont beaucoup moins, ainsi que CASSANDRE et LASCAR. On peut dire d'une manière générale qu'aucun langage de description couvrant les niveaux 2n 3 ou 4 n'est vraiment simple.

5) Extensibilité. Le langage doit pouvoir s'élargir facilement par la définition de nouveaux concepts à partir de ceux déjà existant dans le langage. Aucun des langages existant actuellement n'est vraiment extensible, d'où la nécessité, soit d'en définir un, soit de disposer d'un ensemble cohérent de langages permettant d'exprimer l'enchaînement hiérarchisé des niveaux de modélisation.

6) Fidélité. L'organisation de la description devrait réfléter l'organisation du matériel décrit. ISP permet des descriptions fidèles. CASSANDRE possède cette propriété pour les descriptions des niveaux 3 et 4. CDL et LASCAR ne sont pas aussi satisfaisants au niveau du langage-machine. PMS fournit des descriptions ayant cette propriété.

7) Parallélisme. Les ordinateurs travaillant en parallèle, les langages de description doivent pouvoir exprimer ce parallélisme. Tous les langages de description permettent la description d'actions parallèles.

8) Simplicité syntaxique. Les personnes qui auront à décrire des systèmes dans un langage de description ne seront pas nécessairement des programmeurs. Aussi, le langage doit-il être syntaxiquement simple, c'est-à-dire comporter un petit nombre de constructions syntaxiques. Tous les langages vus au paragraphe vus au paragraphe précédent, conçus dès l'origine comme des langages de description, sont relativement simples syntaxiquement, ce qui n'est pas le cas pour les langages de description basés sur des langages de programmation et possédant de ce fait une syntaxe inutilement riche.

9) Indépendance matérielle. Le langage doit être indépendant d'une technologie ou d'une organisation particulière, ce qui est un peu en contradiction avec la propriété de fidélité. PMS est le langage le plus indépendant. CASSANDRE, et, par conséquent, LASCAR, possèdent mieux cette propriété que ISP qui est orienté vers les systèmes asynchrones, alors que CDL est conçu pour décrire surtout des systèmes synchrones.

10) Séparabilité. Le langage doit permettre des descriptions où la structure de contrôle est séparée de la structure des données. PMS exprime bien cette séparation. CDL et CASSANDRE possèdent cette propriété qui est le propre des langages non procéduraux. Les langages procéduraux expriment souvent mal cette séparation.

Aux propriétés que nous venons d'énoncer, nous croyons nécessaire d'ajouter les deux suivantes :

11) Universalité. Nous appellerons universel un langage qui permet de décrire à la fois le niveau système et un ou plusieurs des autres niveaux définis au début de ce paragraphe. LASCAR est partiellement universel, alors que les autres langages, pris séparément, ne le sont pas.

12) Clarté. Une description peut viser deux buts :

- Décrire la matérialité cablée du système, c'est-à-dire ce qui est statique et permanent. C'est le but des descriptions PMS et des descriptions de réseaux de portes logiques.
- Décrire ce qui est supporté par le matériel et qui est souvent variable et dynamique.

Il y a toujours ambiguïté quand on passe, sans le préciser, d'un domaine à l'autre. Un bon langage de description doit établir clairement la frontière entre les deux.
CASSANDRE est clair, à son niveau, et LASCAR l'est partiellement.

PROPRIETE / LANGAGE	1	2	3	4	5	6	7	8	9	10	11	12
ISP	O	O	O	P	N	O	O	O	P	N	N	NA
PMS	O	O	O	O	O	O	NA	O	O	O	N	NA
CASSANDRE	P	O	O	P	O	P	O	O	O	O	N	O
LASCAR	N	O	O	P	N	N	O	O	O	N	P	P
CDL	P	O	N	P	N	N	O	O	P	O	N	NA

O : OUI
N : NON
P : PARTIELLEMENT
NA : NON APPLICABLE

FIGURE I-5

COMPORTEMENT DES LANGAGES DE DESCRIPTION VIS-A-VIS DES 12 PROPRIETES.

I-6.- CONCLUSIONS.

Nous venons de procéder à un examen critique de quelques langages représentatif de description. Quelles conclusions pourrait-on en tirer ? Affirmer qu'un langage est meilleur qu'un autre dans l'absolu, c'est oublier le contexte de définition de chacun d'eux. Pour nous, la qualité d'un langage, de description ou autre, se mesure par rapport aux objectifs qu'il s'était fixés au départ. Un langage ne peut donc être "bon" que s'il répond de manière satisfaisante aux objectifs qui ont conduit à sa définition. Aussi, serait-il absurde de comparer CDL à PMS ou ISP à CASSANDRE car on ne peut raisonnablement comparer que des choses comparables, à savoir des langages prétendant viser des objectifs identiques ou voisins.

Il n'en reste pas moins vrai que l'ensemble ISP - PMS répond de manière plus que satisfaisante aux 12 propriétés énoncées au paragraphe précédent et que sa généralité (trop grande peut-être) permet de viser les objectifs les plus variés. Ainsi, dans la définition de notre système, nous nous sommes inspirés de cet ensemble en restreignant ou en étandant certains de ses aspects en fonction des objectifs que nous avons visés.

Il convient enfin de signaler qu'actuellement un nouveau langage nommé LASSO [13] est en cours d'implémentation à Grenoble. Ses auteurs se fixent pour objectif des descriptions d'architectures de machines dans lesquelles chaque composant est considéré de manière macroscopique. Ainsi, les fonctions réalisées par un composant sont-elles décrites mais non leur mise en oeuvre interne ni leur micro-synchronisation. Un est aspects intéressants du langage est l'existence, dans une description de composant, d'une partie séparée nommée spécifications et qui sert à indiquer les propriétés que doivent satisfaire certains objets du composant pour que la description ait un sens. Cette partie sert de référence pour effectuer des vérifications. Comme le langage est encore au stade de l'implémentation , nous n'en avons pas tenu compte dans l'analyse de ce paragraphe.

C H A P I T R E II

PRESENTATION DU SYSTEME PYTHIE

CHAPITRE II

PRESENTATION DU SYSTEME PYTHIE

II.1.- LES OBJECTIFS.

Nous nous proposons de construire un système d'aide à la conception d'assemblages comportant des microprocesseurs et des circuits intégrés à large échelle (LSI). La description des assemblages physiques et logiques doit s'appuyer sur un langage afin de déboucher sur les services suivants :

1) Obtenir un moyen simple de description transparente à l'utilisateur et permettant de décrire aussi bien la structure physique et logique d'une configuration que son fonctionnement.
   Ceci implique l'utilisation d'un langage de description exploitable sur ordinateur.
   Ce langage devrait posséder, entre autres, les propriétés suivantes :

   - aspect modulaire,
   - prise en compte du facteur temps,
   - séparation partie statique - partie opératoire

2) Utilisation ultérieure en tant qu'outil pédagogique pour l'apprentissage des systèmes. Dans cette optique, l'intérêt d'une génération graphique, même simple, de la configuration à décrire est incontestable.

3) Utilisation du système en tant qu'outil de conception, ce qui implique la possibilité de précision progressive dans les descriptions, permettant l'étude à des niveaux de détail différents des parties de l'ensemble décrit. Cet objectif implique l'existente de primitives de manipulation de modules correspondant à des configurations ou à des composants d'une configuration. Le système devrait, enfin, prévoir la détection des erreurs de conception. Cette détection devrait être aussi bien statique, pour signaler, par exemple, qu'un fil de sortie d'un module est connecté à un fil de sortie d'un autre module, que dynamique pour vérifier la vraisemblance des résultats obtenus.

4) Utilisation du système en tant qu'outil de détection des pannes dans une configuration existante. Le principe de la détection est le suivant : on fait tourner un programme de test sur plusieurs configurations simulées par le système. On admet que le résultat correct est celui donné par la majorité des configurations. On fait tourner le même programme de test sur la configuration à tester. Il y a panne si le résultat ne coïncide par avec le résultat correct.
Cet objectif nécessite l'existence d'une bibliothèque de modules représentant des configurations ou des parties de configuration.

La définition du système PYTHIE sera entièrement guidée par les objectifs que nous venons d'énoncer. Ces objectifs, on vient de le voir intuitivement, ont des implications matérielles que la définition du système prendra en compte. Aussi, cette définition sera, dans une large mesure, la mise en forme de ces implications.

Il convient de noter ici que le premier objectif du système n'est pas de concevoir des boîtiers, c'est-à-dire de donner la possibilité de dessiner les masques d'un circuit L.S.I. mais de combiner des boîtiers, c'est pourquoi :

a - Il doit exister une bibliothèque standard de boîtiers usuels qui constitue l'un des éléments essentiels du système.

b - L'assemblage de boîtiers peut porter sur :

1) Des modules de la bibliothèque standard désignés par un nom et connus du système.

2) Des modules nouveaux qui peuvent être des boîtiers ou des combinaisons de boîtiers et dont il faut décrire le comportement. Il y a donc besoin d'un langage de description qui constituera le second élément essentiel du système.

c - L'existence de la bibliothèque de modules implique la définition d'un certain nombre d'opérations. On doit, par exemple, pouvoir ajouter un module dans la bibliothèque, retirer un module de la bibliothèque, connecter des modules de la bibliothèque à un module décrit par le langage de description, etc...
Toutes ces opérations, qui sont entièrement indépendantes du langage de description, composeront un langage de commandes et constituera le troisième élément de base du système.

d - La description des activités internes d'un boîtier n'est utile que pour ce qu'elle apporte à l'extérieur. Ainsi, la description du fonctionnement interne n'a pas à être scrupuleusement fidèle à la réalité matérielle, pourvu que les sorties soient correctement positionnées en fonction des valeurs des entrées. Cette orientation du système ne signifie nullement que des modules plus simples qu'un boîtier ne pourront pas faire l'objet d'une description, ni que le système ne se prête pas à une description très fine des objets de l'ordre d'un boîtier. En effet, le système doit permettre de décomposer un objet à des objets de complexité moindre et donc d'obtenir des descriptions de plus en plus fines de l'objet de base. Ainsi, pour un composant donné, deux types de description sont possibles :

Soit une description "grossière" respectant le bon positionnement des sorties en fonction des valeurs des entrées, soit une décomposition en des sous-composants, conduisant à une description plus fine du composant de départ. Inversement, il est toujours possible de composer les objets de base pour obtenir des objets de complexité supérieure.

Le fait que les boîtiers considérés seront souvent des processeurs, des mémoires ou des unités d'interface nous conduit à distinguer les voies servant au transfert de données ou d'adresses des voies servant au transfert des commandes. Ainsi, une broche d'entrée de boîtier reçoit l'attribut entrée ou événement suivant qu'il s'agisse d'une entrée de données ou d'une entrée de commandes. De même, une broche de sortie de boîtier recevra l'attribut sortie ou commande suivant les mêmes critères. Les broches bidirectionnelles recoivent toujours l'attribut entrée ou sortie.
Cette distinction, qui n'est pas toujours facile à faire, est laissée au gré de l'utilisateur. Le système tient compte de cette caractérisation des broches pour proposer des regroupements ayant pour objet l'optimisation des connexions. Il s'appuie pour cela sur les notions d'autorité et de maître que nous définissons dans ce chapitre.

II.2.- DÉFINITION DU SYSTÈME PYTHIE.

En nous basant sur les objectifs énoncés au paragraphe précédent, nous avons été amenés à définir le système PYTHIE de la manière suivante :

Le système PYTHIE est un quadruplet (O, L, P, T) où :

- O est un ensemble fini mais extensible d'objets, éventuellement vide.

- L est un langage de description d'objets.

- P est un ensemble fini mais extensible de primitives de manipulation d'objets.

- T est un objet temporaire.

L'ensemble O correspond à la notion de bibliothèque d'objets. Un objet peut y être ajouté, à partir de l'objet temporaire T , ou retiré.

L'objet temporaire T est créé à l'aide du langage de description et des primitives de manipulation.

La définition du langage L de description sera donnée dans le chapitre III.

Pour ce qui concerne la nature des objets manipulés par le système, nous y reviendrons plus loin dans ce chapitre.

Les primitives de l'ensemble P permettent entre autres :

- De créer l'objet temporaire P en utilisant sa description dans le langage L .

- D'ajouter l'objet T dans l'ensemble O , ce qui correspond à l'opération de catalogage du module décrivant le fonctionnement de l'objet, dans une bibliothèque de modules.

- De supprimer un objet dans l'ensemble O (retirer un module de la bibliothèque des modules).

- D'initialiser l'objet T

- D'activer l'objet temporaire, opération qui correspond à l'exécution du module qui décrit le fonctionnement de l'objet.

- De modifier l'objet temporaire. Cette opération permet de reconfigurer l'objet, c'est-à-dire de remplacer un composant ou un ensemble de composants par un objet ou un ensemble d'objets pris dans l'ensemble O ou décrits par le langage de description.

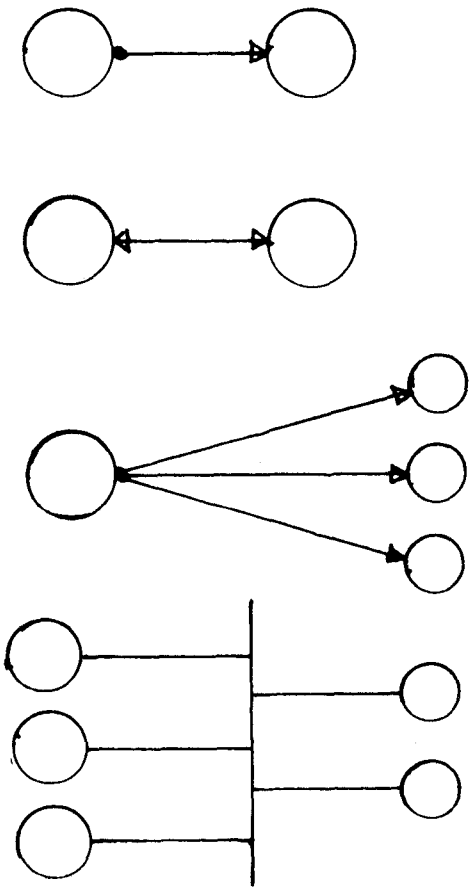
- De restructurer l'objet temporaire. Cette primitive permet au système de regrouper certains composants de l'objet, suivant des critères que nous verrons plus loin.

- De dessiner tout ou partie de l'objet temporaire. Cette primitive permet au système de fournir une représentation graphique simple d'un objet.

II.3.- LES OBJETS DU SYSTÈME.

Ayant doté notre système de primitives de manipulation d'objets, il faut préciser la nature de ces objets. Dans un premier temps, nous avions envisagé d'opter pour des descriptions guidées entièrement par les liaisons. Ceci nous avait conduit à nous orienter vers la définition d'un nombre fini de types d'objets dits de "connexion". La figure II-1 donne un aperçu de quelques types d'objets envisagés.

.../...



liaison simple, sens unique

liaison simple bidirectionnelle

éventail de sortie

bus

FIGURE II-1

Ces types d'objets seraient standardisés et un ensemble d'opérations de composition permettrait de construire des objets de structure complexe formés d'objets simples.

Nous avons finalement abandonné cette idée, cependant élégante et originale, car nous y avons vu quelques inconvénients dont les principaux sont :

- Lourdeur accrue de description provenant du fait qu'il y a, en général, dans une configuration donnée, plus de connexions que d'unités actives.

- Description peu transparente à l'utilisateur et moins naturelle pour qui a l'habitude d'attribuer un rôle secondaire aux

connexions par rapport aux unités actives.

- Enfin et surtout, une telle démarche ne permettrait pas, ou rendrait difficile, la description de la structure logique d'une configuration.

Ainsi, nous avons finalement opté pour la démarche "classique" où la description est plutôt guidée par les unités actives, ce qui nous a amenés à définir ce qu'est un objet de notre système de la manière suivante :

L'objet de base du système, c'est-à-dire l'objet manipulé par ses primitives, est l'unité. Par ce nom, nous désignons une entité logique définie de la manière suivante :

Une unité est un 7-uplet

$$U = (E,V,S,C,Q,f_t,f_s)$$

où :

$E = \{e_1,e_2,\ldots,e_m\}$ est un ensemble fini d'entrées de données.

$V = \{v_1,v_2,\ldots,v_p\}$ est un ensemble fini d'entrées particulières que nous appellerons "événements".

$Q = \{q_1,q_2,\ldots,q_k\}$ est un ensemble fini des états internes.

$S = \{s_1,s_2,\ldots,s_n\}$ est un ensemble fini de sorties de données.

$C = \{c_1,c_2,\ldots,c_\ell\}$ est un ensemble fini de sorties particulières que nous appellerons "commandes".

.../...

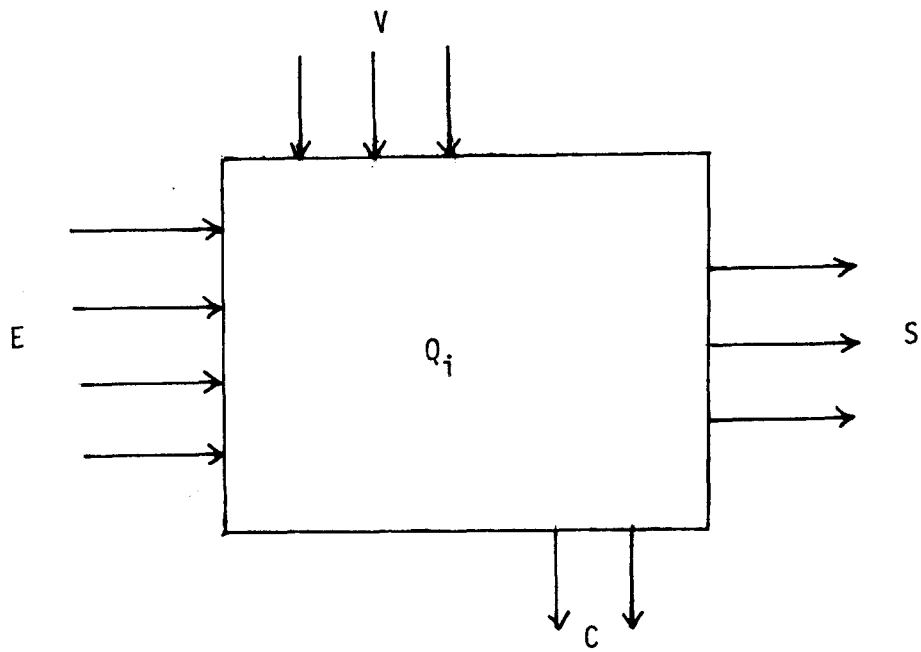


FIGURE II-2

f_t est une fonction de transition d'états

$$f_t : E \times V \times Q \rightarrow Q$$

f_s est une fonction de sortie

$$f_s : E \times V \times Q \rightarrow S \times S$$

Nous avons dit que l'unité est une entité logique; elle peut donc avoir ou ne pas avoir de correspondance avec une entité physique. Nous verrons plus loin des exemples de ce que peut être une unité.

La composition d'unités est une opération logique qui découle de l'idée des machines incluses :

On dispose de la machine M_0, on y connecte la machine M'_0 et l'on obtient la machine M_1 contenant M_0 et M'_0. En continuant de la sorte, on obtient des machines de plus en plus générales, où la machine de niveau n contient toutes les machines des niveaux 0, 1, ..., n-1.

Nous adopterons ici la représentation arborescente de la composition d'unités. Ainsi, l'unité U, composée des unités $U_1, U_2, \ldots, U_n$, aura la représentation :

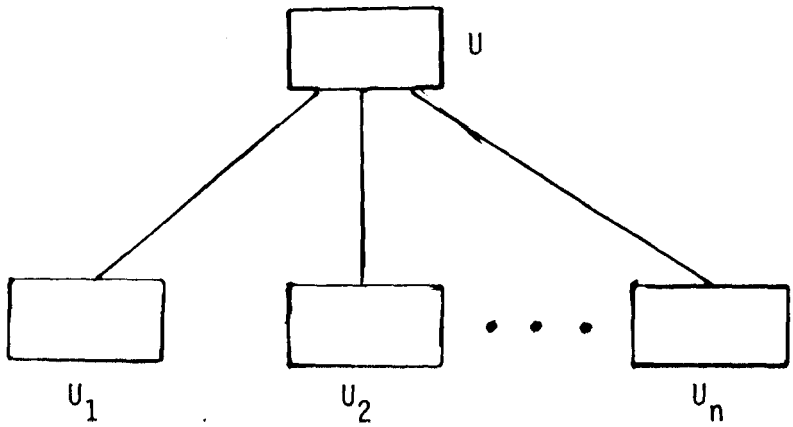


FIGURE II-3

Les unités $U_1, U_2, \ldots, U_n$ pourront à leur tour être décomposées en d'autres unités et ainsi de suite. On arrive ainsi à des unités composées de plusieurs niveaux et dont la figure II-4 donne un exemple de structure logique.

.../...

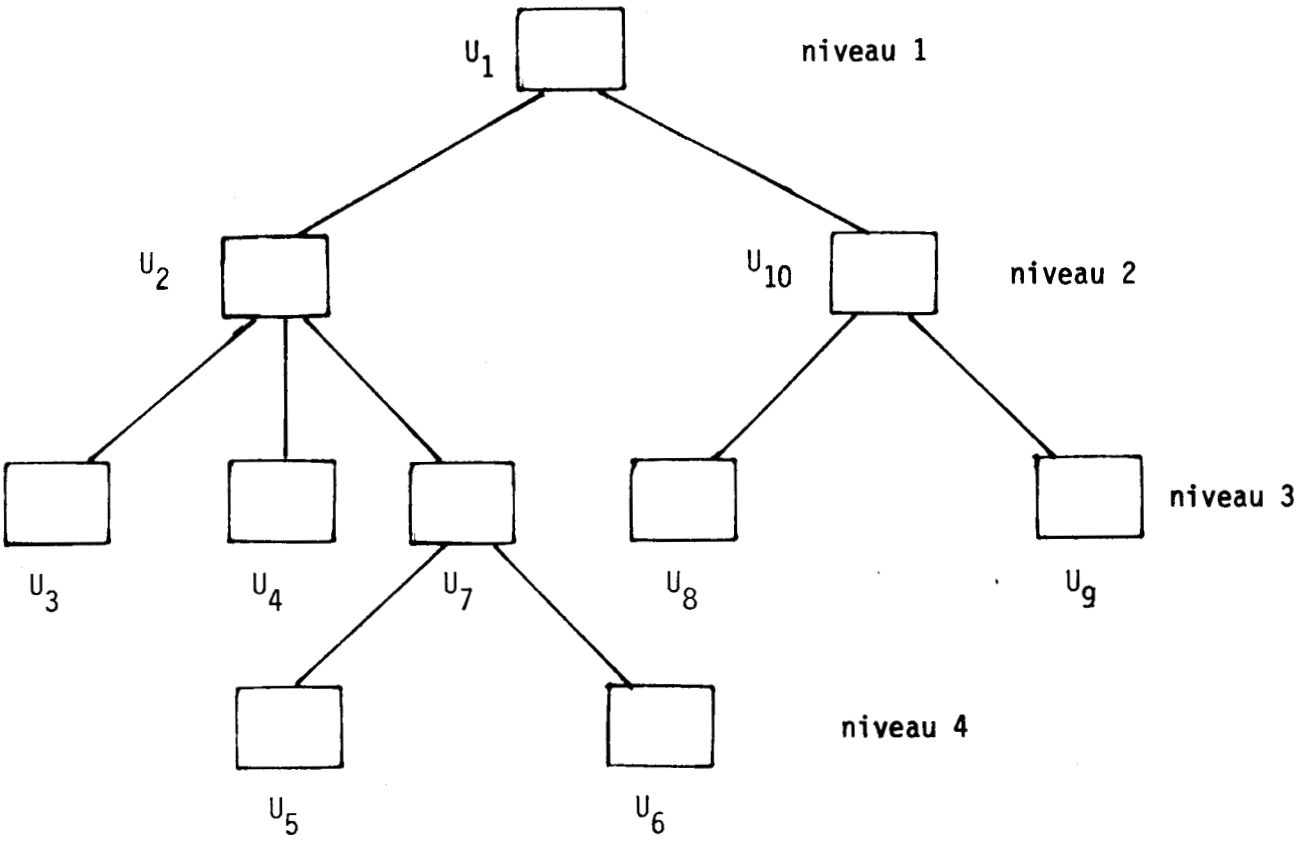


FIGURE II-4

Nous emploierons souvent, pour représenter une unité composée, une notation ensembliste qui n'est, en réalité, qu'une forme condensée de la représentation arborescente. Ainsi, on écrira :

$$U = \{U_1, U_2, \ldots, U_n\}$$

pour représenter l'arborescence de la figure II-3 et :

$$U_1 = \{U_2 = \{U_3, Y_4, U_7 = \{U_5, U_6\}\}, U_{10} = \{U_8, U_9\}\}$$

pour représenter celle de la figure II-4.

Il convient ici de faire la distinction entre la structure logique et la structure matérielle d'une unité composée. Ainsi, les 3 unités de la figure II-5 :

.../...

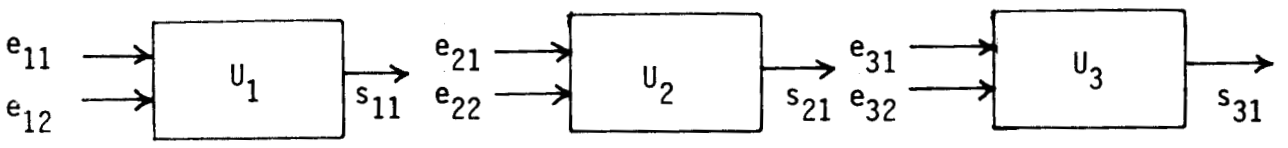


FIGURE II-5

peuvent être assemblées physiquement de la manière suivante :

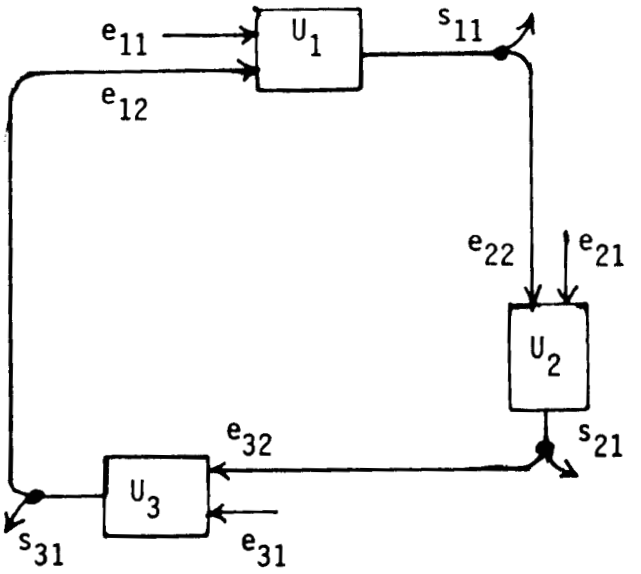


FIGURE II-6

pour donner comme résultat l'unité U de la figure II-7.

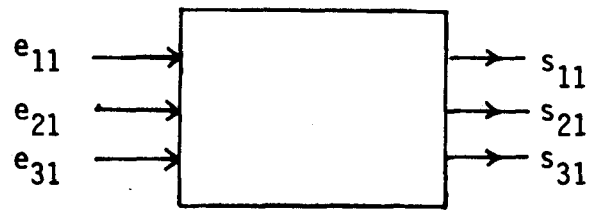


FIGURE II-7

Par contre, la structure logique de l'unité résultante U est toujours arborescente. Ainsi, si l'on considère que l'Unité U est composée des sous-unités U_1, U_2 et U_3, sa structure logique sera :

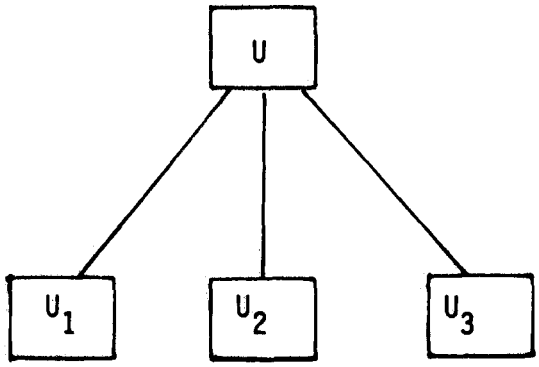


FIGURE II-8

alors que si l'on considère que l'unité U est composée des 2 sous-unités U_1 et U_2', cette dernière étant elle-même compsoée des 2 sous-unités U_2 et U_3, on aura la structure de la figure II-9.

.../...

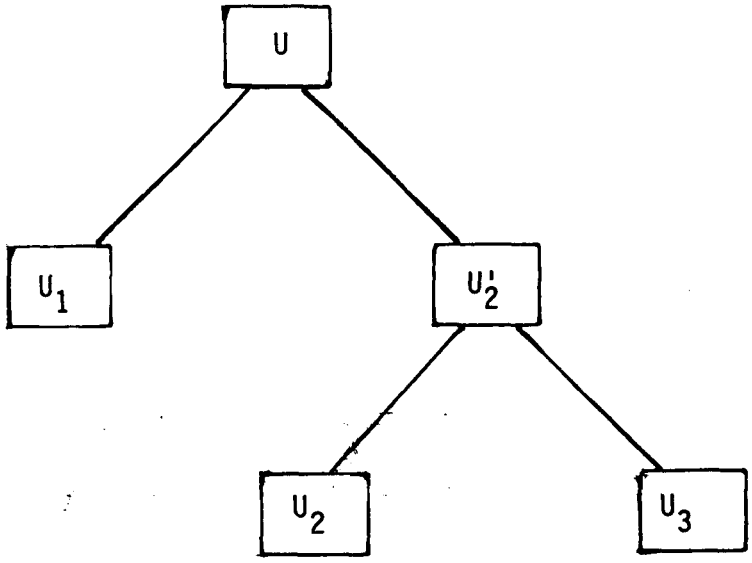


FIGURE II-9

Les unités d'une arborescence donnée peuvent communiquer physiquement entre elles. Toutefois, une unité de l'arborescence ne peut communiquer qu'avec ses frères et ses fils. Nous admettons ici implicitement qu'une unité est toujours en communication avec elle-même.

Dans ce qui suit , nous utiliserons toujours le nom de sa racine pour désigner une arborescence, ce qui est normal puisque ce nom est celui de l'unité composée représentée par l'arborescence.

Nous dirons que l'unité U est une unité simple par rapport à l'unité V si U est une feuille de l'arborescence représentant V .

Nous avons déjà dit qu'une unité est pour nous une entité logique qui peut correspondre ou non à un objet physique. Ainsi, une unité peut être :

- un processeur,
- un boîtier de mémoire,
- une unité d'interface,

mais aussi :

- un ensemble de 3 processeurs,
- une carte-mémoire comportant 2 mémoires RAM (Random Access Memory) et 3 mémoires ROM (Read Only Memory).

On voit qu'une unité peut correspondre à un objet bien délimité matériellement (p.e. processeur), à un ensemble d'objets regroupés physiquement (p.e. placés sur la même carte), ou à un ensemble d'objets regroupés logiquement par les fonctions qu'ils réalisent (p.e. 3 mémoires ROM).

Ainsi, une configuration composée d'un micro-processeur, d'une unité de mémoire elle-même composée d'une mémoire RAM et d'une mémoire ROM, d'une unité d'alimentation et d'une horloge, pourrait être représentée par la structure arborescente de la figure II-10, où µP , RAM, ROM, POWER et H sont des unités simples, ce qui revient à dire que le fonctionnement de ces unités est connu au moment où l'on définit

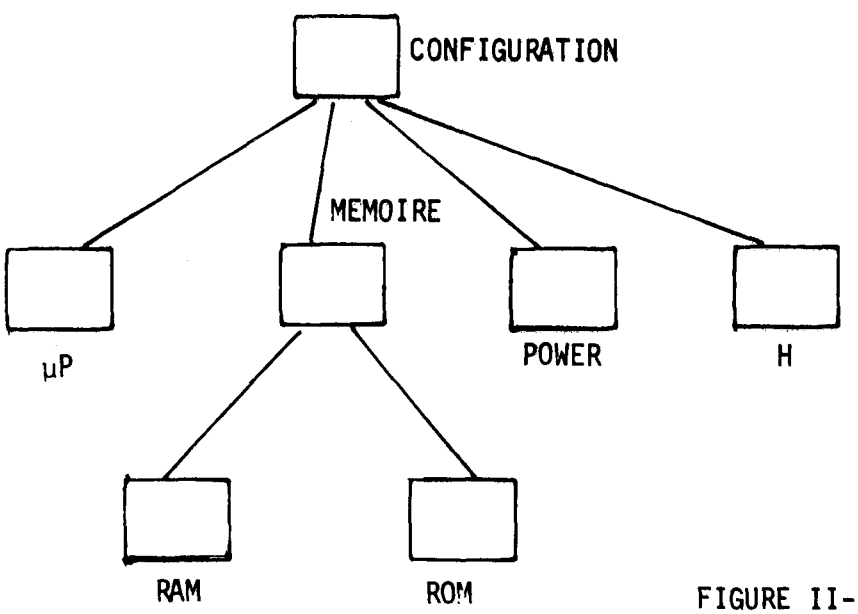


FIGURE II-10

l'unité composée CONFIGURATION. Nous y reviendrons plus en détail dans les paragraphes qui suivent.

II.4.- LES NOTIONS D "AUTORITÉ" ET DE "MAÎTRE".

DEFINITION.- Etant données 2 unités U et V, nous dirons que U exerce une autorité sur V et nous écrirons

$$UaV$$

si et seulement si une commande de U est un événement de V et qu'aucune commande de V n'est un événement de U

On peut généraliser la notion d'autorité et introduire la relation a^* d'autorité indirecte définie comme suit :

1) Si UaV alors Ua^*V
2) Ua^*U
3) Ua^*V et Va^*W implique Ua^*W

où W est une troisième unité.

Si l'on considère maintenant deux unités U_1 et U_2, plusieurs types de relations sont possibles entre elles [19] :

1) U_1 exerce une autorité absolue sur U_2 quel que soit l'état dans lequel se trouve cette dernière unité.
2) U_1 exerce une autorité partielle sur U_2, dans le sens que cette autorité est soumise à des conditions concernant l'état de U_2, ou partagée avec d'autres unités.
3) Il n'existe pas de relation d'autorité entre U_1 et U_2.

Nous dirons que l'unité U_1 est <u>maître</u> de l'unité U_2 si U_1 peut exercer une autorité sur U_2 et que U_2 ne peut pas exercer d'autorité sur U_1.

L'autorité du type 1 (classification ci-dessus) est une autorité statique alors que celle du type 2 est conditionnelle et donc dynamique. Ainsi, dans la figure II-11, les unités U_1 et U_2 sont maîtres de la même unité V mais U_2 n'exerce son autorité sur V que s'il a reçu l'accord explicité de U_1. Dans cet exemple, V pourrait être une mémoire, U_2 un processeur et U_1 une horloge.

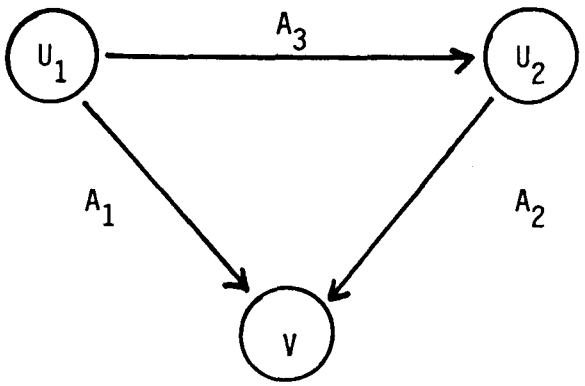


FIGURE II-11

On peut généraliser la situation que l'on vient de décrire en supposant qu'il n'y a pas de communication directe entre U_1 et U_2 (figure II-12) et que U_2 n'exerce son autorité sur V à l'instant t que si U_1 n'exerce pas la sienne au même moment. Il est évident qu'il existe une relation d'autorité implicité entre U_1 et U_2 désignée par la flèche en pointillé.

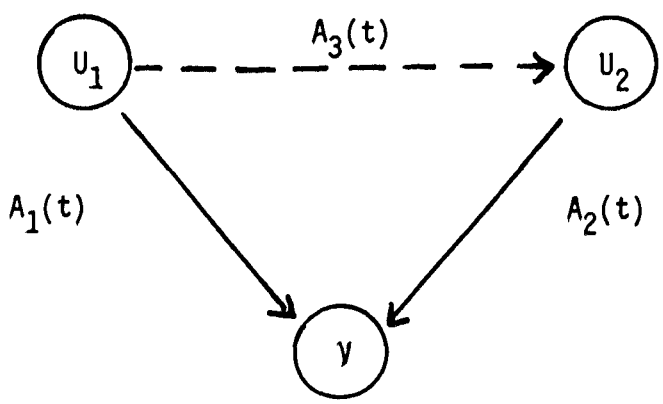


FIGURE II-12

Supposons maintenant qu'à l'instant t , l'autorité explicite A(t) d'une unité sur une autre peut prendre les valeurs 0 et 1 selon que la première est en train d'exercer son autorité sur la seconde ou non.

Soient $A_1(t)$ et $A_2(t)$ les valeurs des autorités explicites relatives de U_1 et U_2 sur V et $A_3(t)$ l'autorité implicite de U_1 sur U_2 à l'instant t . Il est évident que U_1 exerce son autorité implicite sur U_2 si et seulement s'il exerce son autorité explicité sur V . On a ainsi :

$$A_1(t) = A_3(t) \qquad (1)$$

On a, d'autre part :

$$A_2(t) = \overline{A_1(t)} \qquad (2)$$

où $\bar{A}$ désigne le complément à 1 de A , car U_2 n'exerce son autorité sur V que lorsque U_1 n'exerce pas la sienne.

Les relations (1) et (2) conduisent immédiatement aux relations :

$$A_1(t) + A_2(t) = 1 \qquad (3)$$

$$A_3(t) = 1 - \overline{A_1(t)} \qquad (4)$$

Généralisons maintenant la situation de la figure II-12 pour arriver à celle de la figure II-13 où les unités U_1 et U_2 sont maîtres des unités $V_1, V_2, \ldots, V_n$ et où U_2 n'exerce son autorité sur V_i ($i = 1,2,\ldots,n$) que si U_1 n'exerce pas la sienne au même moment. Il existe encore ici une autorité implicite de U_1 sur U_2 .

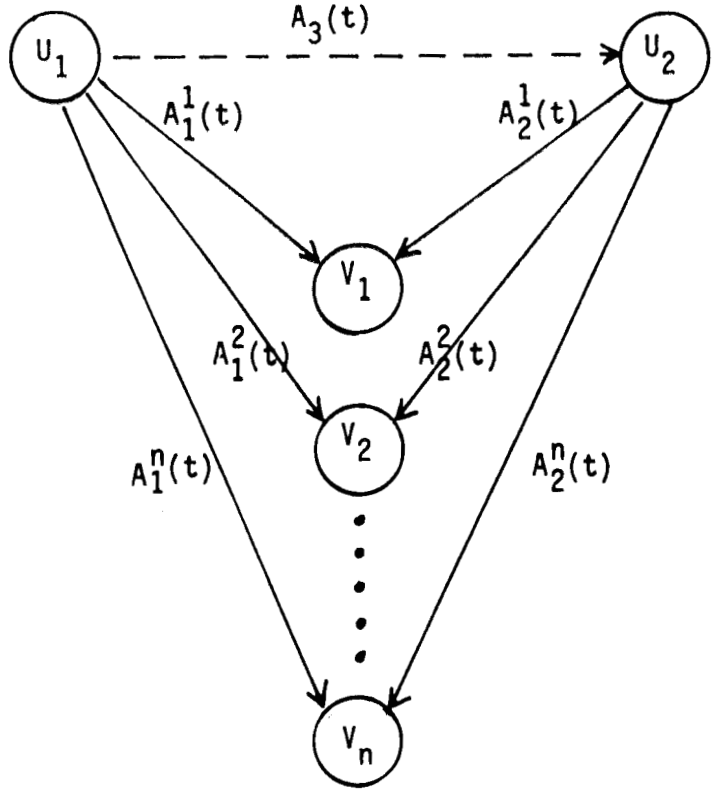


FIGURE II-13

On peut constater que la relation

$$\sum_{i=1}^{n} (A_1^i(t) + A_2^i(t)) = n$$

où $A_1^i(t)$ et $A_2^i(t)$ sont les valeurs des autorités explicites de U_1 et U_2 sur V_i à l'instant t, est toujours vraie.

D'autre part, pour que l'autorité implicite de U_1 sur U_2 rende compte de la situation globale, elle doit prendre ses valeurs dans l'intervalle $[0,1]$. Ainsi, la valeur $c(t)$ de l'autorité implicite exercée par U_1 sur U_2 à l'instant t, sera donnée par la formule :

$$A_3(t) = 1 - \frac{1}{n} \sum_{i=1}^{n} \overline{A_1^i(t)} \qquad (5)$$

Considérons maintenant la situation encore plus générale représentée dans la figure II-14 où les unités U_1 et U_2 exercent leur autorité sur chacune des unités V_i en exclusion mutuelle, ce qui signifie que U_1 n'exerce son autorité sur V_i que si U_2 n'exerce pas la sienne au même moment et réciproquement.

.../...

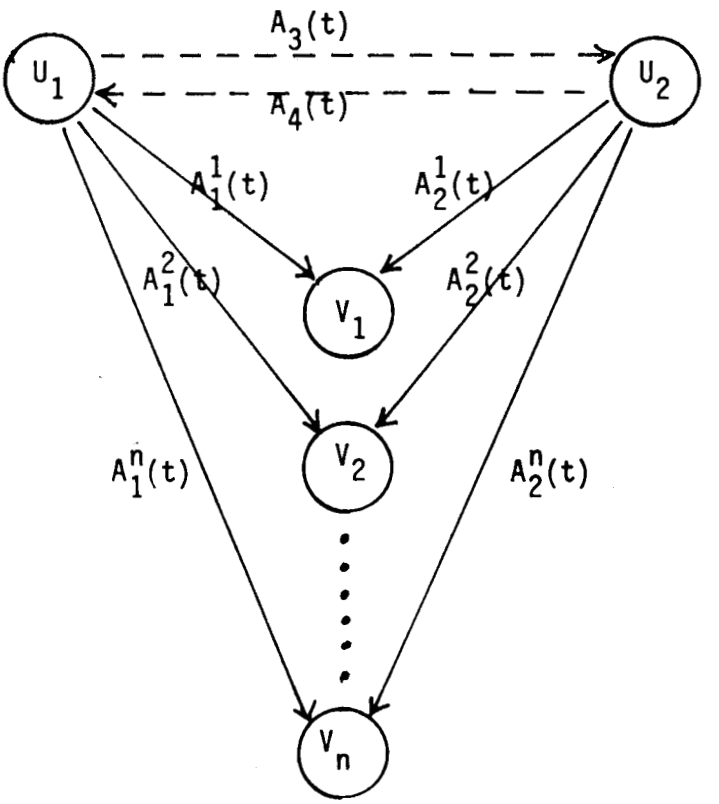


FIGURE II-14

Ainsi, il existe maintenant, à l'instant t, une autorité implicite $A_3(t)$ de U_1 sur U_2 et une autorité implicite $A_4(t)$ de U_2 sur U_1 et les unités U_1 et U_2 sont maître ou esclave l'une de l'autre suivant les valeurs de $A_3(t)$ et $A_4(t)$ qui vérifient, à tout instant, la relation :

$$A_3(t) + A_4(t) = 1 \qquad (6)$$

On a toujours, d'autre part :

$$\sum_{i=1}^{n} (A_1^i(t) + A_2^i(t)) = n$$

et

$$A_3(t) = 1 - \frac{1}{n} \sum_{i=1}^{n} \overline{A_1^i(t)}$$

$$A_4(t) = 1 - \frac{1}{n} \sum_{i=1}^{n} \overline{A_2^i(t)}$$

d'après la formule (5) . Ceci donne :

$$A_3(t) + A_4(t) = 2 - \frac{1}{n} \sum_{i=1}^{n} \overline{(A_1^i(t) + A_2^i(t))} =$$

$$= 2 - \frac{1}{n} \sum_{i=1}^{n} (A_2^i(t) + A_1^i(t)) =$$

$$= 2 - \frac{1}{n} \cdot n =$$

$$= 1$$

ce qui vérifie la relation (6).

On voit que l'autorité dynamique peut prendre des valeurs appartenant à l'intervalle [0,1] alors que l'autorité statique ne prend que des valeurs booléennes. C'est la raison pour laquelle seule la notion d'autorité statique sera introduite dans le système PYTHIE.

II.5.- LA NOTION DE "MAÎTRE" DANS LE SYSTÈME PYTHIE.

Nous avons vu, dans le paragraphe précédent, les différents types de relations d'autorité pouvant exister entre 2 unités. Dans le système PYTHIE, nous ne tenons compte que de l'autorité absolue d'une unité sur une autre. Ainsi, entre 2 unités U_1 et U_2 , il n'existe que l'une des deux possibilités suivantes :

1) U_1 exerce une autorité absolue sur U_2 quel que soit l'état de U_2 et des autres unité environnantes,

2) Il n'existe pas de relation d'autorité entre U_1 et U_2 .

Nous introduirons ces notions sur des ensembles d'unités, descendants directs de la même unité dans une arborescence donnée. Ainsi, tout au long de ce sous-chapitre, toutes les unités considérées seront des éléments de l'ensemble F(A) qui représente l'ensemble des fils d'une unité A.

Considérons une unité composée V . Soient U un de ses descendants de niveau j et $U_1, U_2, \ldots, U_m$ les fils de U .

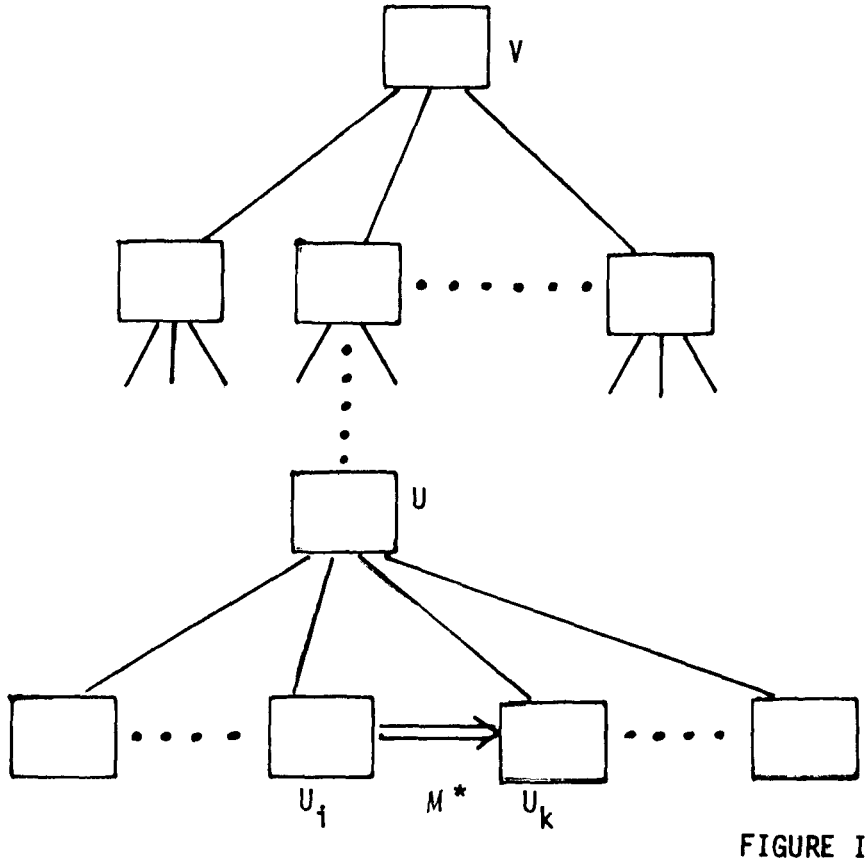


FIGURE II-15

Nous dirons que U_i est <u>maître</u> de U_k et nous écrirons :

$$U_i \; M^* \; U_k$$

si et seulement si, soit $U_i \; a^* \; U_k$, soit $U_i = U_k$.

Entre les deux unités U_i et U_k , trois possibilités peuvent se présenter :

a) $U_i \; M^* \; U_k$

b) $U_k \; M^* \; U_i$

c) $U_i \; \overline{M}^* \; U_k$ et $U_k \; \overline{M}^* \; U_i$

(il n'existe pas de relation d'autorité entre U_i et U_k)

Ainsi, la relation M^* est :

1) <u>Réflexive</u>

2) <u>Antisymétrique</u> car étant données deux unités U et V , descendants directs de la même unité W et telles que $U \neq V$,

$$U \; M^* \; V$$

implique

$$V \; \overline{M}^* \; U$$

3) Transitive car étant données trois unités U , V et W , descendants directs de la même unité ,

$$U \; M^* \; V \quad \text{et} \quad V \; M^* \; W$$

implique

$$U \; M^* \; W$$

par définition,

ce qui fait que la relation M^* est un ordre réflexif. M^* est un ordre partiel car un ensemble d'unités, descendants directs de la même unité, peut ne pas être totalement ordonné par M^* .

Soit $G \subset F(A)$ un ensemble d'unités frères et une unité $U \in G$ de l'ensemble. Nous dirons que U est maître absolu de G et nous écrirons :

$$U \; M_a \; G$$

si et seulement si $U \; M^* \; U'$ pour tout $U' \in G$.

Il est évident par définition qu'un ensemble G d'unités frères n'admet au plus qu'un maître absolu que l'on pourra donc noter $\mu_a(G)$ s'il existe.

On peut considérer à la limite que toute unité U vérifiant la relation $U \; M^* \; V$ est le maître absolu d'un ensemble d'unités mais, vue sous cet angle, la notion de maître absolu ne présente pas beaucoup d'intérêt. Il serait en revanche intéressant de pouvoir déterminer les ensembles les plus larges possibles admettant un maître absolu. On formerait ainsi des classes d'unités dont les représentants seraient les maîtres absolus. Pour former ces classes d'unités, on se base sur la propriété évidente suivante :

Soient $G \subset F(A)$;
$\forall M_a \; G$ et
$U \in F(A) - G$

Si $U \; M^* V$ alors $U \; M_a \; G \cup \{U\}$

D'autre part, les propriétés des unités et des relations M^* et M_a nous amènent à une constatation très intéressante :

Si $F_1 \subset F(A)$ est un ensemble admettant un maître absolu et et $\forall V \in F(A) - F_1$, $\forall W \in F_1 - \{\mu_a(F_1)\}$, $V \; \bar{a} \; U$, alors :

$$E(F_1) = E(\mu_a(F_1))$$

où $E(X)$ désigne l'ensemble des événements sur l'unité (ou l'ensemble d'unités) X , provenant de commandes de ses unités frères.

Dans ce qui suit, nous dirons qu'un ensemble $G \subset F(A)$ est <u>fermé</u> si et seulement si $\forall U \in G$, $\forall V \in [F(A) - G]$, $V \; \bar{a} \; U$, si $U \; \bar{M}_a \; G$.

Toutes ces remarques nous amènent à donner au système la possibilité de restructurer des unités composées, proposées par l'utilisateur. La méthode suivie sera la suivante :

Le système travaille sur l'ensemble des fils d'une unité donnée. Sur cet ensemble, il détermine les sous-ensembles les plus larges possibles admettant un maître absolu. Il est possible que ces sous-ensembles ne soient pas disjoints. A partir de ces ensembles, le système construit les plus grands sous-ensembles fermés, s'ils existent, et propose des regroupements d'unités selon les principes suivants :

1) Si F_1 , F_2 , ..., $F_k \subset F(A)$ sont des ensembles disjoints et fermés d'unités pour lesquels $\mu_a(F_i)$ est défini pour tout i , le système crée K nouvelles unités nommées F_1 , F_2 , ..., F_K , chacune d'elles étant composée des unités de l'ensemble qu'elle nomme.

2) Si F_1 , F_2 , ... $F_K \subset F(A)$ sont des ensembles d'unités pour lesquels $\mu_a(F_i)$ est défini pour tout i et non nécessairement disjoints, le système

a) considère l'ensemble

$$S = \bigcup_{j=1}^{K} \bigcup_{i=j+1}^{K} F_i \cap F_j$$

b) crée K unités

$$G_j = F_j - S \quad \forall\, j = 1,2,\ldots,K$$

c) crée l'unité

$$G_{k+1} = S \quad \text{si elle existe}$$

ce qui revient à dire qu'il regroupe ensemble les unités de F_i admettant un seul maître absolu, et ce maître absolu, pour former une nouvelle unité, et ceci pour tout $i = 1,2,\ldots,K$ Les unités restantes sont regroupées à leur tour ensemble pour former la $K+1$ ème unité.

Ainsi, si F_1 et F_2 sont des ensembles d'unités, fils de la même unité U et admettant chacun un maître absolu, le système proposera la structure ;

.../...

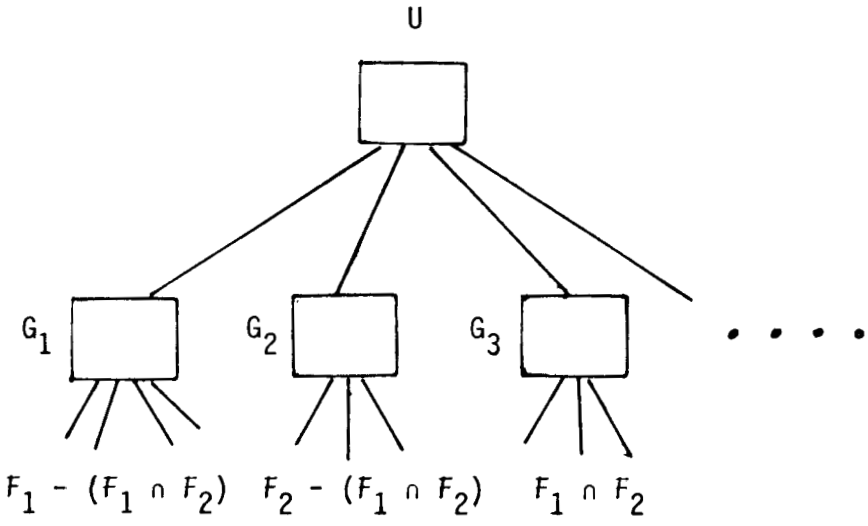


FIGURE II-16

où F_1 et F_2 peuvent être des ensembles disjoints, auquel cas $G_1 = F_1$, $G_2 = F$ et l'unité G_3 disparaît.

Si maintenant

$$U \; M_a \; [F_1 - \{V\}], \quad ,$$

$$V \; M_a \; [F_1 - \{U\}],$$

$$U \; \overline{M}^* \; V \quad \text{et} \quad V \; \overline{M}^* \; U$$

nous avons un cas particulier du précédent qui conduit à la structure :

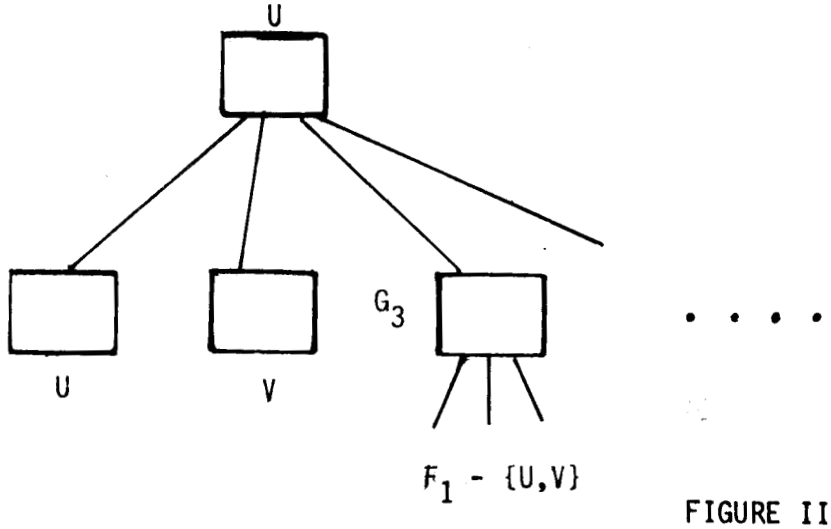


FIGURE II-17

Enfin, si $G_1 = \{U_1, U_2, \ldots, U_n\}$ est un ensemble d'unités frères et que :

$$U_1 \; M_a \; G_1 \quad \text{avec} \quad G_1 = \{U_1, U_2, \ldots, U_k\} \;,$$

$$U_{k+1} \; M_a \; G_2 \quad \text{avec} \quad G_2 = \{U_{k+1}, \ldots, U_m\} \;,$$

$$U_1 \; \overline{M}^* \; U_{k+1} \quad \text{et} \quad U_{k+1} \; \overline{M}^* \; U_1 \;,$$

où $U_{m+1}, \ldots, U_n$ n'admettent pas de maître, la structure proposée par le système sera :

$$G = \{G_1 = \{U_1, \ldots, U_k\} \;,\; G_2 = \{U_{k+1}, \ldots, U_m\} \;,\; U_{m+1}, \ldots, U_n\}$$

Les regroupements logiques proposés par le système ont pour objectif de signaler certains regroupements physiques, tels que l'assemblage sur la même carte d'une unité-maître et de toutes ses unités-esclaves, qui rendraient la configuration plus efficace en optimisant le nombre de connexions. Cette propriété est implémentée grâce à la primitive RESTRUCTURER décrite au chapitre IV.
Il est à noter que le système PYTHIE ne fait des regroupements qu'à partir des unités définies par l'utilisateur et qu'il ne décompose jamais une unité de l'utilisateur.

Voyons, pour terminer, sur un exemple plus concret - la configuration de la figure II-10 - , le mécanisme des regroupements du système. Si le graphe des maîtres des unités-frères du niveau 2 est donné par la figure II-18, où les flèches indiquent que la relation M^* est vérifiée,

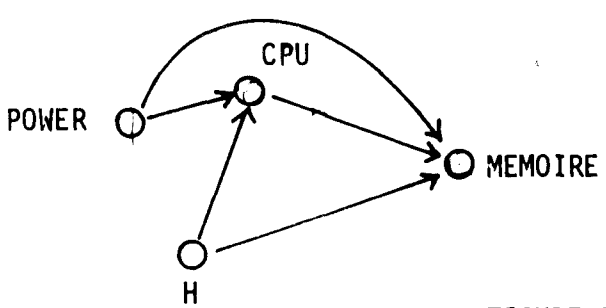


FIGURE II-18

la configuration proposée par le système sera la suivante :

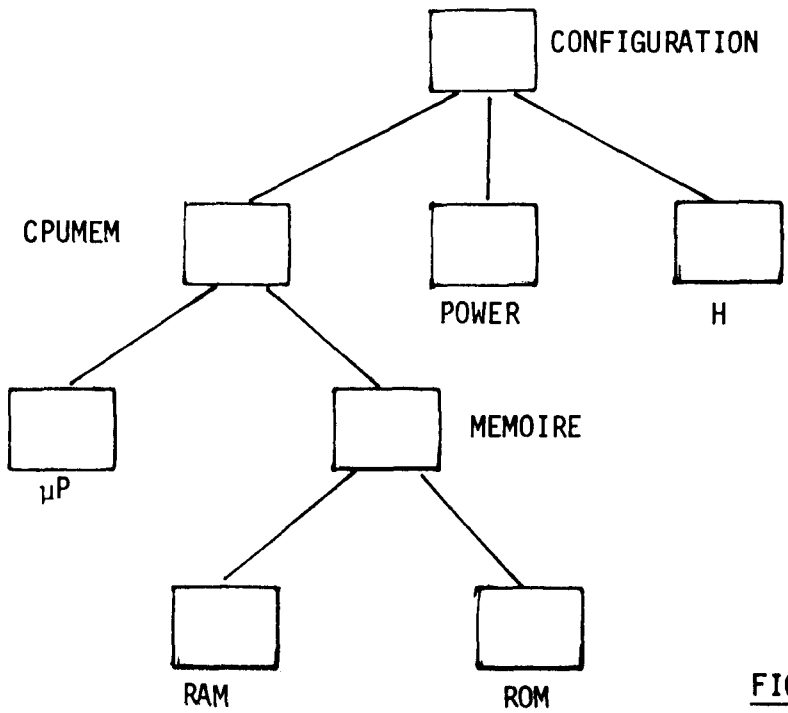


FIGURE II-19

II.5.- LA REPRÉSENTATION GRAPHIQUE.

Lorsque l'utilisateur, que ce soit un concepteur ou un étudiant faisant l'apprentissage des systèmes, propose sa configuration à un système de description en utilisant le langage approprié, il aimerait que celui-ci lui fournisse une représentation graphique, même simple, de cette configuration, lui permettent de vérifier que l'assemblage proposé à travers le langage de description, est bien celui qu'il avait imaginé. De même, lorsqu'il demande au système de faire des restructurations sur une configuration donnée, il aimerait avoir une représentation graphique du résultat de cette restructuration.

.../...

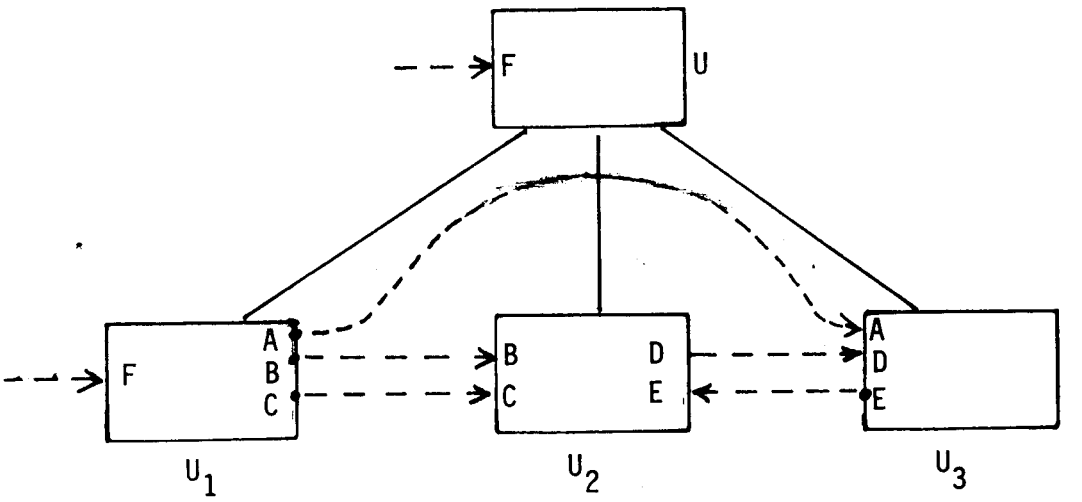


FIGURE II-20

Aussi, le système PYTHIE dispose-t-il d'une primitive de représentation graphique élémentaire d'une unité.

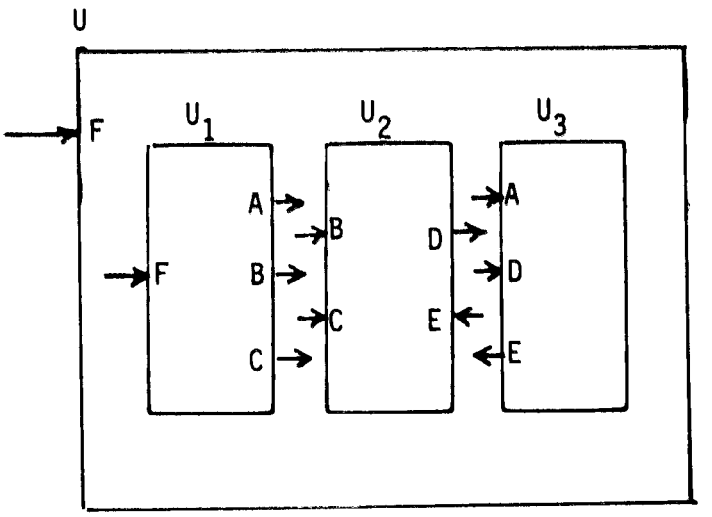


FIGURE II-21

Cette représentation est très simple et met en évidence les connexions et la structure logique de l'unité à représenter et de ses unités composantes.

Le principe de la représentation graphique d'une unité par le système est expliqué sur les figures II-20 et II-21. Ainsi, l'unité U , composée des trois unités U_1 , U_2 et U_3 , de la figure II-20 , où les flèches en pointillé représentent les connexions, se verra attribuer par le système la représentation de la figure II-21.

Nous parlerons des détails techniques de la représentation d'une unité dans la partie consacrée aux primitives du système.

C H A P I T R E III

-=-=-=-=-=-=-=-=-=-=-

LA DESCRIPTION DES OBJETS DU SYSTEME

CHAPITRE III

LA DESCRIPTION DES OBJETS DU SYSTEME

III.1.- LES DESCRIPTEURS

Les objets du système étant en général des unités composées, c'est la description de ces dernières qui sera l'objet de ce paragraphe.

Pour décrire une unité composée, il faut décrire :

a) Le niveau de chaque unité composante dans l'arborescence.

b) Les liaisons permises entre les unités ainsi que le type de ces liaisons.

c) Le fonctionnement interne de chaque unité.

En réalité, pour ce qui concerne le fonctionnement interne des unités, on peut se limiter à la description de celui des seules unités simples car le fonctionnement interne d'une unité est le résultat de la composition des fonctionnements de ses unités composantes.

D'autre part, si plusieurs unités simples sont de même nature (c'est-à-dire qu'elles ont le même fonctionnement interne, le même nombre de paramètres d'entrée et le même nombre de paramètres de sortie), il suffit de décrire une unité-modèle de cette nature que nous appellerons prototype. Par exemple, pour décrire 3 mémoires RAM du même type, il suffit d'en décrire une.

Une unité composée est décrite par ce que nous appellerons un descripteur d'unité composée. Ce descripteur est constitué par

juxtaposition dans l'ordre :

a) D'un descripteur des connexions.

b) D'un descripteur de prototypes.

c) D'un descripteur des actions.

Le descripteur des connexions décrit la structure logique de l'unité composée et les liaisons physiques des unités composantes, ainsi que les types de ces liaisons. Il indique en outre l'unité d'initialisation de l'unité composée. Ce descripteur est la seule partie obligatoire d'un descripteur d'unité composée.

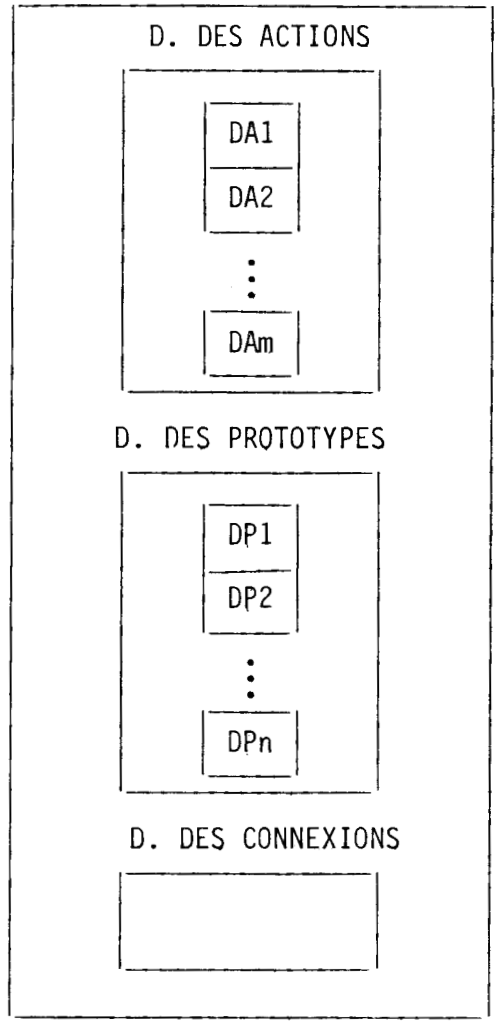


FIGURE III.1

Le descripteur des prototypes est composé d'un ensemble de descripteurs, un par prototype. Il est facultatif dans la mesure où les descripteurs de tous les prototypes auxquels il est fait appel dans le descripteur des connexions sont stockés dans une bibliothèque de descripteurs de prototypes.

Le descripteur des actions est composé d'un ensemble de descripteurs, un par action.

Une action est une suite d'instructions élémentaires susceptibles d'être rencontrées dans plusieurs descripteurs de prototypes et que l'on a été, de ce fait, amenés à regrouper.

Le descripteur des actions est facultatif si les descripteurs de toutes les actions nécessaires sont stockés dans une bibliothèque de descripteurs d'actions.

Pour la mise en oeuvre du descripteur d'unité composée, nous avons fait appel à un langage de description défini suivant les considérations exposées dans ce paragraphe. La présentation détaillée de ce langage se trouve dans le chapitre III.

III.2.- LE LANGAGE DE DESCRIPTION.

Le langage de description LEDA (Langage d'Etude et de Description d'Architectures de micro-processeurs) a été défini pour permettre la mise en oeuvre claire et précise des descripteurs du système PYTHIE.

Pourquoi recourir à un nouveau langage ? N'aurait-on pas intérêt à choisir un des langages de description existants quite à l'adapter aux particularités du système PYTHIE ? Cette solution a finalement été écartée car si la plupart des éléments nécessaires à la mise en oeuvre des descripteurs du système existent dans les langages de description actuels, ils y sont éparpillés à raison de quelques éléments par langage. Dans ces conditions, l'adaptation d'un seul de ces langages aux particularités du système PYTHIE risquait de ne pas être homogène et consistante.

Ce sont ces considérations qui ont finalement conduit à la définition de LEDA qui est un langage de description permettant d'exprimer aussi bien la partie statique, comprenant les liaisons et la structure logique de la configuration décrite, que la partie opératoire correspondant au fonctionnement interne des unités. Il est, d'autre part, interprétable permettant ainsi la mise en oeuvre d'un simulateur exécutable sur un ordinateur classique. Dans la simulation, le temps continu est remplacé par une suite d'unités de temps discrèt t_0, t_0+1,..., ce qui introduit implicitement une horloge externe. Grâce à cette horloge implicite, LEDA permet des descriptions aussi bien synchrones qu'asynchrones.

Un programme écrit en LEDA est composé de trois parties correspondant aux trois descripteurs du système. Les deux dernières parties peuvent être vides. La première est obligatoire et décrit la structure logique d'une unité composée ainsi que l'assemblage des unités qui la composent.

La définition et la décomposition d'une unité sont laissées au gré de l'utilisateur qui peut définir à sa convenance, par exemple :

- Une unité CARTEMEMOIRE comportant
  . 3 mémoires RAM
  . 2 mémoires ROM
- Une unité PROCESSEUR correspondant à un boîtier unique
- Une unité BIPROCESSEUR définissant l'ensemble de deux processeurs.

Toute unité non subdivisée dans le découpage voulu par l'utilisateur (unité simple) doit être définie par un prototype dont le fonctionnement est connu :

- soit par définition explicite de l'utilisateur dans le descripteur des prototypes,
- soit par existance dans une bibliothèque de prototypes.

Nous allons, dans ce qui suit, présenter de manière informelle les différentes parties du langage par des exemples. La définition formelle de LEDA est présentée en annexe.

III.2.1.- LE DESCRIPTEUR DES CONNEXIONS.

Pour présenter le descripteur des connexions, reprenons l'exemple de la figure II-10 qui représente l'unité CONFIGURATION composée d'un µP , d'une mémoire (RAM et ROM), d'une horloge et d'une unité d'alimentation. La figure III-2 présente les connexions physiques entre les unités composantes de CONFIGURATION.

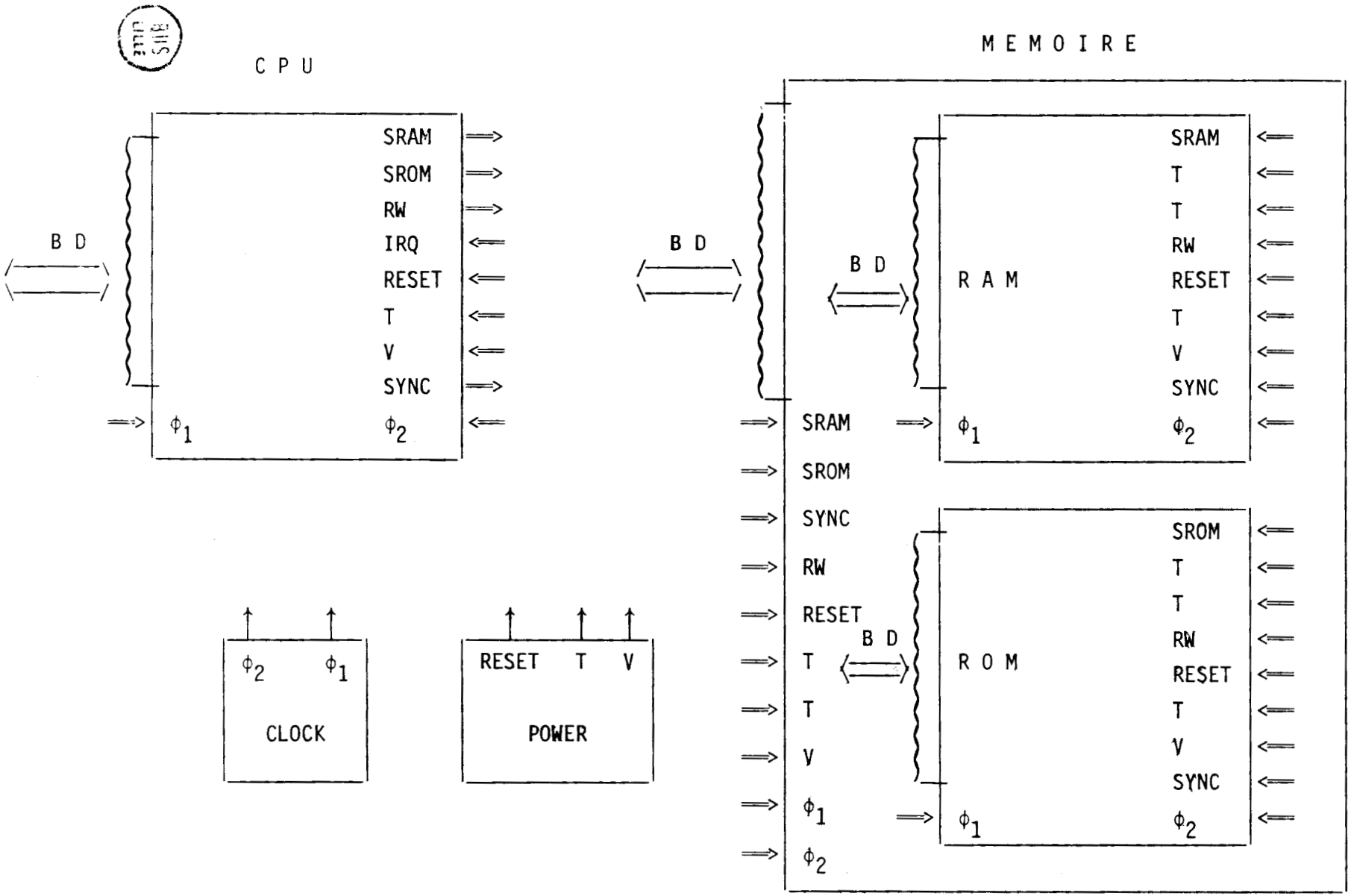


FIGURE III-2

Ces connexions sont constituées essentiellement d'un bus BD assurant aussi bien le transfert des données que celui des adresses et d'un certain nombre de signaux dont les deux phases d'horloge ϕ_1 et ϕ_2 .

A l'exception des liaisons assurant le transfert des données et qui sont du type entrée ou sortie suivant le cas (ici BD est du type entrée ou sortie), toutes les autres liaisons sont supposées être du type événement ou commande.

Voilà maintenant le descripteur des connexions de cette configuration :

N° de ligne	DESCRIPTEUR
1	Connexions;
2	1 Unité CONFIGURATION(POWER),/*POWER EST
3	LA PREMIERE UNITE ACTIVEE*/
4	liaisons BD[0:7],PHI1,PHI2,SYNC,V,T,
5	RESET,SROM,SRAM,RW,IRQ ev;
6	2 Unité CPU(BD es,PHI1 ev,PHI2 ev,
7	SYNC co,V ev,T ev,REJET ev,
8	IRQ ev,RW co,SROM co,SRAM co)
9	def CPUTH;
10	2 Unité MEMOIRE(BD es,SRAM ev,SROM ev,
11	SYNC ev,RW ev,RESET ev,T ev,
12	T ev,V en,PHI1 ev,PHI2 ev);
13	3 Unité RAM(BD es,PHI1 ev,PHI2 ev,
14	SYNC ev,V ev,T ev,RESET ev,
15	T ev,T ev,SRAM ev)
16	def RAMTH;
17	3 Unité ROM(BD es,PHI1 ev,PHI2 ev,
18	SYNC ev,V ev,T ev,RESET ev,
19	RW ev,T ev,T ev,SROM ev)
20	def ROMTH;
21	2 Unité POWER(V co,T co,RESET co)
22	def POWERTH;
23	2 Unité CLOCK(PHI1 co,PHI1 co) def CLOCKTH;
24	Fconnexions;

COMMENTAIRES :

Ligne 1 : Le mot-clé connexions indique le commencement du descripteur des connexions alors que Fconnexions en indique la fin.

Ligne 2 : L'unité de niveau 1 est l'unité composée à décrire. Des commentaires, précédés du symbole "/*" et suivis du symbole "*/", peuvent être insérés à un endroit quelconque de la description.

Ligne 4 : Le mot-clé liaisons est suivi d'une liste de variables définissant les connexions. Les nombres entre crochets représentent le nombre de fils et, éventuellement, leur numérotation.

Ligne 5 : La connexion IRQ est suivie du mot-clé ev (événement) pour indiquer qu'il s'agit d'un fil venant de l'extérieur de la configuration. Les autres liaisons sont internes.

Ligne 6 : La liste des arguments suivant l'unité nommée CPU établit les connexions par rapport aux variables définies par liaisons. Les mots-clés e , s , es , ev ; co permettent de préciser la nature et le sens de transfert des signaux (entrée, sortie, entrée et sortie, événement, commande).

Ligne 9 : déf CPUTH affecte à l'unité logique CPU un prototype nommé CPUTH dont la définition, donnée par ailleurs, permet l'exécution ultérieure en simulation. Le même prototype peut être affecté à plusieurs unités.

La sémantique de cette description est la suivante :

Pas 1 - A l'instant t_0 , seule l'unité d'initiatlisation POWER correspondant au prototype POWERTH est considérée et son exécution initialise ses paramètres de sortie ou de commande.

Pas 2 - A l'instant $t_0 + i$ $(i = 1,2,\ldots,n)$, toutes les unités sont considérées et leur exécution en parallèle positionne leurs paramètres de sortie ou de commande pour l'instant $t_0 + i + 1$. La passage de l'instant $t_0 + i$ à l'instant $t_0 + i + 1$ se fait, pour chaque unité, soit explicitement par la présence d'une instruction ATTENDRE, soit implicitement par l'absence d'autres actions à effectuer. Au cas d'une interruption par ATTENDRE, l'exécution reprendra en séquence à l'instant $t_0 + i + 1$.

III.2.2.- LE DESCRIPTEUR DE PROTOTYPES

Le descripteur des prototypes est composé d'un ensemble de descripteurs, un par prototype. Ainsi, le descripteur des prototypes de la configuration de la figure III-1 doit décrire les prototypes CPUTH, RAMTH, ROMTH, POWERTH et CLOCKTH, à moins que la description de certains d'entre eux n'existe déjà dans une bibliothèque de prototypes.

Le prototype nommé CLOCKTH que nous allons décrire ici en LEDA représente une horloge à 2 phases qui sert à la synchronisation du système et donc le chronogramme se trouve en figure III-3

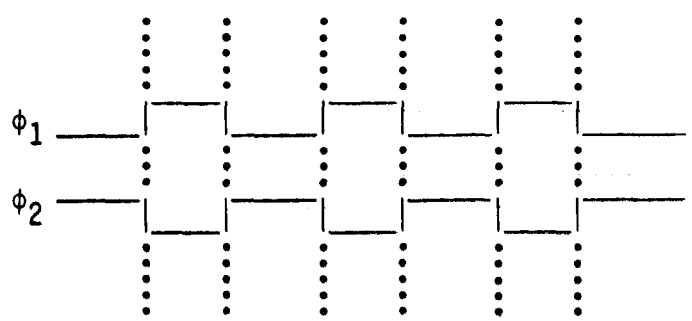


FIGURE III-3

N° de ligne	DESCRIPTEUR
1	Unité CLOCKTH(PHI1 co,PHI2 co);
2	Sélect
3	PHI1=nd : 1 → PHI1; 0 → PHI2; #
4	vrai : non PHI1 → PHI1; non PHI2 → PHI2; #
5	Fsélect;
6	Funité

COMMENTAIRES

Ligne 1 : PHI1 et PHI2 sont les paramètres du prototype CLOCKTH qui sont du type Commande. Ces paramètres représentent des broches ou des ensembles de broches. Leurs noms peuvent être quelconques car ils sont remplacés à chaque fois par ceux des paramètres d'une unité définie par ce prototype.

Ligne 2 : L'instruction sélective regroupe des actions dont l'exécution dépend de la vérification des conditions qui les précèdent et qui sont évaluées séquentiellement. Lorsqu'une condition est vérifiée, l'action correspondante et elle seule est exécutée en un ou plusieurs unités de temps.

Ligne 3 : Le mot-clé nd représente l'état de non définition. Tout objet n'ayant pas reçu d'affectation de valeur se trouve dans cet état. Si PHI1 est dans l'état de non définition, l'action composée des deux instructions de transfert 1 → PHI1 et 0 → PHI2 est exécutée. Les instructions composant une action sont supposées exécutées (dans une seule unité de temps) séquentiellement car ceci n'a pas beaucoup d'importance pour ce qui concerne les résultats, ceux-ci ne devenant effectifs qu'à la fin de chaque unité de temps. A noter le sens du transfert dans les instructions de transfert où c'est la partie droite qui reçoit le résultat. Ceci nous semble en effet plus naturel pour un utilisateur qui n'est pas forcément un programmeur. Le symbole "#" indique la fin de la liste d'instructions à exécuter si une condition est vérifiée.

Ligne 4 : Le mot-clé vrai est une constante booléenne. Elle correspond en fait ici à autre cas d'ALGOL 68 ou à T de LISP. L'opérateur unaire non fournit le complément logique d'une opérande. Ainsi, si PHI1 vaut 1, non PHI1 vaut 0.

L'unité CLOCKTH que nous venons de voir est assez facile à décrire dans la mesure où les seuls objets participant à la description sont les paramètres de l'unité. Dans la plupart des cas, cependant, le fonctionnement d'une unité dépend aussi d'autres objets, internes à l'unité, tels que mémoires, registres, éléments de mémorisation, etc... Il arrive aussi que, pour les besoins de la description, l'on soit amené à définir des objets, e.g. des compteurs, qui peuvent ne pas exister physiquement dans l'unité.

Une illustration de l'utilisation de ces objets internes est donnée dans la description qui suit du prototype RAMTH représentant une mémoire RAM.

.../...

N° de ligne	DESCRIPTEUR
1	Unité RAMTH(BD(8) es, PHI1 ev, PHI2 ev, SYNC ev,
2	V ev; T ev, RESET ev, RW ev, T ev, T ev SRAM ev);
3	Déclarations;
4	mémoire MV(0 : 255) bit(0 : 7);
5	registre BUFES bit(0 : 7);
6	entier CYCRAM;
7	Fdéclarations
8	Sélect
9	(PHI2 = 1 et RESET = 0 et SRAM = 1 et RW = 0 :
10	MV(BD,1) → BD;#
11	(PHI2 = 1 et RESET = 0 et SRAM = 1 et RW = 1 et
12	(CYCRAM = nd ou CYCRAM = 0) :
13	1 → CYCRAM;
14	BD → BUFES;#
15	(PHI2 = 1 et RESET = 0 et SRAM = 1 et RW = 1 et
16	CYCRAM = 1 :
17	BD → MV(BUFES,1);
18	0 → CYCRAM;#
19	Fsélect;
20	Funité

COMMENTAIRES

Ligne 4 : Déclaration d'une mémoire comportant 256 mots numérotés de 0 à 255, chacun des mots étant composé de 8 bascules numérotées de 0 à 7.

Ligne 5 : Le registre BUFES représente une mémoire-tampon de 1 mot composé de 8 bascules.

Ligne 6 : Les objets du type entier peuvent ne pas exister physiquement dans l'unité décrite. Ils sont introduits pour les besoins de la définition. Ici, l'objet CYCRAM sert de compteur de cycles interne à l'unité RAMTH.

Ligne 8 : L'instruction sélective qui se termine en ligne 19 décrit le fonctionnement interne de l'unité. La condition RW = 0 (ou simplement non RW) indique une demande de lecture mémoire alors que la condition RW = 1 (ou RW) indique une demande d'écriture mémoire. Si la première condition est vérifiée, le contenu d'un mot mémoire d'adresse [BD] est transféré dans BD.

Ce genre de description se présente assez bien lorsqu'il s'agit d'unités relativement simples. En revanche, la description pourrait devenir fastidieuse pour des unités capables d'exécuter des instructions. Il faut, en effet, alors décrire le format de chaque instruction, le mode d'adressage qui lui est applicable, ainsi que le processus de lecture ou d'écriture d'une information en mémoire. Il faut d'autre part décrire le processus d'acquisition et de décodage d'une instruction, ainsi que son exécution, tout en respectant le découpage en cycles d'horloge. Or, ceci conduirait à des descriptions relativement lourdes et, souvent, peu claires. Il est évident que l'on ne peut pas se contenter d'écrire

$$A + M \rightarrow A$$

pour indiquer que l'instruction effectue une addition des contenus de l'accumulateur A et de la mémoire M et range le résultat dans l'accumulateur A, car cette notation passe sous silence un certain nombre d'actions comme l'adressage et les accès mémoire (en lecture et en écriture). Il n'en reste pas moins que cette notation est parfaitement lisible et c'est la raison pour laquelle nous essaierons de nous en rapprocher le plus possible. Ainsi, la solution retenue sera celle des schémas d'instruction qui permettent de décrire brièvement l'exécution d'une instruction en laissant au système une partie du découpage en unités de temps. Dans

l'exemple qui suit nous explicitons l'utilisation des schémas d'instructions.

L'unité que l'on se propose de décrire est le processeur très simplifié de la figure III-4. Il s'agit du processeur CPUTH faisant partie de la configuration de la figure III-2. Les voies de communication internes n'apparaissent pas à ce niveau de description.

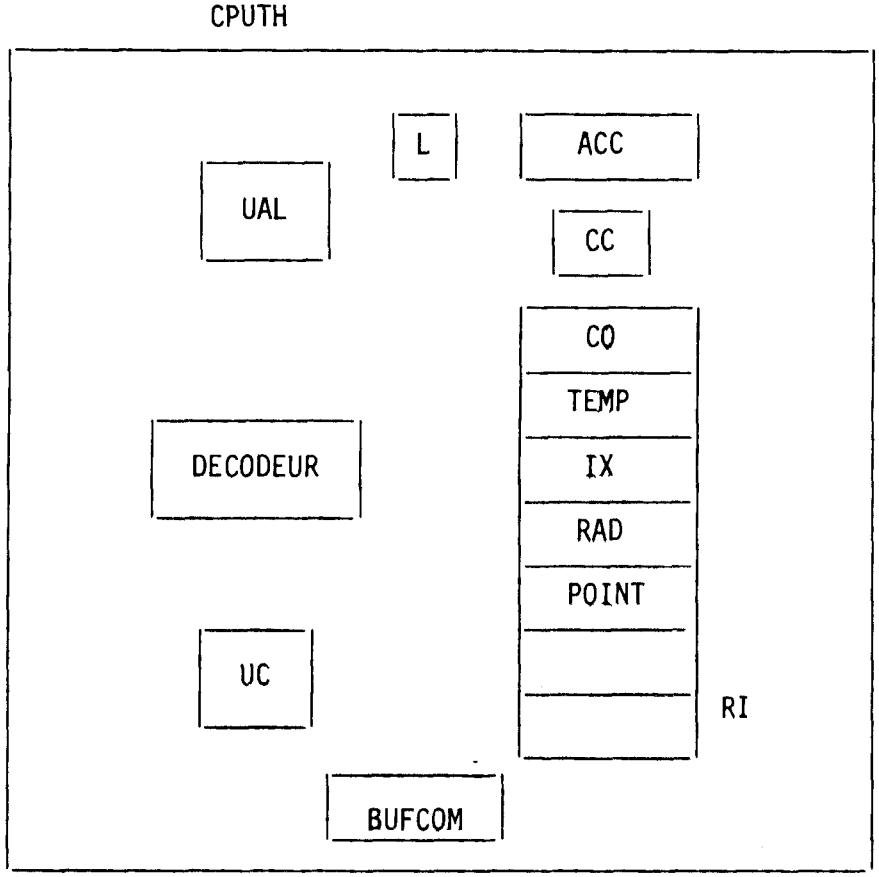


FIGURE III-4

Ce processeur est composé :

- d'une unité arithmétique et logique UAL,
- d'un décodeur d'instructions DECODEUR,
- d'une unité de commande UC,
- d'un registre d'état CC,
- d'un accumulateur ACC,
- d'un compteur ordinal CO,
- d'un registre d'adresse RAD,
- d'un registre d'instruction RI,
- d'un lien L ,
- d'une mémoire-tampon d'entrée-sortie BUFCOM,
- d'un pointeur de pile POINT.

Tous les registres sont composés de 8 bascules à l'exception du registre d'état CC qui est composé de 4 bascules, du registre d'instruction RI composé de 16 bascules, et pouvant de ce fait contenir des instructions d'une taille maximum de 2 octets et du lien L qui est une bascule.

Nous donnons, page suivante, la description du prototype représentant cette unité.

N° de ligne	DESCRIPTEUR
1	Unité CPUTH(BD(8) es, PHI1 ev, PHI2 ev, SYNC co,
2	V ev, T ev, RESET ev, IRQ ev, RW co, SROM co,
3	SRAM co);
4	Déclarations
5	Registre ACC bit(0:7), L bit(0:0), TEMP bit(0:7), etc.
6	mémoire RI(0:1) bit(0:7);
7	entier AQS, CYCIN, IU;
8	format 1 F0(0:7),
9	2 OP(0:7);
10	1 F1(0:15),
11	2 OP(0:7),
12	2 ADR(0:7);
13	Adressage résultat RAD(0:7);
14	sélect
15	$FORMAT(RI) = 'F1' : ADR → RAD;#
16	Fsélect;
17	Description
18	instr LDA : (B'00000001',F1);
19	@. 1 → ACC;
20	Sélect
21	ACC = 0 : 1 → CC(2,1);#
22	fsélect;
23	instr STA : (X'02',F1);
24	ACC → @. 1;
25	Sélect
26	ACC = 0 : 1 → CC(2,1);#
27	fsélect;
28	Fdéclarations;
29	/*DESCRIPTION DU FONCTIONNEMENT INTERNE*/
	⋮

N° de ligne	DESCRIPTEUR
30	Sélect
31	RESET = 0 et AQS = 0 et (IRQ = 0 ou IRQ = nd);
32	activer ACQN; /*ACQUISITION*/#
33	RESET = 0 et AQS = 1 : faire RI; 0 → AQS; /*EXECUTION*/#
34	RESET = 0 et AQS = 0 et CYCIN = 0 et IRQ = 1 et CC(0,1)=0;
35	activer INTER;# /*INTERRUPTION*/
36	RESET = 1 : activer INIT(ACC,L,TEMP,RI,BUFCOM,
37	IX,CO,AQS,RAD,POINT,CC);#
38	/*INITIALISATION*/
39	fsélect;
40	Funité;

COMMENTAIRES :

Ligne 8 : L'unité CPUTH peut exécuter des instructions de 2 formats différents (F0 et F1).

Ligne 13 : Description du processus d'adressage. RAD contient le résultat.

Ligne 18 : Description de l'instruction LDA de code opération B'00000001' et de format F1.

Ligne 19 : Schéma d'instruction.Appel implicite des processus d'adressage (@) et de lecture mémoire (.) à l'adresse obtenue. L'indice qui suit le point sert à identifier l'action qui effectue la lecture-mémoire.

Ligne 21 : Si la condition est vérifiée, la valeur 1 est affectée à la tranche du registre CC commençant en position binaire N°2 et composée d'une position binaire.

Ligne 24 : Appel implicite des processus d'adressage et d'écriture-mémoire. Le contenu de l'accumulateur est rangé à l'adresse ainsi calculée.

Ligne 32 : Le mot-clé activer effectue l'appel d'une action figurant dans le descripteur des actions.
Une action peut comporter ou non des paramètres.

Ligne 33 : L'instruction faire <registre> déclenche l'exécution de l'instruction contenue dans <registre>, c'est-à-dire, en fait, l'interprétation du schéma d'instruction correspondant.

Il faut noter que pour l'utilisateur qui désire utiliser le même processeur dans diverses configurations externes, cette description ne devrait être faite qu'une seule fois par le fabricant (ou le concepteur) du composant et mise à la disposition des usagers en même temps que celui-ci. A partir de cette description, le système PYTHIE permet à l'utilisateur de constituer une bibliothèque de prototypes à partir de laquelle il n'aurait à définir qu'un descripteur des connexions pour composer sa configuration.

III.2.3.- LE DESCRIPTEUR DES ACTIONS.

Les actions, à la différence des unités, ne correspondent pas à des objets ayant une existence physique. La notion d'action est l'équivalent de la notion de procédure dans un langage évolué. Une action peut être appelée, explicitement ou implicitement, par un descripteur de prototype ou par un descripteur d'action.

Dans l'exemple qui suit est illustrée la manière d'écrire un descripteur d'action .

N° de ligne	DESCRIPTEUR
1	Action ACQN; /*ACQUISITION*/
2	entier I,LI;
3	Sélect
4	PHI1 = 1 et PHI2 = 0 :
5	Sélect
6	CYCIN = nd ou CYCIN = 0 :
7	1 → SYNC;
8	CO → BD;
9	0 → RW;
10	0 → SRAM;
11	1 → SROM;
12	1 → CYCIN;#
13	vrai : sélect
14	CYCIN = 2 :
15	BD → RI(0,1);
16	1 → I;
17	$TAILLE(RI) → LI;
18	activer SUITACQ;#
19	vrai : BD → RI(I,1);
20	activer SUITACQ;#
21	fsélect;#
22	fsélect;#
23	PHI1 = 0 et PHI2 = 1 : CO + 1 → CO;
24	CYCIN + 1 → CYCIN;#
25	fsélect;
26	Faction;
27	Action SUITACQ;
28	LI ← UI → LI; I + 1 → I;
29	Sélect
30	LI = 0 : 0 → SROM; 0 → SYNC, 1 → AQS; Attendre(1);
31	vrai : CO → BD; CYCIN + 1 → CYCIN;#
32	fsélect;
33	Faction;

COMMENTAIRES

Ligne 1 : Une action peut comporter ou non des paramètres. Dans le deuxième cas, ceux-ci peuvent être d'entrée, de sortie ou d'entrée et de sortie.
Une action ne peut pas modifier la valeur de ses paramètres d'entrée.

Ligne 2 : Dans une action, on peut déclarer des variables de type entier.

Ligne 17 : La variable du système $TAILLE (<registre>) fournit la longueur en bits de l'instruction contenue dans le <registre> . Il existe d'autres variables du système comme $FORMAT (<registre>) qui fournit le format de l'instruction contenue dans <registre> .

Ligne 27 : La séquence d'instructions contenues dans l'action SUITACQ est invoquée dans deux endroits différents de l'action ACQN.Ceci justifie la création d'une nouvelle action.

Ligne 28 : UI est une variable externe, déclarée dans CPUTH.

Ligne 30 : L'instruction "attendre" met en état d'attente l'unité concernée pendant le nombre d'unités de temps indiqué en paramètre. Il existe une autre forme de cette instruction, la forme attendre ev qui met l'unité en attente jusqu'à l'arrivée d'un événement. La passage à l'unité de temps suivante se fait soit explicitement par l'instruction attendre, soit implicitement par l'absence d'action à effectuer par l'unité.

Nous venons de voir, de manière informelle, comment exprimer les différents descripteurs du système dans le langage LEDA. La définition formelle du langage est donnée en annexe.

C H A P I T R E IV

L'IMPLEMENTATION

C H A P I T R E IV

L'IMPLEMENTATION

Dans les chapitres II et III, on a défini le système PYTHIE et le langage LEDA qui lui est associé.

Les problèmes de leur implémentation sont traités dans ce chapitre.

D'après la définition du système PYTHIE, lorsqu'on parle de son implémentation, on entend notamment celle :

a) de l'ensemble O d'objets qui est, en réalité, une bilbiothèque d'unités prédéfinies.

b) du langage de description L , en l'occurrence LEDA.

c) des primitives de manipulation, c'est-à-dire d'un langage de commande.

d) de l'objet temporaire T .

Les paragraphes qui suivent traitent en détail les problèmes d'implémentation des différents composants du système PYTHIE.

IV.1.- LES BIBLIOTHÈQUES DU SYSTÈME.

Le système PYTHIE comporte 2 bibliothèques : la bibliothèque d'unités et la bibliothèque temporaire.

La bibliothèque d'unités sert au catalogage d'unités prédéfinies et contient tous les renseignements nécessaires à leur mise en oeuvre. Cette bibliothèque est en fait composée de 3 bibliothèques :

- LA BIBLIOTHEQUE DE PROTOTYPES qui contient tous les renseignements relatifs aux unités prédéfinies (prototypes) à l'exception de la description du fonctionnement des actions appelées à partir de certains prototypes.

- LA BIBLIOTHEQUE D'ACTIONS qui contient la description du fonctionnement des actions utilisées par un ou plusieurs prototypes.

- LA BIBLIOTHEQUE DE SCHEMAS D'INSTRUCTIONS dans laquelle sont conservés tous les renseignements nécessaires à l'interprétation des schémas d'instructions, décrivant les instructions du langage-machine des prototypes de type processeur.

La bibliothèque temporaire sert à conserver tous les renseignements concernant la structure physique et logique d'une configuration décrite par le descripteur des connexions d'une description en LEDA. Elle permet à l'usager de stocker des configurations en cours d'étude, à la manière dont sont stockés les programmes dans un système conventionnel.

Le détail de l'organisation des bibliothèques du système est donné en annexe.

IV.2.- LE LANGAGE LEDA.

L'énoncé des objectifs du système PYTHIE impliquait que le langage LEDA ne pouvait pas se limiter à la description de la structure et du fonctionnement des unités. Il fallait aussi que cette description puisse déboucher sur une simulation permettant de tester un fonctionnement.

Ainsi, se pose le problème de l'implémentation du langage LEDA.

On peut envisager 3 possibilités :

- la solution compilée,
- la solution interprétée,
- la solution de génération d'un programme en langage évolué.

Essayons de cerner les avantages et les inconvénients respectifs des 3 solutions.

1) LA SOLUTION COMPILEE

Elle suppose l'écriture d'un programme (compilateur) qui, à partir du texte source, génère du langage machine pour une gamme d'ordinateurs donnée. Le code généré sera ensuite exécuté par le système de l'ordinateur.

L'avantage principal de cette solution est la rapidité à l'exécution puisque le code généré est exécuté directement par le système.

Parmi ses inconvénients, on peut citer :

a) Son faible degré de portabilité au niveau du code généré : ce code, étant composé d'instructions machine d'une gamme d'ordinateurs donnée, il ne peut être exécuté que sur un ordinateur de cette gamme.

b) La difficulté de garder le contrôle à l'exécution, celle-ci étant assurée directement par le système à partir du code généré. Ceci a pour conséquence une grande lourdeur si l'on souhaite un système interactif. En particulier, dans le cas du langage LEDA, si l'utilisateur doit changer de configuration, il doit recompiler.

c) La difficulté de mise au point.

2) LE SOLUTION INTERPRETEE.

Cette solution suppose l'écriture de 2 programmes (qui peuvent n'en faire qu'un seul) : un analyseur syntaxique générateur qui, à partir d'un texte en langage-source, génère une chaîne codée écrite dans un code intermédiaire entre le langage-source et le langage-machine; un programme interpréteur qui interprète la chaîne codée.

Les principaux avantages de cette solution sont la portabilité et la souplesse à l'exécution. En effet, le code à générer n'étant pas lié à un langage-machine donné, les deux parties de l'interpréteur peuvent être écrites dans un langage évolué suffisamment universel pour que le changement de type d'ordinateur ne pose pas de problème majeur. D'autre part, l'exécution étant assurée par le programme interpréteur, l'utilisateur peut plus facilement intervenir durant celle-ci et modifier son cours.

L'inconvénient majeur de la solution interprétée est sa lenteur à l'exécution qui est toutefois moins gênante s'agissant d'un système conversationnel.

3) LA SOLUTION DE GENERATION DE LANGAGE EVOLUE.

Cette solution suppose l'écriture d'une analyseur syntaxique générateur qui, à partir d'un texte en langage-source, génère un programme écrit dans un langage évolué. Ce programme sera ensuite compilé par le compilateur du langage et exécuté par le système.

Parmi les avantages de cette solution, on peut citer la facilité d'écriture et la portabilité, le générateur pouvant être écrit dans un langage évolué. Les inconvénients sont nombreux : Difficulté de mise au point, lenteur, perte de contrôle à l'exécution.

En plus des trois solutions exposées, on peut enfin imaginer des solutions mixtes..

La solution retenue :

Les choix faits lors de la définition du système PYTHIE (Chap. II) impliquent un certain nombre de contraintes et en particulier la possibilité donnée à l'utilisateur d'intervenir à tout moment pour

- modifier sa configuration,
- modifier un prototype ou une action,
- obtenir une représentation graphique,
- donner des valeurs initiales ou changer les paramètres d'exploitation,
- activer la configuration,
- etc...

et ceci autant de fois qu'il s'avère nécessaire pour tester complétement son système.

Il faut donc un système conversationnel, ce qui écarte, en principe, la solution compilée et celle de génération de langage

évolué pour lesquelles l'aspect interactif n'est pas naturel. Des essais sur ordinateur d'un générateur de programmes PL/I ont permis de mieux voir les inconvénients de cette solution.

C'est ainsi que la solution finalement retenue a été celle d'écriture d'un interpréteur.

IV.2.1.- L'interpréteur.

L'interpréteur du langage LEDA est composé de 2 parties.
La première partie vérifie la conformité syntaxique de la description en LEDA et catalogue, d'une part, l'objet décrit par le descripteur des connexions dans la bibliothèque temporaire et d'autre part, les objets décrits par les descripteurs des prototypes et des actions dans les bibliothèques appropriées.

A l'issue de cette première partie, l'utilisateur peut déjà effectuer un certain nombre de tests de correction ou tenter certaines opérations sur la configuration cataloguée par l'intermédiaire des primitives du système.

La seconde partie simule le fonctionnement de l'objet contenu dans la bibliothèque temporaire et réalise en fait 2 fonctions :

- L'édition des liens entre la configuration de la bibliothèque temporaire d'une part, les prototypes et les actions actives d'autre part, comportant de nouveaux tests de conformité.

- La simulation du fonctionnement de la configuration de la bibliothèque temporaire par interprétation de la chaîne codée résultat de la première partie et de l'édition de liens.

IV.2.1.1.- Le générateur de chaîne codée

Le générateur de chaîne codée constitue la première partie de l'interpréteur LEDA. En fait, la génération de chaîne codée n'a lieu que pour les descripteurs des prototypes et des actions. La chaîne codée générée, est cataloguée dans les bibliothèques des prototypes et des actions respectivement. L'unité composée décrite par le descripteur des connexions est cataloguée dans la bibliothèque temporaire sous la forme décrite en annexe.

Comme la forme et l'utilisation des différentes tables contenant les objets internes ou externes et les arguments formels sont explicitées en annexe, on se bornera ici à décrire la chaîne codée générée pour chacune des instructions de LEDA.

1) L'instruction de transfert.

L'instruction de transfert est traduite en forme post-fixée. Ainsi, l'instruction

A(2,3) + B(4,1) → C & D(2,1)

donnera lieu à la génération de la chaîne codée :

A 2 3 (T) B 4 1 (T) + C D 2 1 (T) & →

où (T) est l'opérateur de tranche.

Il est évident que la forme de l'instruction de transfert LEDA (transfert de gauche à droite) entraîne une certaine perte d'efficacité au niveau de l'interprétation mais nous estimons que les avantages de cette forme compensent cet inconvénient.

2) L'instruction sélective

Les conditions de l'instruction sont traduites en forme post-fixée, suivie de l'opérateur (F) (branchement si faux) et de l'adresse de branchement.

La traduction de la liste d'instructions qui suit une condition est suivie de l'opérateur (B) (branchement inconditionnel) et de l'adresse de branchement qui est celle de la fin de l'instruction sélective.

L'insertion des adresses de branchement dans la chaîne codée se fait, à la compilation, par une pile.

Ainsi, l'instruction sélective :

Select

Cond1 : li_1 #
Cond2 : li_2 #
⋮
$Cond_n$: li_n #

fselect

où li est une abréviation pour liste d'instructions, donnera lieu à la génération de la chaîne codée :

Select cond1 (F) C_2 li_1 (B) C_f Cond2 (F) C_3 li_2 (B) C_f

↑ C_2

... $cond_n$ (F) C_{n+1} li_n (B) C_f fselect

↑ C_n ↑ C_{n+1} ↑ C_f

3) L'instruction d'activation

L'instruction :

activer ACTION1(p_1,p_2,...p_n)

donne lieu à la génération de la chaîne codée :

activer ACTION 1 p_1 p_2 ... p_n ↓

où ↓ est l'opérateur d'activation.

4) L'instruction ATTENDRE

La chaîne codée générée a la forme :

expression attendre

ou la forme :

expression d'événement attendre ev

selon la variante de l'instruction ATTENDRE.

5) L'instruction FAIRE

Chaîne codée générée :

objet de registre faire

6) L'instruction de répétition

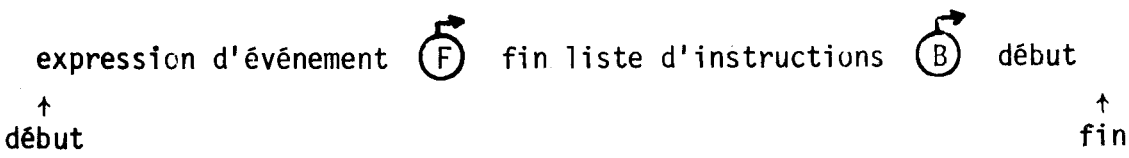
L'instruction de répétition

jusque ev Expression d'événement :

liste d'instructions

recommencer

est transformée en :



7) Les schémas d'instructions.

Tout ce qui vient d'être exposé s'applique aussi aux schémas d'instructions. La seule différence réside dans la traduction des appels implicites de la fonction d'adressage ainsi que des actions d'accès mémoire.

Ainsi, la notation

@ m.n

sera traduite par

m@ n (L)

si elle est rencontrée en partie gauche d'une instruction de transfert et par

m@ n (E)

si elle est rencontrée en partie droite, où (L) et (E) sont, respectivement, l'opérateur de lecture et l'opérateur d'écriture de mémoire.

IV.2.1.2.- L'interpréteur de chaîne codée.

Le but de cette partie de l'interpréteur est la simulation du fonctionnement de la configuration contenue dans la bibliothèque temporaire.

Pour cela, il faut simuler le fonctionnement de chacune des unités simples de la configuration en interprétant la chaîne codée qui décrit le fonctionnement des prototypes correspondants.

Pour les besoins de la simulation, le temps continu de fonctionnement est découpé en unités de temps. Durant chaque unité de temps, les unités composant la configu-

ration sont supposées fonctionner en parallèle.

Lorsque le fonctionnement de plusieurs unités simples est décrit par le même prototype ou lorsque le même unité apparaît plusieurs fois dans la configuration, il faut dupliquer plusieurs fois la table des objets internes du prototype en question. Pour les paramètres, ceci n'est pas nécessaire car leur transmission se fait par nom. Le changement de valeur des paramètres effectifs de sortie, d'entrée et sortie et de commande ne devient effectif qu'à la fin de chaque unité de temps.

Le nom d'un prototype, lorsqu'il décrit le fonctionnement de plusieurs unités, est aussi dupliqué dans un vecteur appelé vecteur d'état des prototypes.

Quant à la chaîne codée décrivant le fonctionnement d'un prototype, elle n'est pas dupliquée mais un pointeur dans le vecteur d'état des prototypes.

Quant à la chaîne codée décrivant le fonctionnement d'un prototype, elle n'est pas dupliquée mais un pointeur dans le vecteur d'état des prototypes indique, pour chaque exemplaire du même prototype, à quel emplacement de la chaîne codée doit reprendre la simulation à l'unité de temps suivante.

Outre le vecteur d'état des prorotypes, le simulateur utilise un certain nombre de piles et de tables dont la signification est donnée ici brièvement :

- la table d'état des prototypes contient l'état courant des objets internes de tous les exemplaires de prototypes. Le premier élément de la table sert à gérer l'espace-mémoire du système.
- la table de formats temporaires contient les valeurs courantes des formats de tous les exemplaires de prototypes,
- la table d'état temporaire d'actions contient l'état courant des objets internes des actions pour chaque exemplaire de prototype,
- la pile d'évaluation est en fait un ensemble de piles associées chacune à un exemplaire de prototype. Elle sert à l'interprétation des instructions des prototypes,
- la pile d'état des actions est aussi un ensemble de piles associées chacune à un exemplaire de prototype. Elle sert à stocker certains renseignements concernant les actions en cours d'exécution.

Un élément du vecteur d'état des prototypes a la structure suivante :

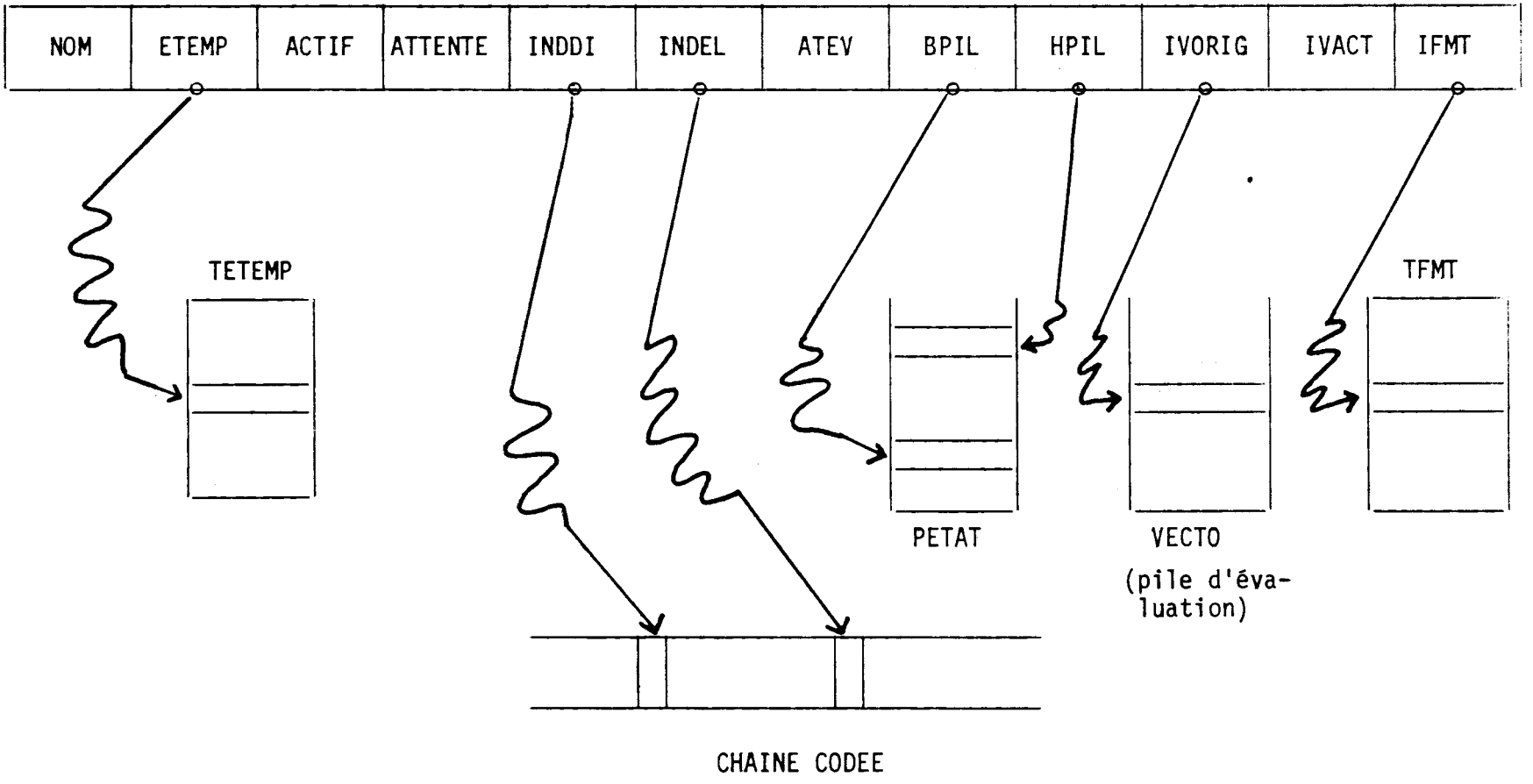


FIGURE IV-4

où : NOM contient le nom d'un prototype,

ETEMP est un pointeur d'accès à la table d'état temporaire contenant l'état courant des objets internes de tous les exemplaires de prototypes,

ACTIF contient une des valeurs booléennes vrai ou faux suivant que le prototype (ou l'exemplaire de prototype) est activé ou non.

ATTENTE indique le nombre d'unités de temps pendant lesquelles le prototype restera en attente. ATTENTE est positionné par l'instruction ATTENDRE. Sa valeur est décrémentée de 1 à chaque activation du prototype.

INDDI indique à tout moment l'adresse dans la chaîne codée du début de l'instruction en cours d'exécution.

INDEL contient à tout moment l'adresse de l'élément courant dans la chaîne codée.

ATEV contient l'une des valeurs booléennes vrai ou faux. Sa valeur est positionnée à l'issue de l'exécution d'une instruction ATTENDRE EV

BPIL et HPIL indiquent respectivement le début et l'élément courant de la pile affectée au prototype dans la pile d'état des actions.

IVORIG et IVACT indiquent respectivement le début et l'élément courant de la pile affectée au prototype dans la pile d'évaluation.

IFMT est une clé d'accès à la table de formats temporaires contenant les valeurs temporaires des formats du prototype.

La table d'état temporaire est composée d'éléments dont la structure est la suivante :

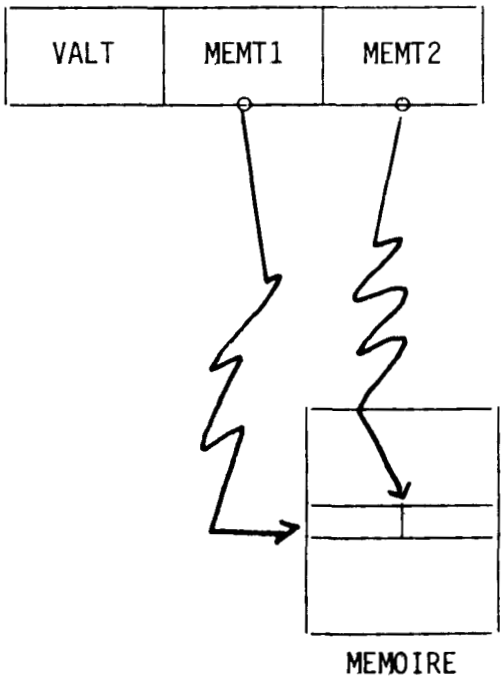


FIGURE IV-5

où : VALT contient la valeur temporaire d'un objet interne d'un prototype ou d'un exemplaire de prototype.

MEMT1 et MEMT2 contiennent des informations servant a déterminer l'adresse d'origine de représentation d'un objet de type mémoire dans l'espace mémoire du système.

.../...

La pile d'état des actions est en fait un ensemble de piles servant à empiler les noms des actions appelées par les prototypes ainsi que certains renseignements les concernant. Chaque pile de l'ensemble est affectée à un exemplaire de prototype contenu dans le vecteur d'état des prototypes. Il est évident que cette organisation en pile est nécessaire, une action pouvant en activer une autre. Chaque élément de la pile a la structure suivante :

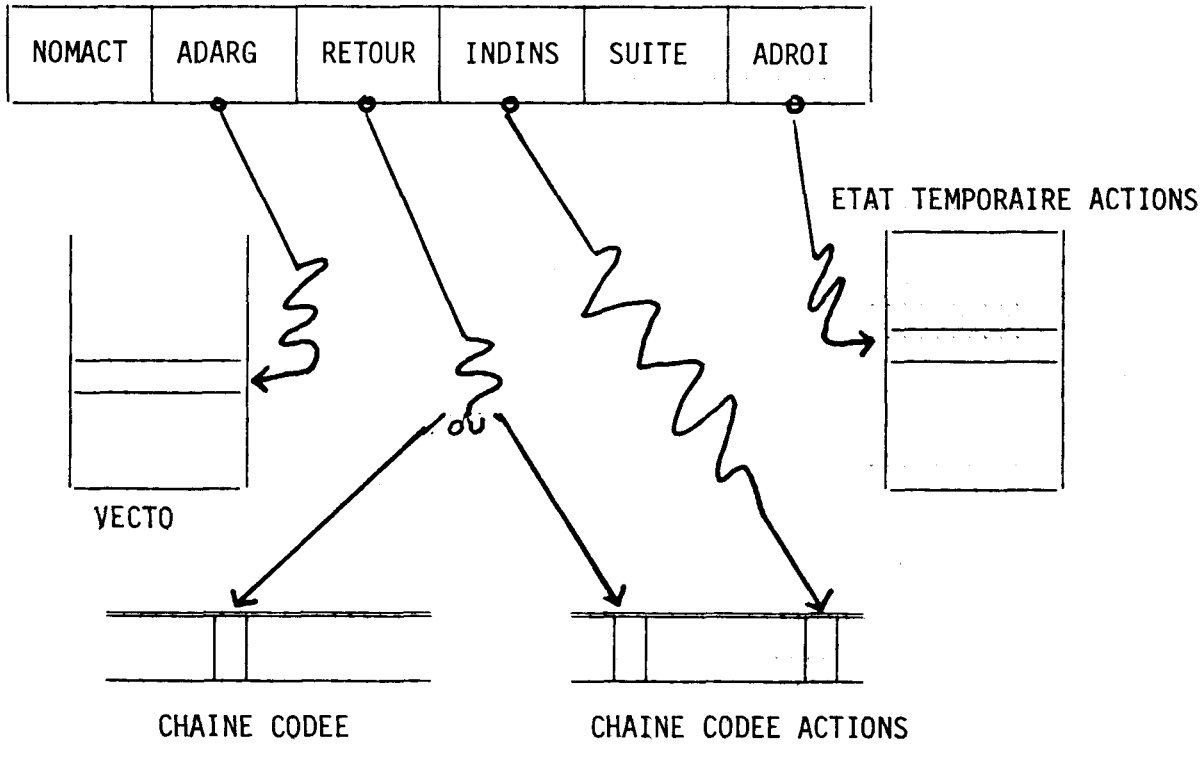


FIGURE IV-6

où NOMACT contient le nom de l'action en cours de traitement,

ADARG contient l'adresse des arguments de l'action dans la pile d'évaluation,

RETOUR contient l'adresse de retour dans la chaîne codée ou dans la chaîne codée des actions,

INDINS contient l'adresse de l'instruction en cours d'exécution dans la chaîne codée des actions

SUITE contient l'une des valeurs booléennes vrai ou faux pour indiquer que le traitement de l'action se poursuivra à l'unité de temps suivante,

ADROI contient l'adresse d'origine des objets internes de l'action dans une table d'état temporaire d'actions contenant les valeurs actuelles des objets internes de chaque exemplaire d'actions.

La pile d'évaluation est en fait un ensemble de piles dont chacune est associée à un exemplaire de prototype. Elle sert à interpréter les instructions des prototypes à partir de leur description dans la chaîne codée et des valeurs actuelles des objets manipulés par ces instructions. Un élément de la pile d'évaluation a la structure suivante :

VCOD	VRB	VLB	VVAL	VTYP	VRM	VLM	VCAT

FIGURE IV-7

où VCOD contient le code d'un objet ou la valeur 0 s'il s'agit d'un résultat intermédiaire,

VRB contient le rang du premier bit d'un objet de type branche de borne ou de registre ou d'un objet de type tranche de tranche de mémoire,

VLB contient la longueur en bits d'un objet de type branche de borne ou de registre ou d'un objet de type tranche de tranche de mémoire,

VVAL contient la valeur actuelle d'un objet ou d'un résultat intermédiaire,

VTYP indique le type de l'objet empilé,

VRM contient le rang de premier mot d'un objet de type tranche de mémoire,

VLM contient la longueur en mots d'un objet de type tranche de mémoire,

VCAT contient l'une des valeurs booléennes vrai ou faux pour indiquer qu'une opération de composition est à effectuer avec l'élément précédent de la pile.

IV.3.- LES PRIMITIVES DE MANIPULATION.

Les primitives du système PYTHIE se composent d'une primitive principale et de 7 primitives secondaires.

La primitive principale (CREER) permet d'initialiser le système PYTHIE, c'est-à-dire de mettre quelque chose dans se bibliothèques rendant ainsi applicables les primitives secondaires.

Les primitives secondaires permettent de tirer des renseignements à partir d'une analyse statique des connexions et soit d'agir sur des unités en fonction de ces renseignements, soit d'en proposer une interprétation.

LA PRIMITIVE "CREER".

Elle permet de "créer" des unités à partir de leur description en LEDA.

En réponse à cette primitive, le système :

a) Reconnaît syntaxiquement la configuration décrite par le descripteur des connexions.

b) Vérifie statiquement la conformité des connexions.

c) Catalogue la configuration décrite, dans la bibliothèque temporaire.

d) Le cas échéant, il reconnaît syntaxiquement les objets décrits par le descripteur des prototypes et le descripteur des actions vérifie statiquement la conformité de leur fonctionnement, produit de la chaîne codée et catalogue les résultats dans les bibliothèques appropriées (bibliothèque de prototypes et bibliothèque d'actions).

LES PRIMITIVES SECONDAIRES

1) La primitive AJOUTER

Elle permet de créer un ptototype (sous forme de chaîne codée) à partir de la configuration contenue dans la bibliothèque temporaire.

En réponse à la primitive AJOUTER, le système :

a) Crée, à partir de la configuration contenue dans la bibliothèque temporaire et des prototypes associés, un prototype ayant le même fonctionnement que l'ensemble de la configuration.

b) Catalogue le prototype ainsi créé, dans la bibliothèque des prototypes, sous le nom de la configuration d'origine.

Afin que le prototype ainsi créé conserve le fonctionnement des prototypes composants, toute broche de sortie d'un d'entre eux qui est broche d'entrée d'un autre, est remplacée, dans le nouveau prototype, par un couple de registres et par une instruction de transfert. Le premier registre du couple représente la broche et est remplacé par le second dans toutes les parties droites des instructions de transfert où il apparaît. Quant à l'instruction de transfert, elle range la valeur du second registre dans le premier à la fin de chaque unité de temps. L'utilisation de 2 registres est nécessaire car la valeur d'une broche de sortie d'un prototype de la configuration d'origine, modifiée à l'instant t_i n'est disponible pour les autres prototypes qu'à l'instant $t_i + 1$.

Ainsi, pour la configuration :

```
Connexions;
     1 Unité ESSAI,
            liaisons L1, L2 co, L3 ev;
       2 Unité UNIT1(L1 co, L3 ev) def U1;
       2 Unité UNIT2(L1 ev, L2 co) def U2;
Fconnexions;
```

où les connexions U1 et U2 sont définies par :

```
Prototypes;
     Unité U1(L11 co, L31 ev);
          Sélect
              L31 = 1 : non L11 → L11; #
          fsélect;
     Funité
     Unité U2(L12 ev, L22 co);
          non L12 → L22;
     Funité
FPrototypes;
```

après application de la primitive AJOUTER, l'on obtient le prototype :

```
     Unité ESSAI(L2 co, L3 ev);
          Déclarations;
             registre R1 bit(0:0), R2 bit(0:0);
          Fdéclarations
          Sélect
             L3 = 1 : non L1 → R2;#
          Fsélect;
          non R1 → L2;
          R2 → R1;
     Funité
```

Dans l'avenir, il serait souhaitable que la primitive AJOUTER soit révisée afin que le nouveau prototype catalogué positionne correctement ses broches de sortie sans que son fonctionnement soit nécessairement identique à l'ensemble des fonctionnements de ses composants. Ceci permettrait de simplifier les prototypes ainsi créés et de réduire, de ce fait, le temps de simulation. Ceci n'est pour l'instant qu'un souhait car il n'est pas évident que ce problème, qui est un problème de simplification de fonctions booléennes, admette une solution satisfaisante.

2) La primitive SUPPRIMER

Elle permet d'effacer un prototype ou une action dans les bibliothèques correspondantes. Elle est utilisée sous la forme :

SUPPRIMER liste d'objets avec type

où une liste d'objets avec type est une suite de doublets

type d'objet nom d'objet

où type d'objet = { P s'il s'agit d'un prototype ; A s'il s'agit d'une action

Pour l'instant, la primitive SUPPRIMER se contente de supprimer la référence à l'action ou au prototype concerné, dans la table des matières de la bibliothèque correspondante, sans récupérer la place occupée par l'objet lui-même.

Il est envisagé de lui associer ultérieurement un programme de gestion permettant de récupérer la place inutilisée lorsqu'une bibliothèque est pleine.

3) La primitive INITIALISER

Elle permet de donner des valeurs initiales aux liaisons d'une configuration ou aux objets internes (registres, mémoires, entiers) d'un exemplaire de prototype. Elle permet, entre autres, de charger, dans la mémoire d'une unité de type mémoire, un programme en langage-machine exécutable pour une unité de type processeur. Sa forme syntaxique est la suivante :

INITIALISER liste d'initialisations

où une initialisation s'écrit sous la forme :

LI liste d'objets de liaison initialisés

ou sous la forme :

OI nom de prototype n° d'exemplaire liste d'objets internes initialisés

où LI et OI désignent respectivement un objet de liaison ou un objet interne. Le n° d'exemplaire est nécessaire lorsque le

fonctionnement de plusieurs unités est décrit par le même prototype. Un objet initialisé est une séquence composée de

nom ou tranche	objet = constante	décimale / binaire / hexadécimale

La primitive INITIALISER est extrêmement puissante et permet de modifier la valeur de n'importe quel objet. Il appartient donc à l'utilisateur de s'assurer de la cohérence des initialisations demandées.

4) La primitive ACTIVER

Elle permet d'activer la configuration de la bibliothèque temporaire, c'est-à-dire de simuler son fonctionnement à partir de ceux de ses unités composantes, dans la bibliothèque des prototypes. Sa forme syntaxique est la suivante :

ACTIVER durée d'activation

Par durée d'activation, on entend la durée de simulation du fonctionnement de la configuration en unités de temps d'horloge de simulation.

Durant la simulation, les exemplaires des prototypes représentant les unités de la configuration sont exécutés les uns après les autres mais les changements de valeur des bornes de sortie (en fait des bornes de type s , es ou co) ne deviennent effectifs qu'à la fin de chaque unité de temps, après l'activation de tous les exemplaires de prototypes. La fin de l'activation d'une unité, c'est-à-dire la fin de l'unité de temps pour l'unité, est indiquée par l'unité elle-même, soit explicitement, par la présence de l'instruction ATTENDRE, soit implicitement, par l'absence d'autres instructions à exécuter.

5) La primitive MODIFIER

Elle permet en fait de reconfigurer l'unité composée de la bibliothèque temporaire en remplaçant une ou plusieurs de ses unités composantes par d'autres unités, simples ou composées, décrites, sous forme de descripteur de connexions, en LEDA. Elle est utilisée sous la forme :

MODIFIER	modification

où une modification est composée de :

nom de l'unité à remplacer	description de l'unité remplaçante

La conformité syntaxique et sémantique de la description de l'unité remplaçante est vérifiée et la nouvelle unité prend la place de l'ancienne dans la bibliothèque temporaire.

6) La primitive RESTRUCTURER

Cette primitive autorise le système à effectuer des regroupements dans les ensembles d'unités-frères d'un niveau donné, à partir de la configuration définie par l'utilisateur et contenue dans la bibliothèque temporaire.

Le critère de ces regroupements est la notion de maître absolu définie dans le Chapitre II. La forme syntaxique de la primitive RESTRUCTURER est la suivante :

RESTRUCTURER	noeud

où noeud est un nom d'unité dont le système essaiera de restructurer l'ensemble des fils.

En réponse à la commande RESTRUCTURER, le système construit dans un premier temps le graphe des maîtres pour l'ensemble des unités-frères du noeud indiqué et ceci à partir de la caractérisation des broches par l'utilisateur (ev, co, etc..). A partir de ce graphe, le système détermine les ensembles les plus larges possibles admettant un maître absolu et effectue des regroupements logiques d'unités suivant le principe énoncé au Chapitre II. Ces regroupements correspondent en fait à la création de nouvelles unités et à l'introduction de niveaux logiques supplémentaires dans l'arbre représentant la configuration à restructurer. Le système génère des noms pour les unités ainsi créées et définit leurs broches. La configuration ainsi restructurée remplace l'ancienne dans la bibliothèque temporaire. Un exemple de restructuration se trouve à la fin du chapitre II.

Les restructurations proposées par le système ont pour objet d'indiquer à une catégorie d'utilisateurs, concepteurs ou étudiants, des possibilités de suppression de connexions intermédiaires et d'optimisation du nombre de connexions entre cartes, par regroupement de certaines unités sur la même carte, ce qui, appliqué à la configuration réelle, la rendrait, en principe, plus efficace. En revanche, le fonctionnement de l'unité simulée par PYTHIE n'est en rien modifié par la restructuration.

7) La primitive DESSINER

Elle permet d'avoir, sur papier (éventuellement sur écran), une représentation graphique simple de la configuration proposée par l'utilisateur et contenue dans la bibliothèque temporaire, ou d'une de ses sous-unités. Sa forme syntaxique est la suivante :

DESSINER nom

où nom indique le nom de l'unité à dessiner.

La primitive fournit comme résultat, d'une part une représentation graphique de l'unité selon le principe exposé en II.5 et, d'autre part, une liste des sous-unités d'un niveau quelconque dont les descendants n'ont pu être représentés, faute de place. Ainsi, l'utilisateur a la possibilité d'appliquer la primitive DESSINER, en connaissance de cause, le nombre de fois nécessaires à la représentation de la totalité de l'unité.

IV.4.- LE PROCESSUS DE CONCEPTION À L'AIDE DE PYTHIE.

La conception d'une architecture à l'aide de PYTHIE peut se faire en plusieurs étapes, en utilisant les primitives existantes.

L'utilisation de ces primitives peut se faire dans un ordre quelconque, à l'exception toutefois de la primitive CREER qui transforme une description en langage-source en une forme manipulable par l'ordinateur et qui doit, de ce fait, précéder toutes les autres. Le processus de conception est illustré schématiquement dans la figure IV-8 (voir page suivante)

.../...

DESCRIPTION SOURCE
CREER
DESCRIPTION SOUS FORME INTERNE
REPRESENTATION GRAPHIQUE
AFFICHAGE DES RESULTATS
DESSINER
RESTRUCTURER
MODIFIER
INITIALISER
ACTIVER
AJOUTER
SUPPRIMER

FIGURE IV-8

L'ordre d'application des primitives sur la description sous forme interne est quelconque et les flèches retournant vers la description sous forme interne indiquent que les primitives soit ne modifient pas la représentation interne, soit elles remplacent par une autre, sur laquelle toutes les primitives sont à nouveau applicables. Certaines de ces primitives fournissent par ailleurs des résultats destinés à guider l'utilisateur, tels que la représentation graphique d'une unité simple ou composée et l'affichage de résultats intermédiaires d'une simulation.

Ainsi, une séquence plausible d'utilisation des primitives du système pourrait être :

/*PHASE DE CONSTRUCTION*/

1)	CREER	création d'une configuration à partir d'une description externe et sortie éventuelle d'une liste d'erreurs
2)	DESSINER	sortie de la représentation graphique de la configuration
3)	MODIFIER	Reconfiguration de l'unité de la bibliothèque temporaire
4)	DESSINER	Nouvelle représentation graphique
5)	RESTRUCTURER	Restructuration de la configuration par le système
6)	DESSINER	Représentation graphique de l'unité restructurée

/*PHASE D'EXPERIMENTATION*/

7)	INITIALISER	Initialisation de la configuration
8)	ACTIVER	Activation durant m unités de temps
9)	ACTIVER	Activation durant n unités de temps
10)	INITIALISER	Initialisation avec des valeurs différentes
11)	ACTIVER	Activation durant k unités de temps

/*PHASE DE SAUVEGARDE*/

12)	AJOUTER	Création d'un nouveau prototype à partir de la configuration de la bibliothèque temporaire dont la conformité vient d'être testée, et catalogage dans la bibliothèque de prototypes.

En utilisant une extension du langage LEDA, le fonctionnement du système PYTHIE pourrait s'écrire comme suit :

```
Action PYTHIE;
        chaîne COMMANDE;
        AFFICHER('SYSTEME PYTHIE ACTIVE');
        LIRE(COMMANDE);
        jusque ev COMMANDE = 'AR';
            sélect
               COMMANDE = 'CREER' : activer CREER;#
               COMMANDE = 'AJOUTER' : activer AJOUTER;#
               COMMANDE = 'SUPPRIMER' : activer SUPPRIMER;#
               COMMANDE = 'INITIALISER' : activer INITIALISER;#
               COMMANDE = 'MODIFIER' : activer MODIFIER;#
               COMMANDE = 'RESTRUCTURER' : activer RESTRUCTURER;#
               COMMANDE = 'DESSINER' : activer DESSINER;#
            fsélect;
              LIRE(COMMANDE);
        recommencer;
        AFFICHER('SYSTEME PYTHIE DESACTIVE');
Faction;
```

IV.5.- LE LANGAGE DE PROGRAMMATION.

Pour implémenter le système PYTHIE, on avait le choix entre deux possibilités : soit écrire tous les programmes en un langage proche d'un langage-machine, par exemple un langage d'assemblage, ce qui assurerait au système une plus grande efficacité, soit les écrire en langage évolué avec pour conséquence une baisse importante des performances mais, en contrepartie, une plus grande portabilité. Une troisième possibilité qui consisterait à écrire une partie des programmes en langage évolué et l'autre partie en langage d'assemblage, n'a pas été jugée satisfaisante car elle n'améliorait pas beaucoup de degré de portabilité de l'ensemble alors que la baisse des performances serait sensible par rapport à la programmation en langage d'assemblage.

La portabilité a semblé un élément suffisamment important pour justifier le choix d'un langage évolué.

Le PL/I présente les avantages d'une grande diffusion et d'une bonne aptitude à supporter les différents types d'information nécessaires (booléens, chaînes, structures, vecteurs, pointeurs). Ainsi tous les programmes d'implémentation de PYTHIE ont été écrits en PL/I et testés sur l'ordinateur 360/40 du Département Informatique de l'I.U.T. de LILLE-I.

IV.6.- CONCLUSIONS.

Le système PYTHIE, tel qu'il est défini actuellement, rend-il à l'utilisateur les services attendus ? Il est encore trop tôt pour répondre à cette question et il est probable que le système devra évoluer à court terme. Ainsi, la version actuelle n'est en aucun cas sa version définitive. Ceci dit, le système PYTHIE offre un certain nombre d'apports qui semblent originaux par rapport aux systèmes existants. Ces apports si situent aussi bien au niveau des primitives qu'à celui du langage de description.

Parmi les apports originaux au niveau des primitives, on peut citer la primitive RESTRUCTURER qui permet de changer la structure logique d'une configuration suivant la relation maître-esclave existant entre ses unités composantes et la primitive DESSINER qui, outre les connexions entre les unités composantes d'une configuration, permet de visualiser leur structure logique.

Quant au langage LEDA, qui sert de langage de description, il permet, grâce au descripteur des connexions et à la notion de schéma d'instruction introduite dans les descriptions de prototypes, d'associer aux avantages des langages modulaires de description, certains aspects intéressants de PMS et de ISP. Ainsi, LEDA peut espérer couvrir, du moins partiellement, les quatre premiers niveaux de la classification de BARBACCI présentée en I.5. Il répond d'autre part de manière satisfaisante aux critères d'universalité et de clarté. En revanche, comme tous les langages procéduraux, il ne possède pas la propriété de séparabilité.

En outre, il a été, autant que possible, tenu compte dans les choix et les propositions essentiels, d'une bonne compatibilité avec les découpages physiques rencontrés dans les micro ordinateurs (boîtiers, modules, groupes de modules identiques, et cartes...).

BIBLIOGRAPHIE

[1] M. BARBACCI — *"A COMPARISON OF REGISTER TRANSFER LANGUAGES FOR DESCRIBING COMPUTERS AND DIGITAL SYSTEMS"*
I.E.E.E. TRANSACTIONS ON COMPUTERS
Vol. C-24, n° 2, février 1975

[2] BARBACCI, BARNES, CATTELL, SIEWIOREK
"THE I.S.P.S. COMPUTER DESCRIPTION LANGUAGE"
CARNEGIE-MELLON UNIVERSITY, 6 Mars 1978.

[3] BARBACCI, BELL, SIEWIOREK
"A NOTATION TO DESCRIBE COMPUTER STRUCTURES"
COMPUTER, Mars 1973

[4] BARBACCI, BELL, SIEWIOREK
"I.S.P. : A NOTATION TO DESCRIBE A COMPUTER'S INSTRUCTION SETS"
COMPUTER, Mars 1973

[5] BARBACCI, NAGLE — *"AN I.S.P.S. SIMULATOR"*
CARNEGIE-MELLON UNIVERSITY, 7 Mars 1978.

[6] BARBACCI, SIEWIOREK
"APPLICATIONS OF AN I.S.P. COMPILER IN A DESIGN AUTOMATION LABORATORY"
1975 INTERNATIONAL SYMPOSIUM ON C.H.D.L. AND THEIR APPLICATIONS. PROCEEDINGS

[7] BARBACCI, SIEWIOREK, GORDON, HOWEBRIGG, ZUCKERMAN
"AN ARCHITECTURAL RESEARCH FACILITY, I.S.P. DESCRIPTIONS, SIMULATION, DATA COLLECTION"
NATIONAL COMPUTER CONFERENCE, 1977

[8] BELL, NEWELL — *"COMPUTER STRUCTURES : READINGS EXAMPLES"*
Mc GRAW-HILL, 1971

[9] H. BERNDT — *"THE LINGUISTIC DEFINITION OF COMPUTER ARCHITECTURES AND IMPLEMENTATIONS"*
INTERNATIONAL WORKSHOP ON COMPUTER ARCHITECTURE. GRENOBLE, Juin 1973

[10] D. BORRIONE — *"LASCAR : UN LANGAGE POUR LA SOMULATION ET L'EVALUATION DES ARCHITECTURES D'ORDINATEURS"*
THESE 3ème CYCLE, GRENOBLE, 16 Avril 1976

[11] D. BORRIONE — *"DESCRIPTION ET SIMULATION D'UNE ARCHITECTURE MULTI-PROCESSEUR A L'AIDE DU LANGAGE LASCAR"*
R.R. n° 87, GRENOBLE, Janvier 1977

[12] BORRIONE, GRABOWIECKI — *"PRESENTATION DE QUELQUES ASPECTS DU LANGAGE LASSO"*
BIGRE, n° 6, Juin 1977

[13] BORRIONE, GRABOWIECKI — *"DEFINITION D'UN LANGAGE POUR LA SOMULATION DE SYSTEMES INFORMATIQUES : LASSO".*
R.R., n° 122, GRENOBLE, Juin 1978

[14] BURSTON, KINNIMENT, HILLARY KAHN — *"A DESIGN LANGUAGE FOR ASYNCHRONOUS LOGIC"*
THE COMPUTER JOURNAL, Vol 21, n° 2
Novembre 1978.

[15] S.S. CHING, J.H. TRACEY — *"AN ITERACTIVE COMPUTER GRAPHICS LANGUAGE FOR THE DESIGN AND SIMULATION OF DIGITAL SYSTEMS".*
COMPUTER, Vol. 10, n° 6, Juin 1977

[16] Y. CHU — *"AN ALGOL-LIKE COMPUTER DESIGN LANGUAGE"*
C.A.C.M., Vol. 8, n° 10, Octobre 1965

[17] Y. CHU — *"INTRODUCING THE COMPUTER DESIGN LANGUAGE"*
I.E.E.E. COMP. CONF., COMPCON 72,
Septembre 1972

[18] Y. CHU — *"INTRODUCING C.D.L."*
COMPUTER, Décembre 1974

[19] V. CORDONNIER — *"ARCHITECTURE DES ORDINATEURS"*
SUPPORT DE COURS. ECOLE D'ETE D'INFORMATIQUE DE L'A.F.C.E.T., LANNION, Juillet 1976

[20] DAHL, NYGAARD — *"SIMULA - AN ALGOL BASED SIMULATION LANGUAGE"*
C.A.C.M., Vol. 1, n° 1, Septembre 1966

[21] M. DEPEYROT — *"UNE GENERALISATION DE LA NOTION D'AUTOMATE ET APPLICATIONS"*
THESE D'ETAT, GRENOBLE, 24 Juin 1975

[22] D.L. DIETMEYER — *"INTRODUCING D.D.L."*
COMPUTER, Décembre 1974

[23] R. EHRMANN — *"LES LANGAGES DE SIMULATION"*
DUNOD, PARIS, 1970

[24] FRANTA, GILOI — *"A.P.L. * DS : A HARDWARE DESCRIPTION LANGUAGE FOR DESIGN AND SIMULATION"*
1975 INTERNATIONAL SYMPOSIUM ON S.H.D.L. AND THEIR APPLICATIONS.
PROCEEDINGS

[25] R. HARTENSTEIN — *"FUNDAMENTALS OF STRUCTURED HARDWARE DESIGN"*
NORTH HOLLAND PUBLISHING COMPANY, Avril 1977

[26] F.J. HILL — *"INTRODUCING A.H.P.L."*
COMPUTER, Décembre 1974

[27] F.J. HILL, BEN HUEY
"A DESIGN LANGUAGE BASED APPROACH TO TEST SEQUENCE GENERATION"
COMPUTER, Vol. 10, n° 6, Juin 1977

[28] JAYAKUMAR, Mc CALLA, Jr
"SIMULATION OF MICROPROCESSOR EMULATION USING GASP-PL/1"
COMPUTER, Vol. 10, n° 4, Avril 1977

[29] JORDAN, SMITH — *"THE ASSIGNEMENT STATEMENT IN HARDWARE DESCRIPTION LANGUAGES"*
COMPUTER, Vol. 10, n° 6, Juin 1977.

[30] H. LILEN — *"DU MICRO-PROCESSEUR AU MICRO-ORDINATEUR"*
Editions RADIO, PARIS, 1975

[31] G.J. LIPOUSKI — *"HARDWARE DESCRIPTION LANGUAGES : VOICES FROM THE TOWER OF BABEL"*
COMPUTER, Vol. 10, n° 6, Juin 1977

[32] J. MERMET — *"ETUDE METHODOLOGIQUE DE LA CONCEPTION ASSISTEE PAR ORDINATEUR DES SYSTEMES LOGIQUES : CASSANDRE"*
THESE D'ETAT, GRENOBLE, 10 Avril 1973

[33] J. MERMET *"HARDWARE DESCRIPTION LANGUAGES IN FRANCE"*
COMPUTER, Décembre 1974

[34] MOALLA, SIFAKIS, ZACHARIADES
"UN LANGAGE D'AIDE A LA CONCEPTION ET A LA SIMULATION DE SYSTEMES COMPLEXES"
R.R. n° 16, GRENOBLE, Octobre 1975

[35] F.J. RAMMING *"DIGITEST II : AN INTEGRATED STRUCTURAL AND BEHAVIORAL LANGUAGE"*
1975 INTERNATIONAL SYMPOSIUM ON C.H.D.L. AND THEIR APPLICATIONS.
PROCEEDINGS

[36] D. SIEWIOREK *"INTRODUCING P.M.S."*
COMPUTER, Décembre 1974

[37] D. SIRWIOREK *"INTRODUCING I.S.P."*
COMPUTER, Décembre 1974

[38] E.P. STABLER *"SYSTEM DESCRIPTION LANGUAGES"*
I.E.E.E. TRANSACTIONS ON COMPUTERS,
Vol. C-19, n° 12, Décembre 1970

[39] J.H. STEWERT *"LOGAL : A C.H.D.L. FOR LOGIC DESIGN AND SYNTHESIS OF COMPUTERS"*
COMPUTER, Vol. 10, n° 6, Juin 1977

[40] S.Y.H. SU *"A SURVEY OF COMPUTER HARDWARE DESCRIPTION LANGUAGES IN THE U.S.A."*
COMPUTER, Décembre 1974

[41] S.Y.H. SU *"HARDWARE DESCRIPTION LANGUAGE APPLICATIONS : AN INTRODUCTION AND PROGNOSIS"*
COMPUTER, Vol. 10, n° 6, Juin 1977

[42] VERNEL P. THESE D'ETAT présenté à l'INSTITUT NATIONAL POLYTECHNIQUE, NANCY, 1977

[43] E.W. VOGEL *"A MODEL APPROACH TO THE DESCRIPTION OF HARDWARE SYSTEMS"*
1975 INTERNATIONAL SYMPOSIUM ON C.H.D.L. AND THEIR APPLICATIONS.
PROCEEDINGS.

[44] M. ZACHARIADES *"MAS : REALISATION D'UN LANGAGE D'AIDE A LA CONCEPTION DE SYSTEMES LOGIQUES"*
THESE DE 3ème CYCLE, GRENOBLE, 14 Septembre 1977

ANNEXE A

DESCRIPTEUR LEDA

1 <description LEDA>::=<description C>$^{(1-C)}$<description optionnelle>

2 <description optionnelle>::=<description P optionnelle>
<description C optionnelle>$^{(4)}$

3 <description P optionnelle>::=∧|<description P>$^{(1-P)}$

4 <description A optionnelle>::=∧|<description A>$^{(1-A)}$

DESCRIPTEUR DES CONNEXIONS

1 <description C>::= connexions;<arbre d'unités>fconnexions;

2 <arbre d'unités>::=1 unité<variable><initialisation>,
liaisons<liste de variables avec longueur>$^{(20)}$;
<sous-arbres>$^{(4)}$

3 <initialisation>::= |(<variable>)

4 <sous-arbres>::=<entier>unité<variable avec paramètres>$^{(6)}$
<suite ou fin>

5 <suite ou fin>::=<quantité>$^{(9)}$def<variable>;<sous-arbres>$^{(4)}$|
<quantité>$^{(9)}$;<sous-arbres>$^{(4)}$|
<quantité>$^{(9)}$;def<variable>

6 <variable avec paramètres>::=<variable>(<liste de paramètres>)

7 <liste de paramètres>::=<paramètre caractérisé>|
<paramètre caractérisé>,<liste de paramètres>

8 <paramètre caractérisé>::=<variable><type>$^{(87\text{-}P)}$

9 <quantité>::=nombre<entier>|^

10 <liste de variables avec longueur>::=<variable avec longueur>|
<variable avec longueur>,<liste de variables avec longueur>

11 <variable avec longueur>::=<variable>|<variable><longueur et type>

12 <longueur et type>::=<longueur>|<longueur><type>$^{(87\text{-}P)}$

13 <longueur>::=[<entier>:<entier>]

DESCRIPTEUR DE PROTOTYPES

1 <description P>::=prototypes;<liste de définitions>Fprototypes;

2 <liste de définitions>::=<définition>|<définition><liste de définitions>

3 <définition>::= tête de définition><corps de définition>(5)

4 <tête de définition>::=Unité<variable avec paramètres>(84);

5 <corps de définition>::=<déclarations><fonction>funité

6 <déclaration>::=déclarations;<liste de déclarations>Fdéclarations; |^

7 <liste de déclarations>::=<partie principale>|
<partie principale><partie optionnelle>(74)

8 <partie principale>::=<liste de déclarations principales>|^

9 <liste de déclarations principales>::=<déclaration principale>|
<déclaration principale><liste de déclarations principales>

10 <déclaration principale>::=<déclaration de registre>|
<déclaration de mémoire>(15)|
<déclaration d'entier>(20)

11 <déclaration de registre>::=registre<liste de registres>;

12 <liste de registres>::=<registre avec longueur>|<registre avec longueur>,
<liste de registres>

13 <registre avec longueur>::=<identificateur de registre><longueur en bits>

14 <longueur en bits>::=bit(<borne inférieure>:<borne supérieure>)

15 <déclaration de mémoire>::=mémoire<liste de mémoire>;

16 <liste de mémoires>::=<mémoire avec longueur>|<mémoire avec longueur>,
<liste de mémoire>

17 <mémoire avec longueur>::=<identificateur de mémoire><longueur>

18 <longueur>::=(<longueur en mots>)<longueur en bits>(14)

19 <longueur en mots>::=<borne inférieure>:<borne supérieure>

20 <déclaration d'entier>::=entier<liste d'identificateurs d'entier>;

21 <liste d'identificateurs d'entier>::=<identificateur d'entier>|
<identificateur d'entier>,<liste d'identificateur d'entier>

22 <fonction>::=<liste d'instructions>

23 <liste de S instructions>::=<S instructions;|<S instruction>;
<liste de S instructions>

24 <S instruction>::=<S instruction de transfert>|<S instruction sélective>(54) |
<S instruction FAIRE>(67) |<S instruction ATTENDRE>(68) |
activer<action>(69) |<S instruction de répétition>(72)

25 <S instruction de transfert>::=<S partie gauche> → <S partie droite>(53)

26 <S partie gauche>::=<S partie gauche 1>(28)<S suite 1>

27 <S suite 1>::= |+<S partie gauche 1>|-<S partie gauche 1>

28 <S partie gauche 1>::=<S partie gauche 2>(30)<S suite 2>

29 <S suite 2>::= <S partie gauche 2>|/<S partie gauche 2>

30 <S partie gauche 2>::=(<S partie gauche>)|<S opérande arithmétique>|
<constante arithmétique>

31 <S opérande arithmétique>::=<S opérande 1>|non<S opérande 1>

32 <S opérande 1>::=<S opérande 2>(34)<S suite 3>

33 <S suite 3>::=∧|ou<S opérande 2>

34 <S opérande 2>::=<S opérande 3>(36)<S suite 4>

35 <S suite 4>::=∧|et<S opérande 3>

36 <S opérande 3>::=<S opérande 4>(38)<S suite 5>

37 <S suite 5>::=∧|&<S opérande 4>

38 <S opérande 4>::=(<S opérande arithmétique>)|<S opérande simple>(39,95)

39 <opérande simple>::=<tranche d'entrée>|<tranche de registre>(41) |
<tranche de mémoire généralisée>(42) |
<identificateur d'entier>|<variable d u système>(88)

40 <tranche d'entrée>::=<entrée>(48) | <entrée>(<troncature>(44))

41 <tranche de registre>::=<registre>(49) | <registre>(<troncature>(44))

42 <tranche de mémoire généralisée>::=<tranche de mémoire> |
<tranche de mémoire>(<troncature>(44))

43 <tranche de mémoire>::=<mémoire>(50)(<rang>(45),<nb de mots>(46))

44 <troncature>::=<rang>,<nb de bits>(47)

45 <rang>::=<expression>(51)

46 <nb de mots>::=<expression>(51)

47 <nb de bits>::=<expression>(51)

48 <entrée>::=<identificateur d'entrée>

49 <registre>::=<identificateur de registre>

50 <mémoire>::=<identificateur de mémoire> |
nom<identificateur de mémoire>

51 <expression>::=<partie gauche>(26)

52 <constante arithmétique>::=<constante décimale> |
X'<constante hexadécimale>' |
B'<constante binaire>'

53 <partie droite>::=<opérande simple>(39) | <opérande simple>(39) &
<partie droite>

54 <instruction sélective>::=sélect<liste d'actions conditionnelles>fsélect

55 <liste d'actions conditionnelles>::=<action conditionnelle># |
<action conditionnelle>#<liste d'actions conditionnelles>

56 <action conditionnelle>::=<liste de conditions>:<liste de S.instructions(23)

57 <liste de conditions>::=<liste de condition 1> |
non<liste de conditions 1>

58 <liste de conditions 1>::=<condition 1>(60)<reste 1>

59 <reste 1>::=^ | ou<condition 1>

60 <condition 1>::=<condition 2>(62)<reste 2>

61 <reste 2>::=∧|et<condition 2>

62 <condition 2>::=(<liste de conditions>(57))|<condition>

63 <condition>::=<opérande arithmétique>(31)|<expression de relation>|
<constante booléenne>(66)

64 <expression de relation>::=<expression>(51)<opérateur de relation>
<expression>(51)

65 <opérateur de relation>::= =|≠|≤|<|≥|>

66 <constante booléenne>::=vrai|faux

67 <instruction FAIRE>::= faire<tranche de registre>

68 <instruction ATTENDRE>::=attendre(<expression>(51))|
attendre ev(<expression d'événement>(73))

69 <action>::=<identificateur d'action>(<liste d'arguments>)

70 <liste d'arguments>::=<arguments>|<argument>,<liste d'arguments>

71 <argument>::=<opérande simple>(39)

72 <instruction de répétition>::=jusque ev<expression d'événement>:
<liste d'instructions>(23) recommencer

73 <expression d'événement>::=<liste de conditions(57)>

74 <partie optionnelle>::=<format>(89)|<format>(89)<queue de partie optionnelle>

75 <queue de partie optionnelle>::=<adressage>|<adressage><descriptions>(77)

76 <adressage>::=adressage résultat<registre avec longueur>(13);
<liste d'instructions>(23)

77 <description>::=description<liste de descriptifs>

78 <liste de descriptifs>::=<descriptif>|<descriptif><liste de descriptifs>

79 <descriptif>::=instr<nom mnémonique>:<suite de descriptions>

80 <suite de descriptions>::=(<code opération>(81),<nom de format>);
<liste de schémas d'instructions>(82)

81 <code opération>::=<constante arithmétique>(52)

82 <liste de schémas de instruction>::=<schéma de instruction>;|
<schéma de instruction>;<liste de schémas de instruction>

83 <objet d'adressage>::<identificateur de registre>.<constante décimale>|
ⓐ .<constante décimale>|
ⓐ<constante décimale>.<constante décimale>

84 <variable avec paramètres>::=<variable d'unités>(<liste de
paramètres formels>)

85 <liste de paramètres formels>::=<paramètre formel>|<paramètre formel>,
<liste de paramètres formels>

86 <paramètre formel>::=<nom de paramètre formel><type>

87 <type>::=e|s|es|ev|co

88 <variable du système>::=$FORMAT|$NADR|$CODE|$TAILLE

89 <format>::=format<liste de formats>

90 <liste de formats>::=<descripteur de format>;|
<descripteur de format>;<liste de formats>

91 <descripteur de format>::=1<nom de format><dimension>$^{(93)}$,
<champs>

92 <champs>::=<champ>|<champ>,<champs>

93 <champ>::=<constante décimale><nom de champ><dimension>

94 <dimension>::=(<constante décimale>:<constante décimale>)

95 <schéma de opérande simple>::=<opérande simple $^{(39)}$|
<objet d'adressage>$^{(83)}$|<tranche fictive>

96 <tranche fictive>::=□|□(<troncature>$^{(44)}$)

97 S::=^|schéma de

où S est une méta-variable

DESCRIPTEUR DES ACTIONS

1 <description A>::=dactions;<liste de descripteurs d'actions>fdactions;

2 <liste de descripteurs d'actions>::=<descripteur d'action>|
<descripteur d'action><liste de descripteurs d'actions>

3 <descripteur d'action>::=<tête d'action><corps d'action>(8)

4 <tête d'action>::=action<nom d'action avec paramètres>;

5 <nom d'action avec paramètres>::=<variable d'action>|
<variable d'action>(<liste de paramètres formels d'actions>)

6 <liste de paramètres formels d'action>::=<paramètre formel d'action>|
<paramètre formel d'action>,<liste de paramètres formels d'action>

7 <paramètre formel d'action>::=<variable><type>(87-P)

8 <corps d'action>::=<spécification><déclaration>(19)<fonction>(22-P)
factions;

9 <spécification>::=^|spécifications<liste de spécification de type>
fspécifications

10 <liste de spécifications de type>::=<spécificateur de type>;|
<spécificateur de type>;<liste de spécifications de type>

11 <spécificateur de type>::=<TYPE D'OBJET><liste de variables de TYPE D'OBJET avec longueur>(13)|entier
<liste d'identificateurs d'entier>(21-P)

12 TYPE D'OBJET::=broche|registre|mémoire

13 <liste de variables de TYPE D'OBJET avec longueur>::=
<variable de TYPE D'OBJET>|<variable de TYPE D'OBJET>,
<liste de variables de TYPE D'OBJET avec longueur>

14 <variable de broche>::=<variable>(<taille en bits>$^{(17)}$)

15 <variable de registre>::=<variable>bit(<taille en bits>$^{(17)}$)

16 <variable de mémoire>::=<variable>(<taille en mots>$^{(18)}$)bit
(<taille en bits>)

17 <taille en bits>::=<entier décimal>

18 <taille en mots>::=<entier décimal>

19 <déclaration>::=<déclaration d'entier>$^{(20\text{-}P)}$

où TYPE D'OBJET est une méta-variable

DÉFINITION DU LANGAGE LEDA

Pour des raisons de commodité de présentation, nous allons considérer ici que le langage LEDA est composé de 3 sous-langages, le sous-langage des connexions, le sous-langage des prototypes et le sous-langage des actions, correspondant aux 3 descriptions du système. Ceci nous permettra de diviser la présentation du langage en 3 parties.

A.1.- LES MOTS-CLES DU LANGAGE

Pour bien comprendre la présentation de LEDA, il est utile d'avoir sous les yeux la liste de ses mots-clés. Cette liste comporte 46 mots-clés classé par ordre alphabétique dans la table qui suit :

.../...

N°	MOT-CLE
1	ACTIVER
2	ACTION
3	ADRESSAGE
4	ATTENDRE
5	BIT
6	BROCHE
7	CO
8	COMPL
9	CONNEXIONS
10	DACTIONS
11	DECLARATIONS
12	DEF
13	DESCRIPTION
14	E
15	ENTIER
16	ES
17	ET
18	EV
19	FACTION
20	FAIRE
21	FAUX
22	FCONNEXIONS
23	FDACTIONS

N°	MOT-CLE
24	FDECLARATIONS
25	FORMAT
26	FPROTOTYPES
27	FSELECT
28	FSPECIFICATIONS
29	FUNITE
30	INSTR
31	JUSQUE
32	LIAISONS
33	MEMOIRE
34	NE
35	NOM
36	NON
37	OU
38	PROTOTYPES
39	RECOMMENCER
40	REGISTRE
41	RESULTAT
42	S
43	SELECT
44	SPECIFICATIONS
45	UNITE
46	VRAI

A.2.- LE SOUS-LANGAGE DES CONNEXIONS

Le sous-langage des connexions est destiné à la description de la partie statique, c'est-à-dire des connexions et de la structure logique, d'une configuration. Aussi, se compose-t-il essentiellement de déclarations. On peut distinguer 3 types de déclarations d'unités :

- la déclaration de racine,
- la déclaration de noeud,
- la déclaration de feuille.

1) La déclaration de racine

Cette déclaration a pour objet :

a) De donner un nom à l'unité composée correspondant à la configuration à décrire,

b) D'indiquer au système le nom de l'unité d'initialisation, c'est-à-dire de celle, parmi les unités de la configuration, dont l'activation est nécessaire à la mise en route de l'ensemble,

c) De préciser et de nommer les liaisons internes, c'est-à-dire les fils qui relient physiquement les unités de la configuration et par lesquels transitent les informations.

Le format d'une déclaration de racine est le suivant :

```
1  Unité nom d'unité composée (nom d'unité d'initialisation),
       liaisons liste de variables de liaison;
```

où le chiffre 1 indique le niveau logique de l'unité composée à décrire et qui est toujours égal à 1.

Le nom de l'unité composée ainsi que celui de l'unité d'initialisation obéissent d'une manière générale aux même règles que les identificateurs de variable dans les langages de programmation. Ainsi, si L est l'ensemble des lettres, C est l'ensemble des chiffres décimaux, un nom d'unité est un élément de l'ensemble :

$$L(L \cup C)^*$$

Dans la pratique, le nombre de caractères composant un nom d'unité est limité à 8 par le système en vue d'une implémentation sur ordinateur.

La liste des variables de liaison est composée d'une ou plusieurs déclarations de variable de liaison, séparées par des virgules.

Une déclaration de liaison peut avoir un des 4 formats suivants :

a) nom de variable de liaison
b) nom de variable de liaison
   [borne inférieure : borne supérieure]
c) nom de variable de liaison spécification
   de type
d) nom de variable de liaison
   [borne inférieure : borne supérieure]
   spécificateur de type

Un nom de variable de liaison suit les mêmes règles d'écriture qu'un nom d'unité.

Les bornes inférieure et supérieure servent à indiquer le nombre de fils composant une liaison et sont des nombres décimaux entiers non négatifs.

Le spécificateur de type sert à repérer celles des liaisons de la configuration qui assurent la communication avec des unités externes, et à préciser le type de ces liaisons qui peur être e (entrée), s (sortie), es (entrée ou sortie), co (commande) ou ev (événement).

2) La déclaration de noeud.

Cette déclaration a pour objet d'indiquer les unités logiques des niveaux intermédiaires voulus par l'utilisateur et qui ne correspondent pas à des unités ayant une existence physique.

Voici le format d'une déclaration de noeud :

```
N unité nom d'unité (liste de broches) nombre K ;
                                       optionnel
```

où N est un entier décimal de 2 à 255 indiquant le niveau de l'unité logique.

Une liste de broches est une suite de spécificateurs de broches effectives séparés par des virgules.

Un spécificateur de broche effective peut avoir l'un des deux formats suivants :

a) nom de broche effective spécificateur de type
b) nom de broche effective
   [Borne inférieure : Borne supérieure]
   spécificateur de type.

Le nom de broche effective doit être l'un des nom de variables de liaison déclarées au niveau 1 .

Borne inférieure et borne supérieure sont des entiers décimaux. Ainsi, une broche effective peut être une tranche de variable de liaison.

Le spécificateur de type doit prendre les mêmes valeurs que dans la déclaration de racine.

Une déclaration de noeud est obligatoirement suivie d'une autre déclaration de noeud ou d'une déclaration de feuille.

L'entier décimal K qui suit le mot-clé nombre est un facteur de répétition pour le cas où il existe, au niveau N , plusieurs unités identiques.

3) La déclaration de feuille

La déclaration de feuille a pour objet de définir une unité simple et d'indiquer le nom du prototype qui lui est associé. Le prototype est une unité dont le fonctionnement interne est décrit soit dans le descripteur de prototypes, soit dans la bibliothèque des prototypes.

La forme normale d'une déclaration de prototype est la suivante :

```
N unité nom d'unité simple (liste de broches)
        nombre L déf nom de prototypes
        ~~~~~~~~~~
        optionnel
```

Le nom du prototype associé à l'unité simple suit le mot-clé def. Ceci permet de différencier des unités définies par le même prototype.

A.3.- LE SOUS-LANGAGE DES PROTOTYPES.

Le sous-langage des prototypes est destiné à la mise en oeuvre du descripteur des prototypes et donc à la description du fonctionnement interne de chaque prototype. Il est composé de déclarations et d'instructions qui apparaissent, dans un descripteur, entre l'en-tête et le mot-clé funité.

LES DECLARATIONS.

Les déclarations constituent la première partie d'une description de prototype. Elles ont pour objet la définition des composants statiques d'un prototype ainsi que de variables définies par l'utilisateur pour les besoins de la simulation. Il existe 6

types de déclarations, la déclaration de registre, la déclaration de mémoire, la déclaration d'entier, la déclaration de format, la déclaration d'adressage et la déclaration de description. Dans un descripteur de prototype, les déclarations sont précédées du mot-clé déclaration et suivies du mot-clé fdéclarations.

1) La déclaration de registre.

La déclaration de registre sert à définir des registres, c'est-à-dire des ensembles de bascules, dont dispose le prototype à décrire. La forme d'une déclaration de registre est la suivante :

registre liste de registres avec longueur

Une liste de registres avec longueur est composée d'une suite de variables de registre avec longueur, séparées par des virgules. Une variable de registre avec longueur a la forme suivante :

nom de registre bit (bi : bs)

où le nom de registre est limité à 8 caractères et b_i et b_s sont des constantes entières décimales, binaires ou hexadécimales indiquant la dimension du registre et la numérotation des bascules qui le composent.

2) La déclaration de mémoire

Cette déclaration sert à définir des mémoires, c'est-à-dire des ensembles totalement ordonnés et indicés de registres dont la fonction principale est la mémorisation, permanente ou temporaire, d'informations. Une déclaration de mémoire s'écrit de la manière suivante :

mémoire liste de mémoires avec longueur;

Une liste de mémoires avec longueur est composée d'une suite de variables de mémoire avec longueur, séparées par des virgules. Une variable de mémoire avec longueur a la forme suivante :

nom de mémoire (mi : ms) bit (bi : bs)

où m_i et m_s sont des constantes entières décimales, binaires ou hexadécimales indiquant la dimension de la mémoire et la numérotation des registres qui la composent.

3) La déclaration d'entier

La déclaration d'entier sert à définir des variables de type entier que l'utilisateur peut introduire pour les besoins de la simulation.

La forme d'une déclaration d'entier en LEDA est la suivante :

entier liste de variables entières;

Une liste de variables entières est composée d'une suite de variables entières, séparées par des virgules.

4) La déclaration de format

Cette déclaration n'a de sens que pour les unités pouvant exécuter des instructions. Elle sert à définir les formats des instructions de l'unité à décrire. Une déclaration de format s'écrit comme suit :

format liste de formats;

Une liste de formats est composée d'un ou plusieurs descripteurs de format séparés par des point-virgules.

Le descripteur de format divise une instruction en un ou plusieurs champs chacun desquels peut être divise en sous-champs. On accède aux différents champs et sous-champs par leur nom. Un descripteur de format s'écrit, en LEDA, de la façon suivante :

```
1 nom de format (bi_1 : bs_1),
  2 nom de champ (bi_2,x : bs_2,x),
  :
```

où les points de suspension indiquent d'autres champs de niveau supérieur ou égal à 2 .

Les bornes bi et bs indiquent les dimensions des champs et le premier indice correspond au niveau du champ courant. Le second indice contient un ensemble d'informations permettant de distinguer les bornes entre elles. Pour un champ de niveau M :

```
M nom de champ (bi_M,x : bs_M,x)
```

composé des ℓ sous-champs :

```
N nom de champ 1 (bi_N,x1 : bs_N,x1)
N nom de champ 2 (bi_N,x2 : bs_N,x2)
:
N nom de champ   (bi_N,xℓ : bs_N,xℓ)
```

l'on doit avoir d'une part :

$$N = M + 1$$

et d'autre part :

$$bi_{N,x_1} = bi_{M,x} \quad , \quad bs_{N,x_\ell} = bs_{M,x} \quad \text{et} \quad bi_{N,x_j} = bs_{N,j_x-1} + 1$$

La correspondance entre le format et le registre d'instruction se fait par l'intermédiaire de l'instruction LEDA FAIRE qui permet d'exécuter l'instruction dont le code est contenu dans le registre d'instruction. L'exécution se fait suivant la déclaration de description associée à cette instruction.

Une déclaration de format est toujours suivie d'une déclaration d'adressage.

5) La déclaration d'adressage

La déclaration d'adressage sert à décrire le processus d'adressage d'une unité capable d'exécuter des instructions. Elle a la forme suivante :

```
Adressage résultat variable de registre avec longueur;
          liste d'instructions;
```

La variable de registre qui suit le mot-clé résultat sert à désigner le registre qui contiendra le résultat du calcul d'adresse. Une variable de registre avec longueur s'écrit

```
nom de registre (bs : bi)
```

où le nom de registre doit être un des noms déjà déclarés dans les déclarations de registres.

Les instructions qui composent la liste d'instructions sont séparées entre elles par des point-virgules et servent à décrire le processus du calcul d'adresse.

La fonction d'adressage n'est pas paramétrisable car le mode d'adressage est indiqué par le format.

6) La déclaration de description

La déclaration de description sert à décrire le fonctionnement des instructions d'une unité programmable.
Cette déclaration a la forme suivante :

```
Description liste de descripteurs d'instructions
```

Un descripteur d'instruction s'écrit comme suit :

```
instr nom mnémonique : (code opération, nom de format);
                       liste de schémas d'instructions;
```

Le nom mnémonique sert à rendre plus claire la description.

Le code opération est une constante décimale, binaire ou hexadécimale et le nom de format est le nom d'un des formats déclarés dans la déclaration de format et sert à indiquer le format de l'instruction décrite.

La liste des schémas d'instructions est composée d'un ou plusieurs schémas d'instructions séparés par des point-virgules.

Un schéma d'instruction ressemble, à quelques détails près, à une instruction ordinaire. La différence principale réside

dans le fait que le schéma d'instruction décrit statiquement un processus qui n'est activé qu'explicitement par l'exécution d'une instruction FAIRE que nous verrons plus loin. Le déroulement d'un processus pouvant durer plusieurs unités de temps, il en va de même de l'exécution d'un schéma d'instruction alors que l'exécution d'une instruction ordinaire en LEDA dure généralement une fraction d'unité de temps. D'autre part, un schéma d'instruction peut comporter des symboles spéciaux dont nous verrons la signification dans ce qui suit.

Le symbole " @ "

C'est le symbole d'adressage. Son apparition dans un schéma d'instruction implique l'activation du processus d'adressage décrit dans la déclaration d'adressage. Il peut apparaître dans un schéma d'instruction sous la forme suivante :

forme : @[indice]

résultat : Activation du processus d'adressage et mise de l'adresse calculée dans le registre résultat de la déclaration d'adressage. L'indice, qui est optionnel, sert, pour une instruction comportant plus d'une adresse, à indiquer à la fonction d'adressage laquelle de ces adresses il faut calculer. Pour cela, il positionne la variable du système $NADR à la valeur de l'indice. La valeur de la variable $NADR peut être testée par la fonction d'adressage.

Le symbole "."

C'est le symbole de lecture ou écriture mémoire, selon qu'il apparaît en partie gauche ou en partie droite d'une instruction de transfert. On le rencontre sous la forme suivante :

forme : variable ou symbole d'adressage . indice.

résultat : Activation d'une action de lecture ou d'écriture mémoire à l'adresse contenue dans la variable qui précède le symbole ou calculée par la fonction d'adressage. Le processus de lecture et écriture mémoire sera décrit par l'utilisateur dans une ou plusieurs actions dont le nom, imposé, sera respectivement $LEC et $ECR suivi d'un indice. Ainsi, l'apparition en partie droite d'un schéma d'instruction de transfert de .18 provoque l'appel implicite de l'action $ECR018.

Le symbole "□"

C'est le symbole de variable fictive. Il est utilisé dans la description des instructions de comparaison. Sa première apparition doit se faire en partie droite d'un schéma d'instruction de transfert pour indiquer la création par le système d'une variable fictive, de type registre et de taille égale à celle de la première variable de la partie gauche, destinée à recevoir le résultat du calcul effectué en partie gauche.

LES INSTRUCTIONS

Les déclarations que l'on vient de voir servent à décrire la partie statique et une unité. Pour la description de sa partie opératoire, c'est-à-dire de son fonctionnement interne, on a besoin d'un outil dynamique, c'est-à-dire d'instructions. Aussi, le sous-langage des prototypes comporte-t-il un certain nombre de types d'instructions que l'on va décrire dans ce qui suit.

Avant de voir en détail les instructions du sous-langage des prototypes, parlons un peu des objets qu'elles manipulent. Ces objets appartiennent obligatoirement à l'une des catégories suivantes :

1) Constantes entières

Ces constantes peuvent être décimales, binaires ou hexadécimales. Leurs formes respectives sont :

- chaînes de chiffres décimaux,
- B 'chaîne de chiffres binaires'
- X 'chaîne de chiffres hexadécimaux'

2) Constantes booléennes :

Il existe deux constantes booléennes, vrai et faux.

3) Variables de broche

Ce sont les variables formelles figurant dans l'en-tête d'un descripteur de prototype et désignant les broches de l'unité. Leurs valeurs peuvent être testées ou apparaître en partie gauche d'une instruction de transfert. Elles ne peuvent être modifiées que pour les variables désignant des broches de sortie ou de commande. Les variables de broche ont des propriétés de mémorisation.

4) Variables de registre :

Ce sont les variables déclarées dans la déclaration de registre. Leurs valeurs peuvent être testées ou modifiées.

5) Variables de mémoire :

Ce sont les variables déclarées dans la déclaration de mémoire. Comme pour les variables de registre, leurs valeurs peuvent être testées ou modifiées.

6) Variables d'entier :

Elles doivent être déclarées dans la déclaration d'entier du descripteur de prototype. Leurs valeurs peuvent être testées ou modifiées.

7) Variables d'action :

Ce sont des variables désignant des actions. Elles n'ont pas à être déclarées. Elles sont utilisées exclusivement dans une instruction d'activation afin de donner le

contrôle aux actions correspondantes.

8) Variables du système

Elles sont au nombre de trois : $FORMAT , $TAILLE et $NADR de types chaîne de caractères et entiers respectivement. Le langage LEDA autorise l'utilisation de constantes de type chaîne de caractères uniquement pour tester la valeur de $FORMAT. Une constante de type chaîne de caractères est composée d'une chaîne de caractères entre apostrophes. L'utilisation des variables du système n'a de sens que dans une unité capable d'exécuter des instructions. Ainsi, pour chaque instruction exécutable de l'unité, le système affecte une valeur à $FORMAT et $TAILLE suivant la déclaration de format et la déclaration de description de l'instruction.

A partir des catégories d'objets que l'on vient de voir, l'utisateur peut construire des objets plus complexes, les tranches; dont il existe 4 sortes :

- la tranche de broche dont la forme est :

  variable de broche(rang 1ère broche, nb de broches)

- la tranche de registres dont la forme est :

  variable de registre(rang 1er bit, nb de bits)

- la tranche de mémoire qui s'écrit sous la forme :

  variable de mémoire (rang 1er mot, nb de mots)

- la tranche de tranche de mémoire qui s'écrit :

  tranche de mémoire (rang 1er bit, nb de bits)

où les deux éléments entre parenthèses peuvent être des expressions comportant tous les objets que l'on vient de définir, y compris les tranches.

Voici maintenant les instructions du langage, qui se terminent toujours par un ";" .

1) L'instruction de transfert

C'est l'instruction qui réalise le transfert de signaux. Sa forme syntaxique est la suivante :

expression → variable simple ou composée

Une expression est une suite d'opérandes reliés entre eux par des opérateurs et, éventuellement, des parenthèses.

Un opérande peut être un quelconque des objets que l'on a défini précédemment.

Les opérateurs sont de 3 types : arithmétiques, logiques et de composition.

- Opérateurs arithmétiques

+ Addition - Soustraction	}	Priorité 1
* Multiplication / Division	}	Priorité 2

où les opérateurs de multiplication et de division sont prioritaires par rapport à ceux d'addition et de soustraction.

Les opérations désignées par ces opérateurs sont des opérations arithmétiques portant sur des nombres binaires en virgule fixe avec un bit de signe. L'opérateur de division désigne une division entière.

- Opérateurs logiques

ou	Union logique,	priorité 3
et	Intersection,	priorité 4
non	complément,	priorité 5

Les opérateurs logiques sont prioritaires par rapport aux opérateurs arithmétiques. L'opérateur non est un opérateur unaire.

- Opérateur de composition

  &

  Cet opérateur réalise la composition des opérandes correspondants en mettant fictivement le second à la suite du premier. Cette composition n'a d'existence que durant le déroulement de l'instruction de transfert. L'opérateur de composition est transitif et non commutatif.

Une variable simple est un objet quelconque parmi ceux définis précédemment à l'exception, bien entendu, des constantes.

Une variable composée est formée d'un ensemble de variables simples, reliées par des opérateurs de compositions. Ainsi, l'opérateur de composition peut figurer en partie droite d'une instruction de transfert.

La sémantique de l'instruction de transfert est la suivante :

> Le signal de la partie gauche est évalué et transféré dans la variable de la partie droite. Ce processus dure, au plus, une unité de temps.

2) L'instruction sélective

C'est l'instruction d'activation conditionnelle. Sa forme syntaxique est la suivante :

```
Sélect
     Condition 1  :  Action 1 #
     Condition 2  :  Action 2 #
          :
          :
     Condition n  :  Action n #
fsélect
```

où Action i est une suite d'instructions terminées par des point-virgules et condition i est une suite d'expressions de relation reliées par des opérateurs logiques et, éventuellement, des parenthèses. Une condition peut aussi se réduire à une constante booléenne.

Une expression de relation s'écrit sous la forme suivante :

expression opérateur de relation expression

où expression se définit de la même façon que pour la partie gauche d'une instruction de transfert.

L'instruction sélective s'exécute comme suit : Les conditions sont parcourues séquentiellement et chacune d'elles est évaluée. Une seule action est exécutée, celle qui correspond à la première condition vérifiée.

L'exécution de l'action qui suit une condition se fait en 1 ou plusieurs cycles, suivant l'absence ou la présence d'une ou plusieurs instructions ATTENDRE. Dans ce dernier cas, l'unité se mettra en attente, le nombre de cycles nécessaire et l'exécution reprendra à l'endroit où elle avait été interrompue.

3) L'instruction d'activation

C'est l'instruction d'activation d'une action parmi celles figurant dans le descripteur des actions ou dans la bibliothèque des actions. Sa forme syntaxique est la suivante :

activer variable d'action (liste de paramètres)

où la liste de paramètres est une suite de paramètres effectifs séparés par des virgules dont les noms remplacent les paramètres formels de l'action.

L'exécution d'une action activée dure un ou plusieurs cycles. Dans ce dernier cas, l'exécution reprend à l'endroit où elle avait été interrompue.

4) L'instruction ATTENDRE

C'est l'instruction de mise en attente d'une unité. Il en existe 2 variantes. Dans la première variante, dont la forme est

attendre (expression)

l'expression est évaluée et sa valeur entière indique le nombre d'unités de temps durant lesquelles l'unité restera en attente. A la fin de l'attente, l'exécution reprend où elle avait été interrompue.

La forme de la seconde variante est

attendre ey (expression d'événement)

où une expression d'événement est une condition.

Ici, l'expression d'événement est évaluée à chaque tentative d'activation de l'unité et l'exécution ne reprend, à l'endroit où elle avait été interrompue, que lorsque la valeur de l'expression est vrai. Sinon, l'unité reste en attente.

5) L'instruction FAIRE

C'est l'instruction qui déclenche l'exécution d'une instruction d'une unité capable d'en exécuter. Elle n'a de sens que dans le descripteur d'une unité de ce type et comportant de surcroît une déclaration de description. Sa forme syntaxique est la suivante :

faire objet de registre

où un objet de registre est soit une variable de registre, soit une tranche de registre.

L'instruction FAIRE déclenche l'exécution de l'instruction contenue dans l'objet de registre. Cette exécution se fait suivant la déclaration de description associée à cette instruction.

6) L'instruction de répétition.

C'est une instruction désignant un ensemble d'actions à répéter jusqu'à l'arrivée d'un événement. Sa forme syntaxique est la suivante :

```
jusque ey Expression d'événement :
        liste d'instructions
recommencer
```

L'exécution de la liste d'instructions est recommencée tant que l'expression d'événement a la valeur faux.
Lorsque l'expression d'événement prend la valeur vrai, le contrôle est donné à l'instruction qui suit l'instruction de répétition.

L'EN-TETE D'UN DESCRIPTEUR

L'en-tête d'un descripteur de prototype a la forme syntaxique suivante :

```
unité nom d'unité (liste de paramètres formels de broche)
```

Un paramètre formel de broche est un nom de broche formelle, suivi éventuellement d'un entier décimal entre parenthèses indiquant la taille (nombre de fils) lorsque cette taille n'est pas égale à 1 ,

suivi d'un indicateur de type de la broche (e, s, es, co, ey). Les paramètres formels de broche sont séparés par des virgules. Lors de l'activation d'un prototype, les noms de broches formelles sont remplacés par des objets de broche effectifs figurant dans la ligne d'appel du prototype, à l'intérieur du descripteur des connexions.

A.4.- LE SOUS-LANGAGE DES ACTIONS.

Le sous-langage des actions sert à la mise en route du descripteur des actions. Un descripteur d'action a la forme suivante :

```
Action nom d'action (liste de paramètres formels avec type);
        spécifications
                liste de spécifications de nature
        fspécifications
        déclaration d'entiers;
        liste d'instructions
faction;
```

Un paramètre formel avec type est un paramètre formel suivi d'un indicateur de type. Un indicateur de type peut prendre pour valeur e , s ou es . Les paramètres de type e ne doivent pas figurer en partie droite d'une instruction de transfert.

Les spécifications, qui sont sans objet pour une action sans paramètres, servent à spécifier la nature et la taille des paramètres formels. Un paramètre formel pouvant être soit de type broche, soit de type registre, soit de type mémoire, soit de type entier, les spécifications respectives ont la forme :

.../...

broche liste de variables de broche avec longueur;
registre liste de variables de registre avec longueur;
mémoire liste de variables de mémoire avec longueur;
entier liste de variables d'entier;

où une variable de broche avec longueur s'écrit :

variable de broche (t_B)

une variable de registre avec longueur s'écrit :

variable de registre bit (t_B)

et une variable de mémoire avec longueur s'écrit

variable de mémoire (t_M) bit (t_B)

t_B et t_M étant des constantes décimales indiquant respectivement la taille en mots et en bits.

La déclaration d'entier, qui est facultative, a la même forme que dans un descripteur de prototype et sert à introduire des objets de type entier nécessaires à la description et qui n'ont d'existence que dans le corps d'une action.

Les variables utilisées dans une action et n'y étant ni spécifiées ni déclarées, sont supposées être des variables externes déclarées dans le descripteur de prototypes faisant appel à l'action.

Les instructions d'une action sont les mêmes que celles d'un descripteur de prototype et permettent d'accéder aux objets suivants :

- variables de broche,
- variables de registre,
- variables de mémoire,
- tranches de ces variables,
- variables composées à partir des objets cités précédemment,
- variables d'entier figurant dans les spécifications dans la déclarations d'entiers,
- constantes.

A N N E X E B

-=-=-=-=-=-=-=-

L'ORGANISATION DES BIBLIOTHEQUES DU SYSTEME

A N N E X E B

L'ORGANISATION DES BIBLIOTHEQUES DU SYSTEME

B.1.- LA BIBLIOTHÈQUE D'UNITÉS.

La bibliothèque d'unités est composée en fait de 3 bibliothèques :

1) LA BIBLIOTHEQUE DE PROTOTYPES

Cette bibliothèque contient tous les renseignements relatifs aux unités prédéfinies qui y sont cataloguées soit implicitement lorsque leur comportement est décrit dans le descripteur des prototypes d'une description écrite en LEDA, soit explicitement par l'utilisation de la primitive AJOUTER.

On accède aux différentes parties de la bibliothèque des prorotypes par l'intermédiaire d'une table des matières qui comporte le nom de chaque prorotype et divers renseignements le concernant. Un élément de la table des matières se présente sous la forme suivante :

.../...

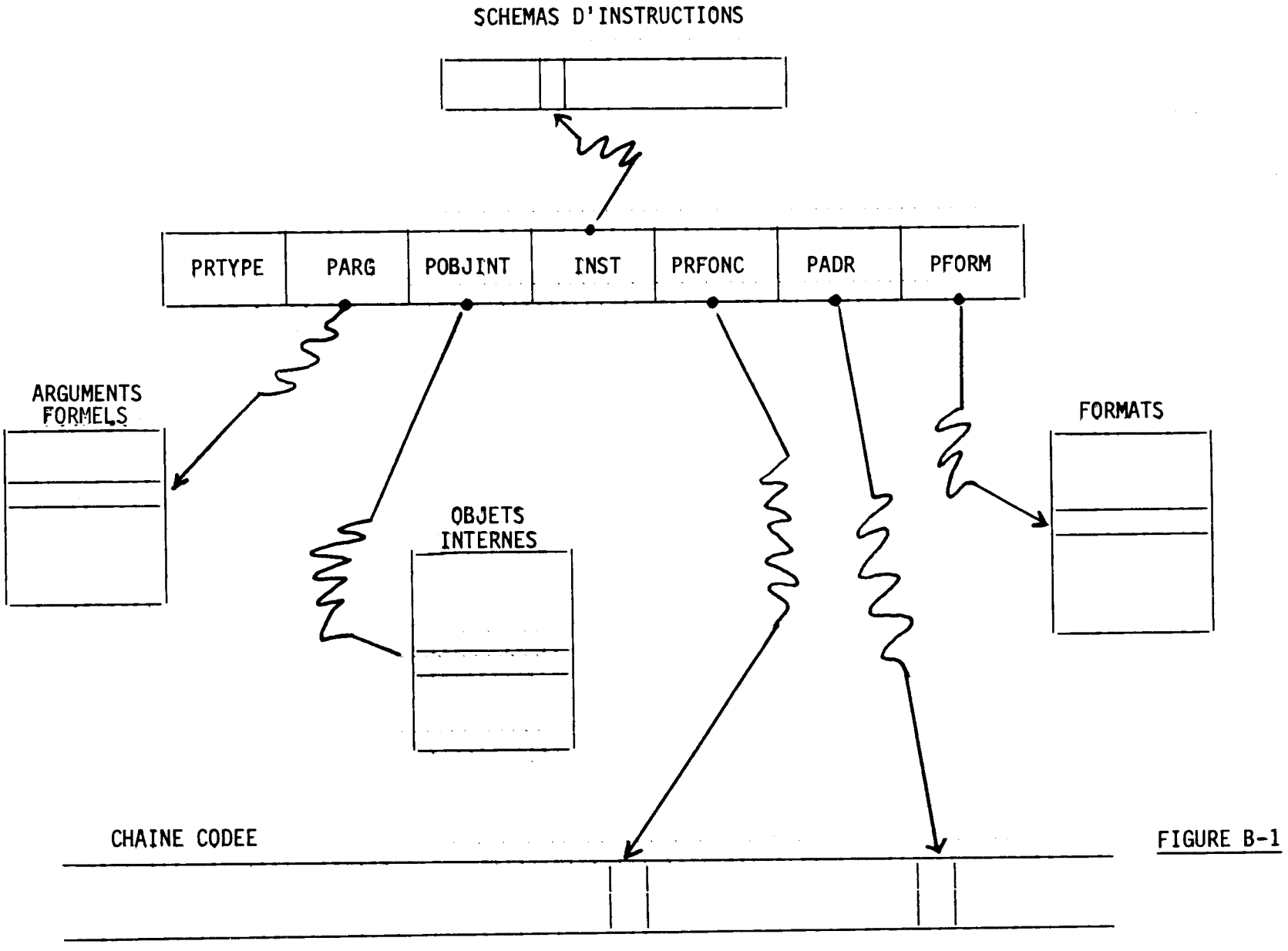

SCHEMAS D'INSTRUCTIONS
PRTYPE
PARG
POBJINT
INST
PRFONC
PADR
PFORM
ARGUMENTS FORMELS
OBJETS INTERNES
FORMATS
CHAINE CODEE

FIGURE B-1

où PRTYPE contient le nom du prototype,
PARG contient l'adresse d'origine des arguments (broches) du prorotype dans une table d'arguments formels,
POBJINT contient l'adresse d'origine des objets internes du prorotype dans une table d'objets internes,
INST sert de point d'entrée à la bibliothèque des schémas d'instructions si le prototype exécute des instructions machine,
PRFONC contient l'adresse d'origine de la chaîne codée décrivant le fonctionnement du prototype,
PADR contient l'adresse d'origine de la chaîne codée décrivant la fonction d'adressage d'un prototype capable d'exécuter des instructions,
PFORM contient l'adresse d'origine de représentation des formats des instructions du langage-machine exécuté par le prototype, dans une table de formats.

La table d'arguments formels contient des renseignements relatifs aux broches de tous les prototypes catalogués dans la bibliothèque. Un élément de la table se décompose comme suit :

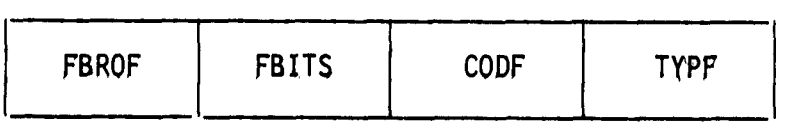


FIGURE B-2

où FBROF contient le nom d'une broche ou d'un ensemble de broches regroupées par leur fonction,
FBITS contient le nombre de broches composant l'élément courant (FBITS = 1 lorsque l'élément est composé d'une seule broche),
CODF contient un code associé à l'élément courant,
TYPF contient un indicateur de type qui a pour valeur e, s, es, co ou ev suivant que l'élément est une entrée, une sortie, une entrée et sortie, une commande ou un événement.

La table d'objets internes contient des renseignements relatifs à tous les objets (registres, mémoires, entiers,...) définis dans chaque prototype de la bibliothèque. La forme d'un élément de la table est la suivante :

OBJETS	BINFER	BSUPER	BITI	BITS	TYPO

FIGURE B-3

où OBJETS contient le nom de l'objet,

BINFER ET BSUPER contiennent respectivement la borne inférieure et la borne supérieure d'un vecteur de mots représentant un objet de type mémoire,

BITI et BITS contiennent respectivement la borne supérieure et la borne inférieure d'un vecteur de bascules représentant un objet de type registre ou un mot d'un objet de type mémoire. Ainsi, l'on aura :

OBJETS = 'MV' , BINFER = 0 , BSUPER = 255,
BITI = 0 , BITS = 7 pour un objet déclaré :

mémoire MV(0 : 255) bit(0 : 7)

TYPO contient des informations sur le type de l'objet (registre, mémoire, entier, objet d'adressage, objet d'action)..

La chaîne codée décrivant le fonctionnement d'un prototype est composée d'un ensemble de triplets chaînés entre eux. La forme d'un triplet de l'ensemble est la suivante :

.../...

NOINST	CHCODUN	SUI

FIGURE B-4

La chaîne codée est construite par l'analyseur syntaxique du langage LEDA et contient des images d'instructions destinées à être interprétées. La signification de chacun des composants d'un triplet de la chaîne codée est la suivante :

NOINST contient le numéro de l'instruction à laquelle appartient l'élément courant de la chaîne.

CHCODUN contient le code de l'élément courant de la chaîne (opérande ou opérateur)

SUI contient l'adresse du triplet suivant.

La Table des formats contient tous les renseignements relatifs aux différents formats d'instructions de tous les prototypes munis d'un langage-machine. Cette table est constituée d'éléments de la forme :

FORMAT	NIVFMT	SFMT	CHFMT	BIFMT	BSFMT	CODFMT

FIGURE B-5

où

FORMAT contient le nom d'un format ou d'un champ de format,

NIVFMT contient le niveau d'un format (NIVFMT = 1) ou d'un champ de format,

SFMT contient l'indice de l'élément de la table représentant le premier champ d'un format ou d'un champ,

CHFMT contient l'indice de l'élément de la table représentant le format suivant d'un prototype ou le champ suivant, de même niveau, d'un format donné,

BIFMT contient la borne inférieure d'un format ou d'un champ,

BSFMT contient la borne supérieure d'un format ou d'un champ,

CODFMT contient le code attribué à un format ou à un champ.

2) LA BIBLIOTHEQUE D'ACTIONS

Cette bibliothèque contient tous les renseignements relatifs aux actions qui y sont cataloguées. Le rangement d'une action dans la bibliothèque d'actions se fait implicitement lorsqu'une action apparaît dans la description des actions d'une description écrite en LEDA.

On accède aux éléments de cette bibliothèque par l'intermédiaire d'une table des matières dont les éléments ont la structure suivante :

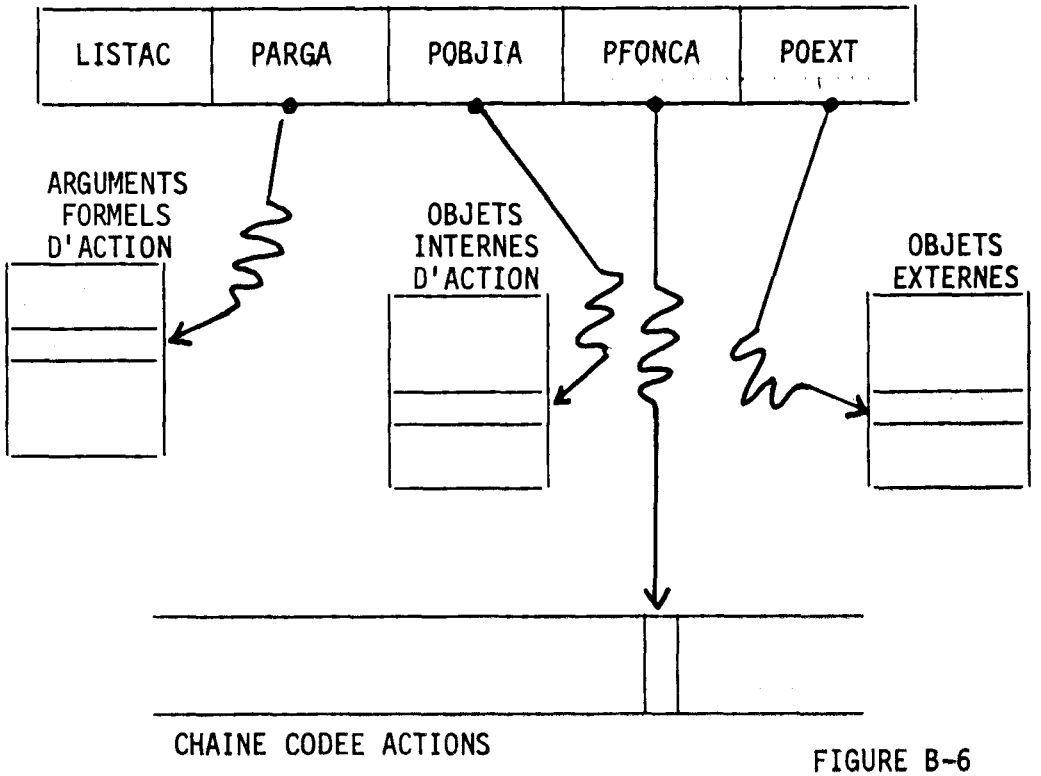


FIGURE B-6

où

LISTAC contient le nom d'une action,

PARGA contient l'adresse d'origine des arguments de l'action dans une table d'arguments formels d'actions,

POBJIA contient l'adresse d'origine des objets internes de l'action dans une table d'objets internes d'actions,

PFONCA contient l'adresse d'origine de la chaîne codée décrivant l'action,

POEXT contient l'adresse d'origine des objets externes utilisés dans l'action, dans une table d'objets externes.

La table d'arguments formels d'actions contient des renseignements relatifs aux arguments de toutes les actions. Un élément de la table a la structure suivante :

ARGUF	TYPA	LMOT	LBIT	NATARG	CODAC

FIGURE B-7

où

ARGUF contient le nombre d'un argument,

TYPA contient le type de l'argument (broche, registre, mémoire, entier),

LMOT contient le nombre de mots d'un argument de type mémoire,

LBIT contient la longueur en bits d'un argument de type broche ou registre ou de chaque mot d'un argument de type mémoire,

NATARG indique la nature de l'argument (entrée, sortie, entrée et sortie),

CODAC contient le code de l'argument.

La table d'objets internes d'actions contient des renseignements sur les objets internes de toutes les actions existant dans la bibliothèque. Ces objets internes, d'après la définition du langage, ne peuvent être que de type entier. La structure d'un élément de la table est la suivante :

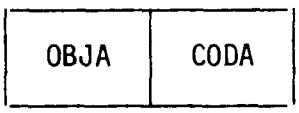


FIGURE B-8

où

OBJA contient le nom de l'objet interne,

CODA contient le code associé à l'objet.

La chaîne codée décrivant une action est composée d'un ensemble de triplets chaînés entre eux. La structure d'un triplet de l'ensemble est la suivante :

FIGURE B-9

La chaîne codée est construite par l'analyseur syntaxique du langage LEDA et contient des images d'instructions destinées à être interprétées. La signification de chacun des composants d'un triplet de la chaîne codée est la suivante :

NOI indique le numéro de l'instruction à laquelle appartient l'élément courant de la chaîne,

CHCODAC contient le code de l'élément courant de la chaîne (opérande ou opérateur).

SUIAC contient l'adresse du triplet suivant.

La table d'objets externes contient des renseignements sur les objets externes de chaque action figurant dans la bibliothèque. Un objet externe est un objet figurant dans le corps d'une action et ne faisant partie ni des arguments ni des objets internes de l'action. Un élément de la table a la structure suivante :

FIGURE B-10

où :

EXT contient le nom de l'objet externe,

CODEX contient le code associé à cet objet.

3) LA BIBLIOTHEQUE DE SCHEMAS D'INSTRUCTIONS.

L'utilisation de cette bibliothèque est facultative. Elle ne sert qu'aux prototypes pouvant exécuter des instructions-machine et se servant de schémas d'instructions pour décrire le fonctionnement de ces instructions. La bibliothèque de schémas d'instructions est composée d'une table des matières comportant des renseignements sur toutes les instructions de chaque prototype et d'une chaîne codée de schémas d'instructions à laquelle on accède par l'intermédiaire de la table. La structure d'un élément de la table des matières est la suivante :

.../...

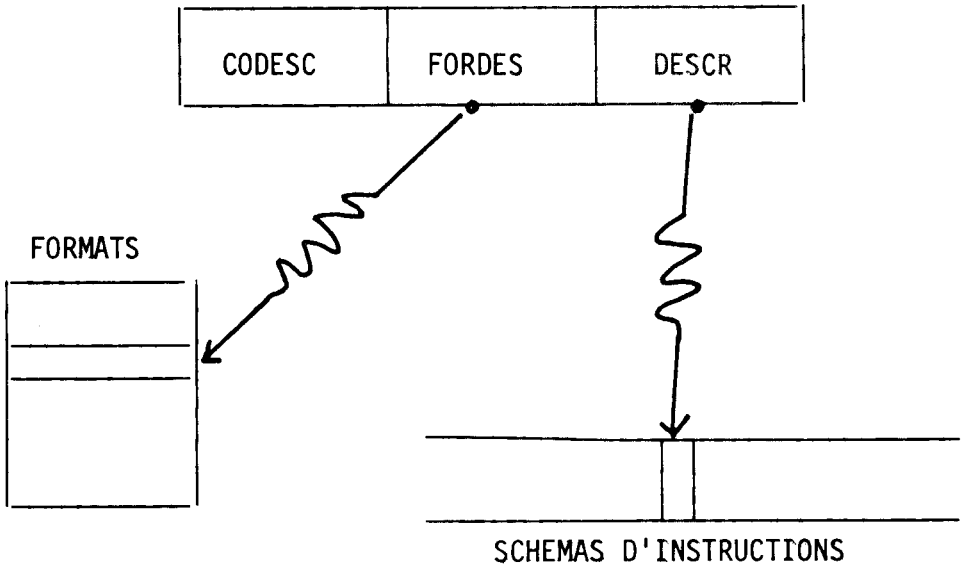


FIGURE B-11

où :

CODESC contient le code opération de l'instruction-machine décrite dans la bibliothèque,

FORDES contient le code du format de l'instruction-machine ce qui permet d'accéder par associativité à la table des formats que nous avons déjà décrite,

DESCR contient le point d'entrée dans la chaîne codée de schémas d'instructions.

L'accès à la table des matières de la bibliothèque de schémas d'instructions se fait en deux temps :

a) Accès direct à la première instruction-machine d'un prototype par l'intermédiaire du champ INST de la table des matières de la bibliothèque des prototypes.

b) Accès associatif à une instruction précise par l'exécution de l'instruction FAIRE dans une description écrite en LEDA.

La chaîne codée de schémas d'instructions fait partie de la chaîne codée d'instructions dont on a déjà décrit la structure.

B.2.- LA BIBLIOTHÈQUE TEMPORAIRE.

La bibliothèque temporaire est une bibliothèque provisoire où l'on peut ranger une unité composée décrite par le descripteur des connexions d'une description écrite en LEDA. Ranger une unité composée dans la bibliothèque provisoire c'est conserver un certain nombre de renseignements concernant la structure physique et logique de l'unité, tels que les liaisons, les bornes d'entrée et de sortie de ses sous-unités, le niveau logique de chaque unité, l'ensemble de ses sous-unités, l'ensemble de ses unités-frères et, enfin, le nom du prototype décrivant le fonctionnement de chaque unité simple.

La bibliothèque temporaire est composée d'un répertoire d'unités et d'une table des liaisons.

1) LE REPERTOIRE D'UNITES

Le répertoire d'unités contient des renseignements concernant la structure lofique de l'unité composée et de ses sous-unités ainsi que leurs bornes d'entrée et sortie et le nom du prototype correspondant à chaque unité simple. La structure d'un élément du répertoire d'unités est la suivante :

.../...

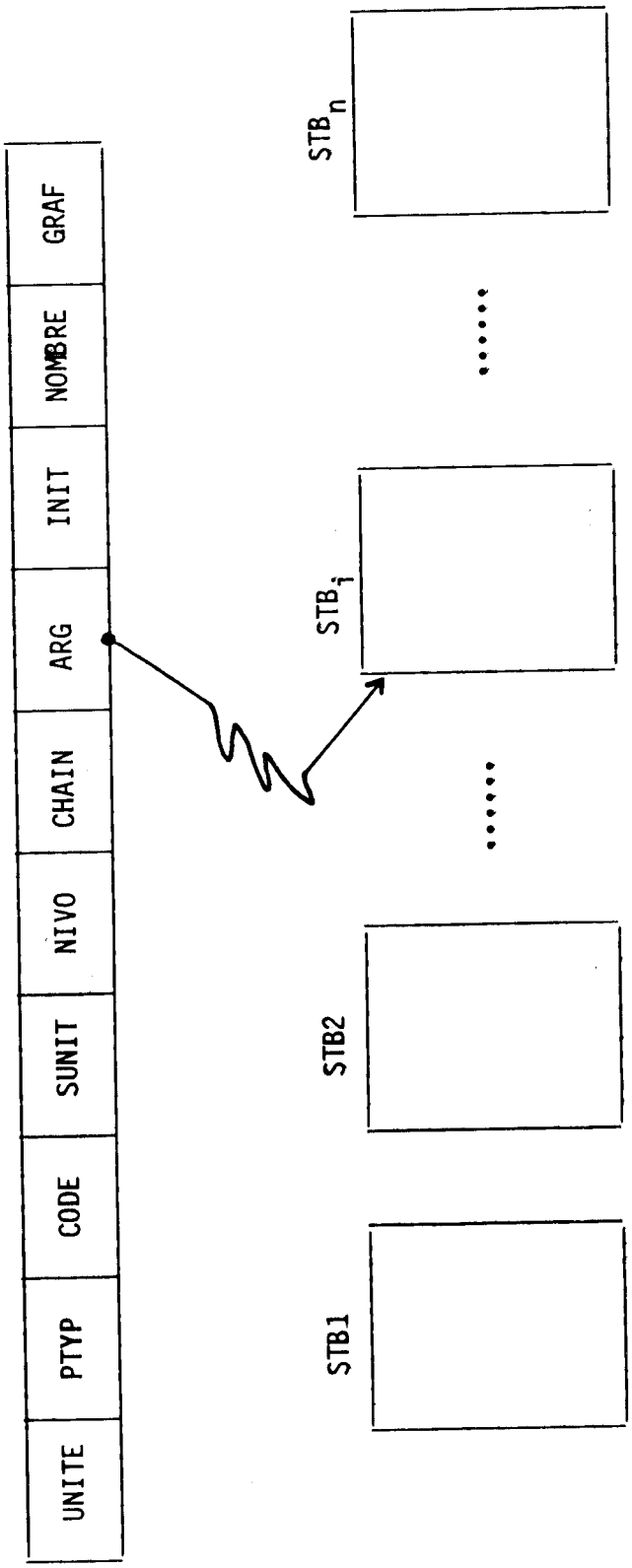

UNITE
PTYP
CODE
SUNIT
NIVO
CHAIN
ARG
INIT
NOMBRE
GRAF
STB1
STB2
STB$_i$
STB$_n$

FIGURE B-12

où :

UNITE contient le nom de l'unité,

PTYP contient le nom du prototype associé à l'unité lorsqu'il s'agit d'une unité simple,

CODE contient le code associé à l'unité logique,

SUNIT contient l'indice de la "première" sous-unité de l'unité,

NIVO contient le niveau logique de l'unité dans l'arborescence,

CHAIN contient un indice qui sert à chaîner entre elles les unités-frères,

ARG sert d'aiguillage vers une sous-table contenant des renseignements relatifs aux bornes d'entrée et de sortie (arguments) de l'unité (voir figure IV-12),

INIT sert à indiquer s'il s'agit d'une unité d'initialisation de l'unité composée,

NOMBRE indique le nombre d'unités strictement identiques à l'unité courante parmi ses unité-frères,

GRAF est utilisé par la primitive de génération graphique du système.

Ainsi, le répertoire d'unités est en fait une représentation de l'arborescence correspondant à l'unité composée, munie d'un certain nombre de renseignements complémentaires. Cette représentation utilise le chaînage d'une unité avec sa première sous-unité et ses unités-frères, ce qui donne la structure de la figure B-13

.../...

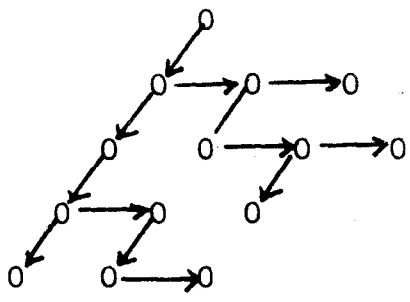


FIGURE B-13

On accède à tous les éléments d'une telle structure, à l'aide d'une pile.

Les sous-tables de bornes sont constituées chacune d'un certain nombre d'éléments ayant tous la même structure. La structure d'un élément de sous-table de bornes est la suivante :

BROCHES	BTYPE	INF	SUP

FIGURE B-14

où :

BROCHES contient le nom d'une borne ou d'un ensemble de bornes consécutives. Ce nom doit figurer dans la table des liaisons.

BTYPE indique le type de la borne ou de l'ensemble de bornes (entrée, sortie, entrée et sortie, événement, commande),

INF et SUP servent à définir une tranche de l'ensemble de bornes regroupées sous le nom BROCHES. Ils indiquent respectivement le rang (en commençant à 0) du premier et

dernier fil de la tranche. Le nombre de fils de la tranche est égal à SUP - INF + 1 .

2) LA TABLE DE LIAISONS

La table de liaisons sert à représenter tous les fils assurant les connexions physiques de la configuration. Certains de ces fils peuvent être regroupés sous le même nom et, dans ce cas, le nombre de fils correspondant à ce nom est indiqué. Cette table est construite à partir de renseignements fournis par le descripteur des connexions de la configuration, écrit en LEDA.
Un élément de la table a la structure suivante :

LIAIS	INTEX	BINF	BSUP	VALLI	APRES

FIGURE B-15

où :

LIAIS contient le nom sous lequel sont regroupés un ou plusieurs fils,

INTEX indique le type du fil par rapport à l'ensemble de la configuration (interne, entrée, sortie, entrée et sortie)

BINF et BSUP servent à indiquer le nombre de fils regroupés sous le nom contenu dans le champ LIAIS. Ce nombre est égal à BSUP - BINF + 1.

VALLI contient à chaque instant la valeur temporaire affectée à la liaison,

APRES a été défini pour les besoins de la simulation. Durant une unité de temps, la valeur de VALLI est constante et tous les changements portent sur la valeur d'APRES. A la fin de l'unité de temps, la valeur d'APRES est rangée dans VALLI.