

Analyse de modèles écrits en CASCADE et superviseur du système de simulation associé

Yvon Bressy

▶ **To cite this version:**

Yvon Bressy. Analyse de modèles écrits en CASCADE et superviseur du système de simulation associé. Langage de programmation [cs.PL]. 1985. dumas-00316038

HAL Id: dumas-00316038

https://dumas.ccsd.cnrs.fr/dumas-00316038

Submitted on 2 Sep 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

2 1 9X1

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE AGREE
DE GRENOBLE (C.U.E.F.A)

MEMOIRE

présenté en vue d'obtenir

le Dilôme d'Ingénieur C.N.A.M.

en

INFORMATIQUE

par

Yvon BRESSY

ANALYSE DE MODELES ECRITS EN CASCADE ET SUPERVISEUR DU SYSTEME DE SIMULATION ASSOCIE.

soutenu le 28 mai 1985

Les travaux relatifs au présent mémoire ont été effectués au laboratoire ARTEMIS,

Unité Associée au C.N.R.S. n°396

sous la direction de Monsieur Jean MERMET

REMERCIEMENTS AUX MEMBRES DU JURY

Je tiens à remercier Monsieur J.Y. RANCHIN,
Professeur au Conservatoire National des Arts et Métiers, d'avoir bien voulu présider le Jury de ce mémoire,

et suis reconnaissant à

Messieurs L. BOLLIET,
Professeur, Directeur du GIS de Mini et Micro Informatique de Grenoble ;

J. DOUSSY,
Ingénieur, Responsable du Service Logiciel de SEMS-BULL à Echirolles ;

C. LE FAOU,
Ingénieur C.N.R.S., Chef du Projet CASCADE ;

J. MERMET,
Maître de Recherche C.N.R.S.,
Directeur du Laboratoire ARTEMIS

d'avoir bien voulu accepter de faire partie de ce Jury.

REMERCIEMENTS

Je tiens à remercier Monsieur Jean MERMET, Maître de Recherche au C.N.R.S. et Directeur du Laboratoire ARTEMIS, d'avoir bien voulu accepter la préparation de ce mémoire dans son Laboratoire.

J'exprime ma reconnaissance à Monsieur Claude LE FAOU, Ingénieur C.N.R.S. et Chef du Projet CASCADE, avec qui j'ai toujours eu le plus grand plaisir à travailler.

Je ne saurais oublier Madame Dominique BORRIONE pour ses conseils tout au long de ce projet, et Monsieur Patrice UVIETTA pour son aide précieuse.

Que ma gratitude aille aussi à l'ensemble du "Groupe CASCADE" pour son rôle dans la réalisation de ce travail, qui n'est qu'une partie d'un immense effort collectif.

Mesdames Claudine MEYRIEUX, Josiane CARRY et Francette BONNABAUD ont bien voulu assurer la frappe de ce mémoire. Qu'elles soient remerciées pour le soin et la compétence qu'elles ont apportés à ce travail, ainsi que le Service Reprographie de l'IMAG pour la qualité du tirage qu'il en a effectué.

S O M M A I R E

CASCADE est un système de modélisation de circuits basé sur un langage de description. Il comprend aussi l'ensemble des logiciels permettant de vérifier les modèles écrits a l'aide de ce langage, tant sur le plan statique au moyen du compilateur que sur le plan dynamique au moyen du simulateur. Le langage de description est unique et multi niveaux, du niveau électrique au niveau "système". Il permet la modélisation hybride jusqu'à la simulation mixte discrète-continue. Sa caractéristique principale est qu'il permet d'exprimer directement les aspects structurels et de comportement des circuits, y compris le parallélisme de leur fonctionnement.

Le langage CASCADE offre de nombreuses nouveautés par rapport à l'état de l'art actuel. Il représente une synthèse de notions très modernes, dispersées dans d'autres langages. C'est, par exemple, un langage fortement typé, regroupant des types "matériels" et des types "valeurs". Ces derniers, à comportement ensembliste, sont partiellement définissables par l'utilisateur sous forme de types énumérés ou à propriétés caractéristiques.

Le travail considéré ici est la conception des mécanismes généraux de traitement des modèles écrits en CASCADE.

Plus précisément, il consiste tout d'abord en l'organisation de la première phase de compilation, indépendamment de toute application, puis à l'intérieur de cette phase, en le développement du Vérificateur Syntaxique et Sémantique par l'élaboration des traitements des différentes notions du langage et la définition des structures de données internes correspondantes.

Une contribution est en outre apportée à la mise en oeuvre de la Simulation. Il s'agit de la conception d'un Superviseur assurant l'enchaînement des opérations élémentaires de simulation et réalisant, à l'aide d'un noyau de primitives spécifiques, les fonctions de communication du modèle avec son environnement et l'utilisateur.

TABLE DES MATIERES

pages

INTRODUCTION 13

CHAPITRE I : LE PROJET CASCADE 15

I.1. DESCRIPTION FONCTIONNELLE 16
I.1.1. CASCADE 16
I.1.2. Autres fonctions, offertes par l'environnement CASCADE 24

I.2. DESCRIPTION STRUCTURELLE DU NOYAU CASCADE 26
I.2.1. Structure de la chaîne Modélisation-Simulation 27
I.2.2. Compilateur pour simulation de modèles mixtes hiérarchisés. 28

I.3. LES OUTILS UTILISES POUR LA REALISATION 33
I.3.1. Générateur d'analyseurs syntaxiques 33
I.3.2. Outils de communication homme-machine 36

PARTIE A PREMIÈRE PHASE DE COMPILATION 39

CHAPITRE II : ETUDE DES NOTIONS DU LANGAGE CASCADE, DÉFINITION DES MÉCANISMES D'ANALYSE DES MODÈLES EN VUE DE LEUR MISE EN OEUVRE 43

II.1. INTRODUCTION 43

II.2. STRUCTURE DU LANGAGE CASCADE 44

II.3. GRAMMAIRE CASCADE 45

II.4. ANALYSE LEXICALE 45

II.5. UNITES LEXICALES, NOTATIONS 47

II.5.1. Identificateurs 47
II.5.2. Mots clés 48
II.5.3. Constantes littérales 48
II.5.4. Commentaires 49
II.5.5. Constantes numériques 49
II.5.6. Constantes numériques 50
II.5.7. Symboles simples et doubles 53

II.6. LE LANGAGE CASCADE ET SA MISE EN OEUVRE 53

II.6.1. Généralités 53
II.6.2. La modularité 54
II.6.3. Axiome de la grammaire CASCADE 56
II.6.4. Les Compléments de Langage et l'ordre lanref 57
II.6.5. Les descriptions 63
II.6.6. Valeurs et porteuses, types 65
II.6.7. Les fonctions et procédures 72
II.6.8. Les attributs 74
II.6.9. Les interfaces 76

II.6.10. Les tableaux 77
II.6.11. Les descriptions externes 82
II.6.12. Les déclarations internes 82
II.6.13. Description des réseaux : les exemplaires de descriptions et l'ordre "unité" (use) 85
II.6.14. Les assertions 96
II.6.15. La partie fonctionnelle d'une description 97
II.6.16. Les formes conditionnelles "si" et "cas" (instructions et expressions) 98
II.6.17. Instruction répétitive : boucle "pour" 99
II.6.18. Les connexions d'unités 100
II.6.19. Les expressions, opérandes et opérateurs 104
II.6.20. Les transferts 116

II.7. LA REALISATION DES NIVEAUX DE LANGAGE DANS CASCADE 118

CHAPITRE III : ORGANISATION DE LA PREMIÈRE PHASE DE COMPILATION CASCADE ET TRAITEMENTS DES VÉRIFICATIONS 121

III.1. ORGANISATION DE ϕ1, DEFINITION ET PRINCIPE DE TRAITEMENT DES DONNEES 122

III.1.1. Principes de Compilation des modèles écrits en langage CASCADE 122
III.1.2. Organisation de la première phase de compilation 124

III.2. LE VERIFICATEUR 136

III.2.1. Mise en oeuvre de la tâche 136
III.2.2. Structures de données : définition et principe de fonctionnement 149

PARTIE B SUPERVISEUR DE SIMULATION 177

CHAPITRE IV : RÉALISATION DU SUPERVISEUR DE SIMULATION CASCADE 180

IV.1. DEFINITION DU JEU DE COMMANDES 180

IV.2. STRUCTURE ET FONCTIONNEMENT DU SUPERVISEUR 181

IV.3. FONCTIONS D'ACCES A LA SOS 183

IV.3.1. Fonction de localisation d'un module dans la SOS 183

IV.3.2. Fonction de localisation d'un descripteur de porteuse dans la SOS 186

IV.3.3. Fonction de localisation de la zone valeur d'une porteuse 186

IV.3.4. Autres fonctions 187

IV.3.5. Compléments 188

IV.4. FONCTIONNALITE DETAILLEE DES COMMANDES, TRAITEMENT 192

IV.4.1. BOX et WAB 192

IV.4.2. STO 194

IV.4.3. PRI 199

IV.4.4. TRA et ETR 202

IV.4.5. Autres commandes 204

IV.4.6. Extension du jeu de commandes 206

IV.5. EXEMPLE D'UTILISATION DU SUPERVISEUR DE SIMULATION 209

IV.6. PROGRAMMATION 218

CONCLUSION 219

ANNEXES

ANNEXE 1 : Grammaire de vérification de la syntaxe CASCADE 221

ANNEXE 2 : Les expressions 251

ANNEXE 3 : Réalisation des niveaux CASSANDRE et LASCAR du langage CASCADE 257

ANNEXE 4 : Codes internes des unités lexicales 269

ANNEXE 5 : Exemple d'éléments contenus dans le Descripteur Principal d'une Description 277

BIBLIOGRAPHIE 283

INTRODUCTION

L'informatisation et l'automatisation de la société dans tous les domaines amènent les industries et laboratoires à concevoir des circuits de plus en plus complexes. Cette évolution rend indispensable le recours à de nouveaux outils de CAO. Le Système Intégré CASCADE, permettant le suivi du processus global d'élaboration des ensembles électroniques, répond à cette complexité.

Traitant des circuits électroniques en général, CASCADE met en oeuvre les mécanismes nécessaires à l'enchaînement des phases de conception et à la manipulation intéractive des grandes quantités d'informations mises en jeu, limitant ainsi le risque d'erreur humaine.

Son organisation est celle d'un système unique, ouvert à l'intégration de tout outil adapté à des phases de la conception, et assurant tout au long des différentes étapes la gestion des modèles, la cohérence de leurs différentes représentations et celles de leurs données et résultats associés. Il est bâti sur un seul langage de description, couvrant tous les niveaux de modélisation, du niveau architecture système au niveau électrique fin, et permettant la modélisation mixte (Mer.83). Le langage CASCADE, permettant d'exprimer, entre autres caractéristiques, la structure et le comportement des circuits en cours de conception, est pour l'utilisateur le moyen principal de communication avec les différentes composantes du système CAO (Bor.85b).

CASCADE et son environnement en cours de développement comprennent :

- un langage de description multi-niveaux,
- un ensemble de simulateurs combinés en un seul, multi-mode hiérarchisé,
- un éditeur graphique, travaillant sur des éléments structurels,
- un compilateur logique, laissant les choix stratégiques à l'utilisateur,

des outils tels que :

- un langage de modélisation de pannes,
- un simulateur concurrent, pour la simulation de pannes,
- un générateur de séquences de tests, limité aux PLA,
- un générateur de plans-masse, complété par des compilateurs de silicium et des programmes d'implantation symbolique débouchant sur la production de masques.

Le logiciel CASCADE est actuellement développé sur VAX 11/780, sous système VMS. Une version fonctionnant sous système UNIX est prévue. Ecrit en FORTRAN 77, CASCADE doit pouvoir être porté sur d'autres machines, telles que SM90 et APPOLO.

Le travail relaté dans ce document porte sur la conception des mécanismes généraux d'analyse des modèles écrits à l'aide du langage de description de circuits CASCADE (partie A), et sur la réalisation du superviseur de simulation associé (partie B).

Le trait d'union existant entre ces deux interventions, contribuant à la réalisation du système, est qu'elles ont toutes deux un lien avec l'expression externe des modèles :

- leur définition dans le premier cas,
- la référence à leurs différents éléments, au cours de leur activation et de leur observation, dans le second.

Avant d'exposer ce travail en détail dans (A) et (B), nous situons ce dernier au sein du projet, dans le chapitre qui suit. Les principaux outils intervenant souvent dans la réalisation y sont aussi présentés.

CHAPITRE I

LE PROJET CASCADE

Le Projet CASCADE (*), dont la Maîtrise d'Oeuvre appartient au Laboratoire ARTEMIS, est soutenu par le CNRS et l'Agence De l'Informatique. Il fait actuellement l'objet d'un contrat multinational entre les Communautées Européennes, le Laboratoire ARTEMIS, le Politecnico di Torino (autre laboratoire universitaire, en Italie), et six industriels concepteurs de circuits : TMC (Angleterre), RTC, CTI, TRT (France), APT (Pays-Bas) et SGS (Italie). Ce dernier financement a pour but le développement d'un environnemênt CASCADE, CERES (**), que l'on peut voir figure I.2 en I.1.2 (Mer.85).

(*) CASCADE : *Conception Assistée de Systèmes et de Circuits Analogiques et Digitaux Electroniques.*

(**) CERES : *CASCADE Environment for the Realization of Electronic Systems.*

I.1. DESCRIPTION FONCTIONNELLE

Nous présentons ici les fonctions du Système CASCADE et de son Environnement réalisées par notre groupe. Nous évoquons ensuite, plus succinctement, les plus importantes parmi celles prises en charge par nos partenaires (Mer.85).

I.1.1. CASCADE

I.1.1.1. UN LANGAGE POUR LA MODÉLISATION

La caractéristique principale du Système est l'approche Langage. Le noyau de CASCADE est constitué autour du langage de description qui permet d'exprimer directement les aspects structurels et de comportement des circuits, y compris le parallélisme de leur fonctionnement. En effet, un système digital peut être arbitrairement décomposé en un réseau de modules interconnectés, constituant ainsi une structure arborescente d'une profondeur quelconque. Chaque noeud de l'arbre est une instance d'un module de description qui est soit défini indépendamment, soit prédéfini. Chacun de ces modules est une description a part entière, et chaque sous-noeud de l'arbre peut être considéré comme un modèle à part entière. D'une manière générale, un module est composé des deux parties suivantes : une partie structurelle décrivant la décomposition en sous-modules et leurs interconnexions, et une partie comportementale décrivant les relations entre les objets de l'interface du module et/ou les objets locaux du module, sous forme d'équations, d'algorithmes, ou d'actions parallèles ou séquentielles. On peut donc identifier deux parties distinctes dans le langage, un langage de description de réseaux et un langage de description comportementale. Une troisième partie vient se superposer aux premières, il s'agit d'un langage de description de conditions et de propriétés temporelles permettant de prendre en compte aisément les évènements dans la modélisation.

Le langage CASCADE est un langage générique permettant de décrire systèmes et circuits à tous les niveaux d'abstraction souhaitables. Il est inspiré des travaux du groupe CONLAN (CONsensus LANguage) (PBB.83, Bor.81, Kre.84) qui avait pour but la définition d'un standard et d'une formalisation rigoureuse en matière de langages de description de matériels. Notons pourtant que CONLAN ne couvre pas le niveau électrique de description des circuits, ni les niveaux physiques où il faut décrire la géométrie. Enfin, l'aspect modèles mixtes, que nous verrons plus loin, n'y est pas abordé.

Concrètement, le langage CASCADE est une famille de langages définis autour d'une syntaxe unique et d'une sémantique unique. Chaque langage de cette famille appelé niveau de langage, est défini par restriction de cette syntaxe et de cette sémantique, sachant que la plupart des notions sont communes à tous ou à plusieurs niveaux (Bor.85b).

Cet important aspect du langage est un facteur d'intégration certain, car il permet d'assurer plus aisément la cohérence sémantique des différentes représentations du circuit au cours du processus de conception, ainsi que des données qui lui sont associées. En effet, lorsqu'on passe d'un niveau à un autre, on ne change pas de langage, ni de compilateur et de structure de données interne.

Six langages prédéfinis, associés à différents niveaux de modélisation existent actuellement.

Niveau de modélisation	Niveau de langage
Système	LASSO
Comportemental	LASCAR
Transfert de registres	CASSANDRE
Logique ou "Porte logique"	POLO
Transistor ou "Interrupteur"	CASTOR
Electrique	IMAG

Dans ce tableau, chaque niveau est une abstraction du niveau inférieur. Les niveaux bas sont ceux où les circuits sont décrits d'une manière fine, proche du comportement technologique du système. Le haut de l'échelle correspond aux niveaux abstraits, où l'on s'éloigne des phénomènes physiques en faisant des approximations sur le comportement temporel et logique du système. Mais si en prenant de la hauteur, on perd en précision on gagne en efficacité.

Par rapport à la modélisation :

- le niveau système correspond au niveau architectural ou algorithmique ; les modèles étant exprimés, par exemple, en termes de graphes de contrôle ;
- le niveau transfert de registres se présente comme une charnière entre le structurel et le comportemental ;
- au niveau logique, ou PORTES, le circuit décrit est vu comme un réseau de portes interconnectées. Il est purement structurel ;
- le niveau transistor ("switch" en anglais) est une approximation du niveau inférieur, en discrétisant les conductances ;
- le niveau électrique ou analogique est celui où l'on représente un réseau de transistors par un modèle mathématique.

Notons que LASSO (BoG.79, BoG.80, Gra.81), LASCAR (Bor.76), CASSANDRE (Mer.73, Bre.75a, Bre.75b) et IMAG (Lef.74) existaient déjà sous forme de logiciels indépendants, sans aucun lien entre eux, sauf pour CASSANDRE et LASCAR, le second étant une extension du premier.

Le langage CASCADE offre de nombreuses nouveautés par rapport à l'état de l'art actuel. Il représente une synthèse de notions très modernes, dispersées dans d'autres langages. C'est par exemple, à tous niveaux, un langage :

- fortement typé, regroupant des types "matériels" et des types "valeurs". Ces derniers, à comportement ensembliste sont partiellement définissables par l'utilisateur sous forme de types énumérés ou à propriétés caractéristiques ;

- généralisant la notion de variable à la notion de porteuse, objet d'un type matériel contenant des valeurs ;

- permettant l'appel de fonctions et procédures ;

- permettant à l'utilisateur, par la notion de "complément de langage" d'étendre la définition du niveau de langage pratiqué et de définir ainsi son propre langage ;

- permettant, par la notion d'attributs, de paramétriser les descriptions par exemple en dimension et sur les grandeurs temporelles (attributs structurels et attributs comportementaux) ;

- mettant à la disposition de l'utilisateur un mécanisme d'assertions caractérisant formellement le fonctionnement souhaité du modèle, dans un langage autre que le langage de description ;

- permettant enfin, et c'est l'un des points forts du langage, de mélanger dans un même modèle des descriptions écrites à des niveaux de modélisation différents. Les interfaces entre ces descriptions sont alors gérés automatiquement. Cette mixité permet d'avoir par exemple, un effet de loupe en observant de façon très fine une partie du circuit dans un environnement moins précis.

I.1.1.2. UNE VÉRIFICATION STATIQUE ET DYNAMIQUE DES MODÈLES

CASCADE comprend l'ensemble des logiciels permettant de vérifier les modèles écrits à l'aide du langage, tant sur le plan statique au moyen du compilateur, que sur le plan dynamique au moyen du simulateur. Le principe même de ce dernier n'impose pas la pratique ou la cohabitation de niveaux particuliers. Des modèles mixtes, généraux, constitués de parties écrites à n'importe quels niveaux de modélisation, sans restriction,

sont admis à condition que leur cohabitation ait un sens (Exemple : mélange des deux niveaux discrets CASSANDRE et POLO). Le simulateur est donc aussi conçu pour fonctionner, si nécessaire, en multi-modes, c'est-à-dire pour dérouler des simulations discrètes-continues. Ce dernier type de simulation correspond aux circuits dont une partie est décrite à un niveau électrique, où l'on considère le temps comme une variable à évolution continue, et l'autre décrite à un niveau logique où le temsp progresse de manière discrète.

Il est à remarquer que les simulateurs dits multi-niveaux existants font cohabiter deux ou parfois trois niveaux de modélisation donnés voisins. Dans CASCADE, seule les performances de la simulation peuvent limiter le nombre de niveaux considérés à un instant donné.

Le compilateur transforme les descriptions d'entrée en une structure interne hiérarchisée unique, simulable. La simulation met alors en oeuvre plusieurs mécanismes (BHL.83) :

a) un algorithme de séquencement interprèté pour le parcours de la hiérarchie interne du modèle,

b) des algorithmes de simulation entièrement compilés pour les parties fonctionnelles du modèle,

   * algorithmes de simulation discrète pour les parties décrites aux niveaux logiques et système,
   * algorithmes de simulation continue pour les parties décrites au niveau électrique.

Tous ces algorithmes coexistent, pilotés par un mécanisme de gestion de temps continu-discret.

L'efficacité de la simulation est obtenue dans CASCADE par un processus d'ordonnancement des actions à dérouler durant la simulation. Le modèle est découpé en une liste ordonnée de blocs, contenant des listes ordonnées d'instructions. Les parties impossibles à ré-arranger à cause d'un couplage mutuel fort sont automatiquement isolées et font l'objet d'un traitement par un algorithme de stabilisation.

D'autre part, en simulation de circuits, seulement une faible partie du circuit simulé est active en un instant donné. Ce fait est aussi exploité dans CASCADE en s'appuyant sur le découpage en blocs précédent.

I.1.1.3. Un Editeur Graphique

L'Editeur Graphique CASCADE est un moyen pour représenter d'une manière plus visuelle l'information que l'on peut définir avec le langage textuel CASCADE, dans tous ses niveaux. Là encore, la représentation graphique des modèles est partagée en deux parties :

- la description structurelle, qui donne une vision de la structure de l'objet ;
- la description comportementale qui permet de décrire le fonctionnement de l'objet et les contraintes à respecter. Ceci lorsqu'il est possible de définir une sémantique graphique sur les primitives comportementales que l'on désire représenter (Mar.84).

Les moyens graphiques de CASCADE sont généralisés aussi au niveau de la simulation où il sera possible de visualiser la structure interne des modèles en s'appuyant sur la nouvelle hiérarchie induite par le découpage en blocs. On peut alors envisager de représenter clairement (par la couleur par exemple) l'activité/inactivité des modèles en cours de simulation.

Enfin, CASCADE est conçu pour être compatible avec beaucoup d'autres produits graphiques afin que des utilisateurs possédant déjà des postes de travail graphiques puissent les conserver.

I.1.1.4. Un Compilateur Logique

La compilation logique mise en oeuvre dans CASCADE, a pour but de transformer un circuit digital spécifié au niveau RTL (CASSANDRE, LASCAR) en un réseau de portes (niveau POLO) au fonctionnement équivalent.

Cette transformation d'une spécification essentiellement comportementale d'un circuit en une description purement structurelle tient compte :

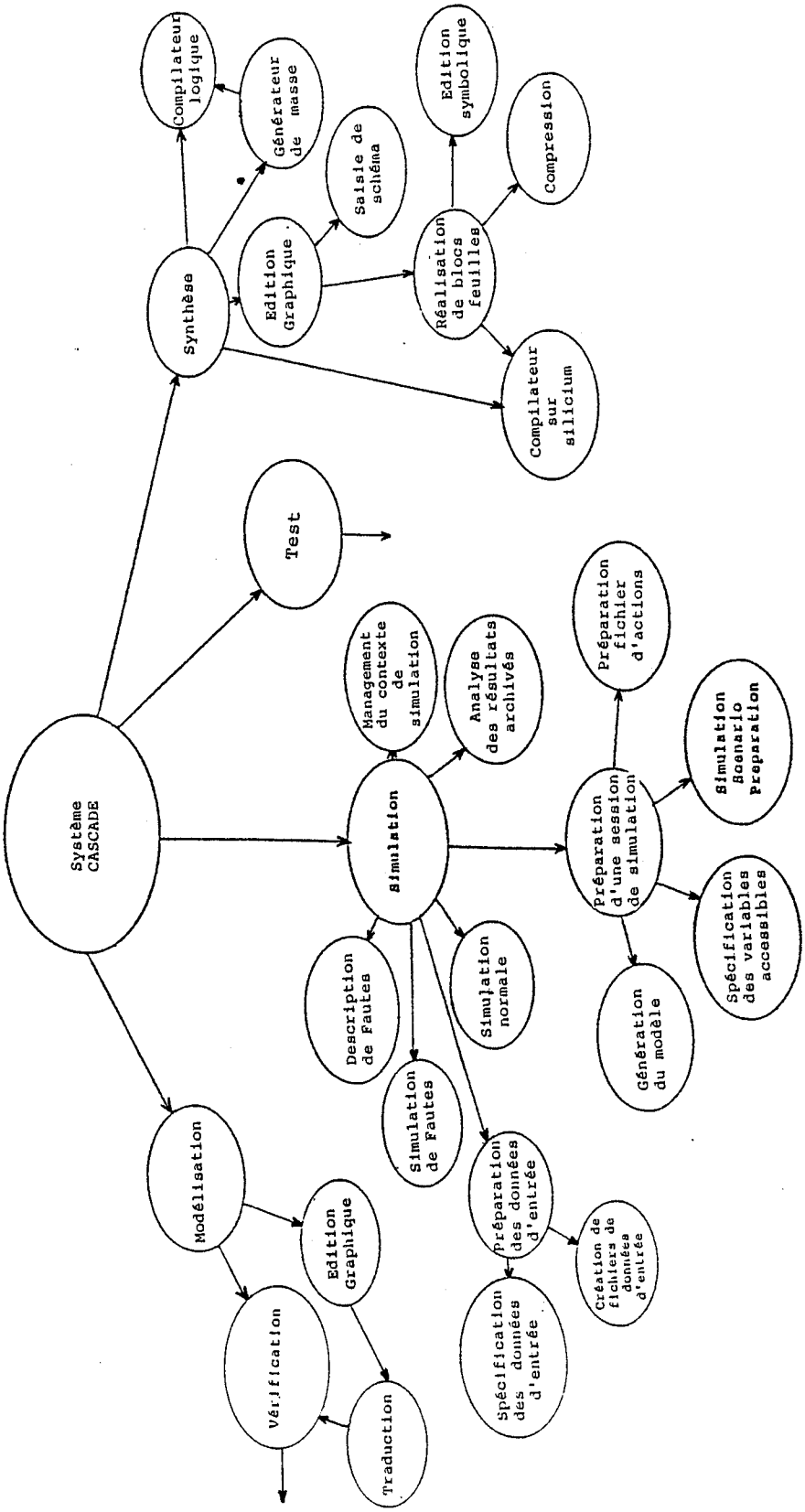
- de la sémantique des langages de départ et d'arrivée,
- des contraintes de la technologie spécifique choisie par l'utilisateur,
- des éléments prédéfinis disponibles dans chaque cas particulier,
- de la stratégie de conception utilisée.

Le système n'est pas dédié à une architecture cible particulière. Sa vocation est assez générale (Gate arrays, Macrocells, circuits imprimés,...) et impose à l'outil d'être intéractif. Le choix est laissé à l'utilisateur entre un ensemble de solutions fonctionnellement équivalentes (architecture bus ou distribuée...), l'outil effectuera automatiquement les transformations requises par celui-ci. L'utilisateur a accès au circuit après chaque étape de synthèse, à la fois textuellement (langages POLO, CASSANDRE), et graphiquement (Editeur Graphique CASCADE) (Dur.84).

I.1.1.5. UNE ORGANISATION EN SYSTÈME INTÉGRÉ

Après avoir vu des facteurs d'intégration caractéristiques de CASCADE, tels que le langage de description, le simulateur multi-niveaux multi-mode hiérarchisé et l'Editeur Graphique, nous en venons ici à des aspects plus classiques de l'intégration.

L'espace de conception est organisé en un réseau hiérarchisé d'environnements (voir figure I.1 ci-après), dans lequel on se déplace à l'aide du langage de commande. Ce dernier est unique pour tout le système et hiérarchisé de façon similaire au réseau d'environnements. Au niveau de chaque environnements, l'utilisateur dispose d'un menu, élément terminal du langage de commande hiérarchisé. Le déplacement d'un environnement, ou sous-environnement, à un autre permet ainsi l'enchaînement des phases



RESEAU PARTIEL DES ENVIRONNEMENTS CASCADE

Figure I.1

du processus de conception (CDMS : CASCADE Design Management System). On notera, dans la figure I.1 proposée, et à titre d'exemple, les environnements de modélisation, simulation, test et synthèse ; ces deux derniers seront vus dans le paragraphe I.1.2.

Enfin, la connexion de CASCADE avec une base de données relationnelle est à l'étude. Ce qui permettrait d'assurer la cohérence des données et la gestion des projets, alternatives, documentation, etc...

I.1.2. AUTRES FONCTIONS, OFFERTES PAR L'ENVIRONNEMENT CASCADE

On peut voir figure I.2 ci-après les principaux composants de l'environnement CASCADE. Nous terminons cette description fonctionnelle du système en évoquant rapidement ceux qui concernent les pannes et la synthèse des circuits.

I.1.2.1. MODÉLISATION ET SIMULATION DE FAUTES, LE TEST

Un langage de description de pannes et son simulateur associé (simulateur concurrent) sont prévus pour être intégrés au système CASCADE. Conçu pour traiter plusieurs milliers d'évènements par secondes au niveau portes logiques, le but recherché est un simulateur multi-niveaux pour accélérer la propagation des symptômes de fautes à travers le réseau. On utilise, par exemple un niveau de description fonctionnelle pour la propagation des symptômes de fautes et le niveau de description réseaux de portes pour la description des modules qui comportent des pannes.

La génération de séquences de tests prévue est limitée, dans le projet CEE-CERES, au test des PLA.

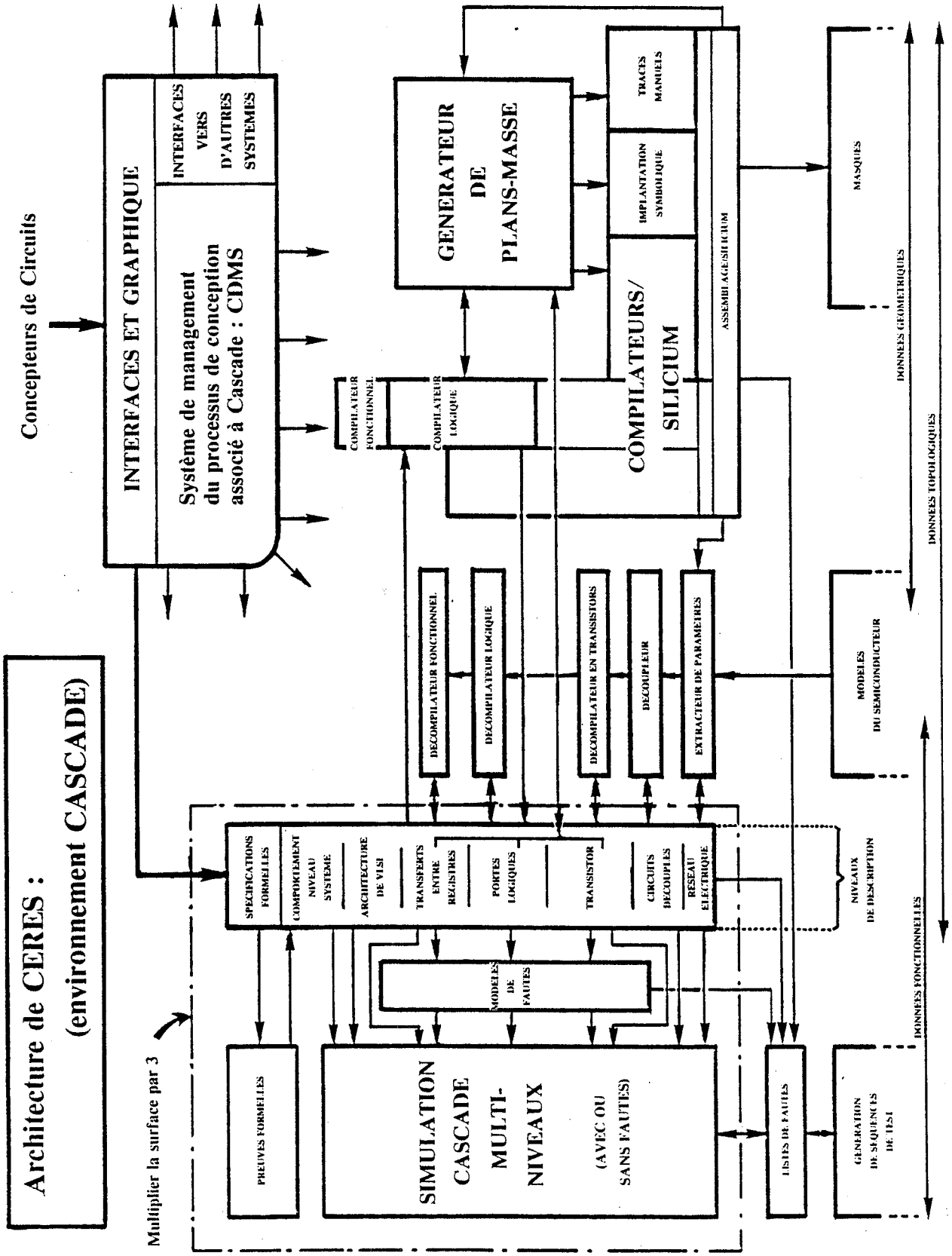

Architecture de CERES :
(environnement CASCADE)
Concepteurs de Circuits
INTERFACES ET GRAPHIQUE
Système de management du processus de conception associé à Cascade : CDMS
INTERFACES VERS D'AUTRES SYSTEMES
GENERATEUR DE PLANS-MASSE
TRACES MANUELS
IMPLANTATION SYMBOLIQUE
COMPILATEURS/ SILICIUM
ASSEMBLAGE/SILICIUM
MASQUES
COMPILATEUR FONCTIONNEL
COMPILATEUR LOGIQUE
DONNEES GEOMETRIQUES
DONNEES TOPOLOGIQUES
DECOMPILATEUR FONCTIONNEL
DECOMPILATEUR LOGIQUE
DECOMPILATEUR EN TRANSISTORS
DECOUPLEUR
EXTRACTEUR DE PARAMETRES
MODELES DU SEMICONDUCTEUR
SPECIFICATIONS FORMELLES
COMPORTEMENT NIVEAU SYSTEME
ARCHITECTURE DE VLSI
TRANSFERTS ENTRE REGISTRES
PORTES LOGIQUES
TRANSISTOR
CIRCUITS DECOUPLES
RESEAU ELECTRIQUE
NIVEAUX DE DESCRIPTION
MODELES DE FAUTES
Multiplier la surface par 3
PREUVES FORMELLES
SIMULATION CASCADE MULTI-NIVEAUX
(AVEC OU SANS FAUTES)
LISTES DE FAUTES
GENERATION DE SEQUENCES DE TEST
DONNEES FONCTIONNELLES

Figure I.2

I.1.2.2. OUTILS DE SYNTHÈSE (Mer.85)

Un environnemnet de synthèse est celui d'où résulte la description physique du circuit. On y opère partitionnement fonctionnel, traduction du comportement en structure, dimensionnement, placement, interconnexion, cela sur les différentes couches qui définissent un circuit (imprimé ou intégré). Le but du découpage en fonctions est de séparer dans la description du système le hardware du reste et de le répartir en un certain nombre de morceaux intégrables.

CASCADE peut servir de support à ces processus de synthèse, en utilisant de façon naturelle les mécanismes de compilation. Dans cette perspective le Compilateur de Silicium n'est que le dernier maillon, aboutissant au dessin des masques, de toute une famille possible de compilateurs qui permettent de passer d'un niveau de représentation à un autre tout en restant au sein de CASCADE. Pendant ce processus de synthèse, on dispose ainsi à chaque instant du modèle simulable de ce que l'on est en train de concevoir, ce qui permet de vérifier la cohérence de la conception quand elle n'est pas produite automatiquement par programme mais "fait main" par l'utilisateur.

Enfin, tout au long de ce traitement, le rôle du générateur de plans-masse (floor planner) est d'associer une structure géométrique, créée par le concepteur ou par programme, à toute description comportementale d'un morceau du circuit à concevoir. Il ne décrit que des polygones.

I.2. DESCRIPTION STRUCTURELLE DU NOYAU CASCADE

La figure I.3 montre l'enchaînement des modules de traitement pour la chaîne Modélisation-Simulation.

I.2.1. STRUCTURE DE LA CHAÎNE MODÉLISATION - SIMULATION

C'est sur cette structure que peuvent se raccorder les autres outils de conception déjà évoqués.

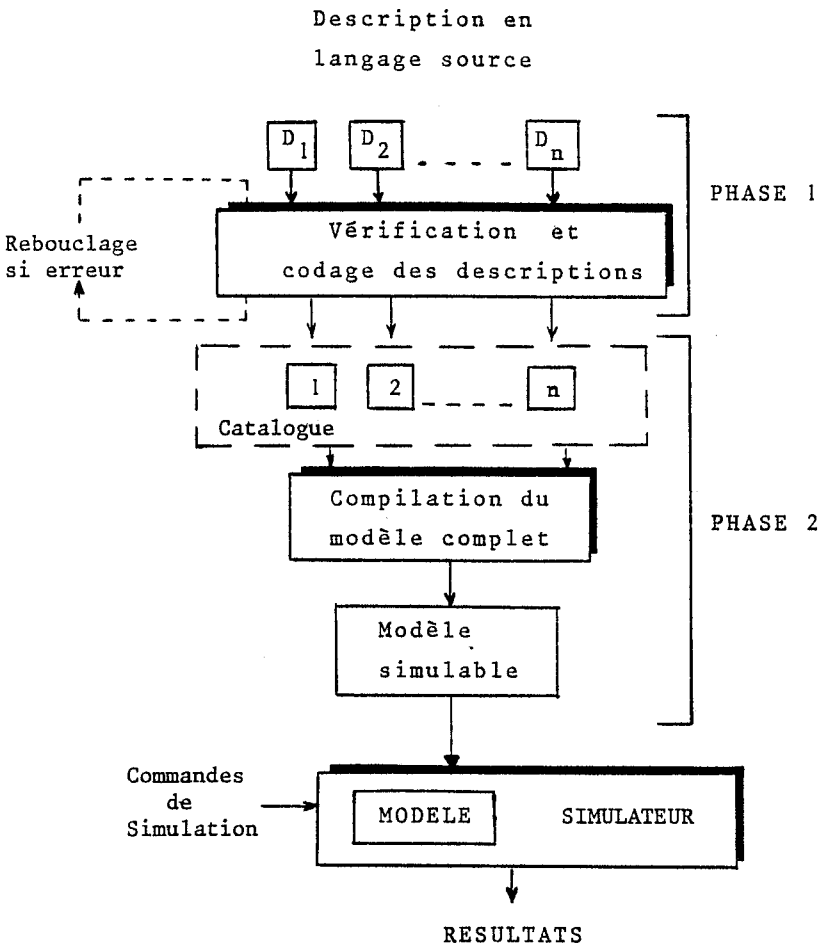


Figure I.3

En CASCADE, tout modèle de circuit se présentant comme un ensemble de descriptions des modules qui composent ce circuit, nous distinguerons les modules de traitement travaillant sur les descriptions séparément en les considérant les unes après les autres, de ceux qui travaillent sur le modèle global. D'où la distinction entre la première phase de compilation ($\phi 1$) et la seconde ($\phi 2$).

La première phase admet donc en entrée les descriptions écrites en langage CASCADE, par l'utilisateur. Elle effectue la vérification statique de ces blocs et les traduit en structures internes stockées dans le Catalogue. A ce stade là, certains paramètres peuvent ne pas être encore fixés dans les descriptions (descriptions génériques).

La phase $\phi 2$ de compilation, orientée vers la simulation, extrait du catalogue les blocs précédemment stockés par $\phi 1$, nécessaires pour construire le modèle complet. Ce dernier, après une série d'importantes modifications (Lef.85), est chargé en mémoire avec le simulateur et le tout constitue une maquette virtuelle du circuit pour le ou les niveaux de modélisation choisis.

L'utilisateur converse alors avec le modèle et le simulateur à l'aide d'un langage de commande approprié.

Le travail personnel, objet de ce mémoire, portant sur la phase $\phi 1$ de compilation et le simulateur, ces modules de traitement sont détaillés à cette occasion dans les chapitres suivants. Seule la phase $\phi 2$ de compilation est donc exposée dans le présent chapitre, de description du projet CASCADE.

I.2.2. COMPILATEUR POUR SIMULATION DE MODÈLES MIXTES HIÉRARCHISÉS (figure I.4)

Le premier programme activé dans la phase $\phi 2$ de compilation est le Constructeur de Modèles (Gen.83). A partir du nom de la description du module le plus englobant du circuit ou du système modélisé, il puise dans le catalogue l'ensemble des descriptions nécessaires (Lef.84c) et construit la première structure interne de $\phi 2$, la Structure Du Projet (SDP)(Lef.84b). Cette dernière est constituée de l'arbre des imbrications des modules et de l'ensemble de leurs descriptions. C'est au cours de ce traitement que la connexion des interfaces des différents modules, jusqu'ici traités séparément, est effectivement réalisée.

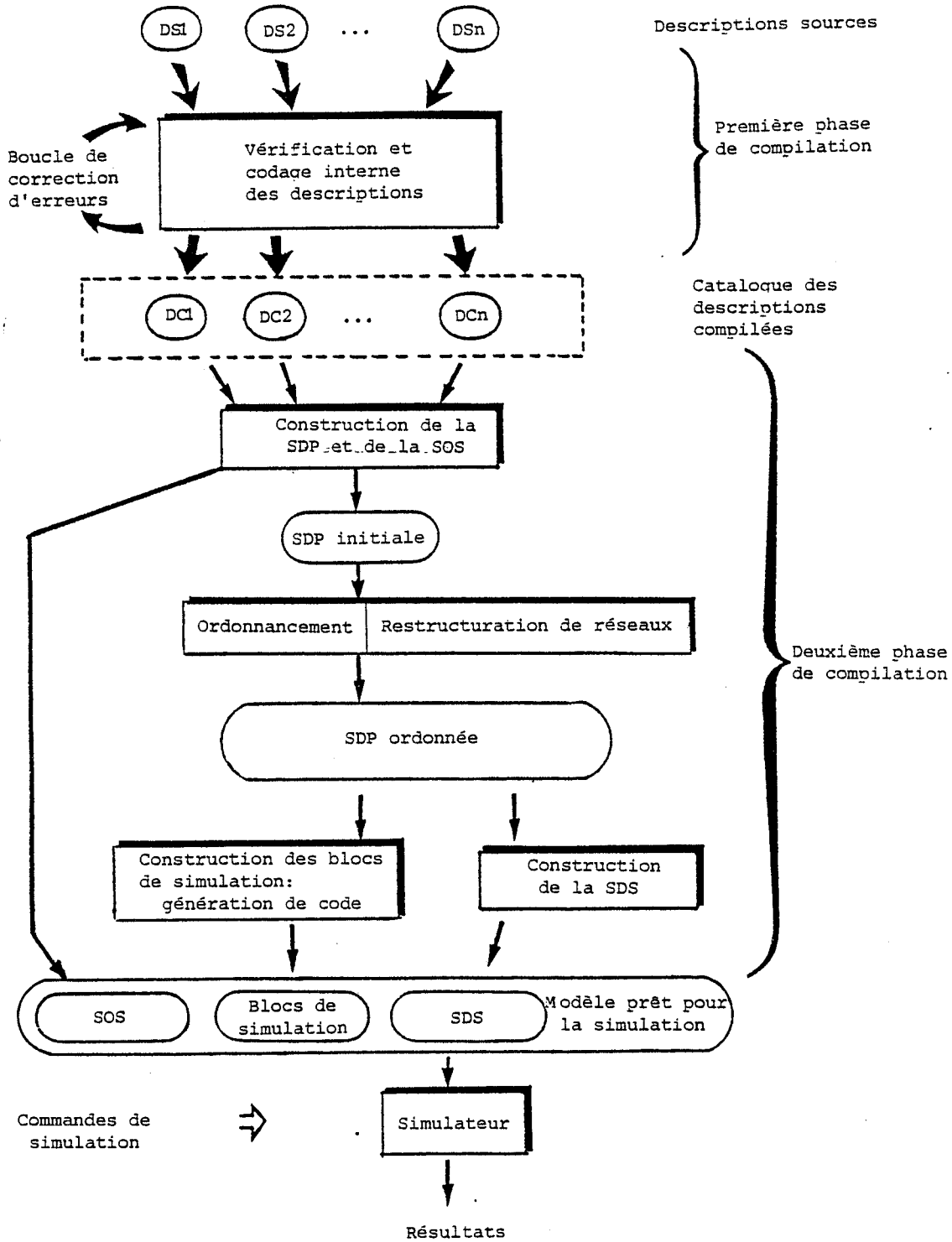

DS1
DS2
...
DSn
Descriptions sources
Boucle de correction d'erreurs
Vérification et codage interne des descriptions
Première phase de compilation
DC1
DC2
...
DCn
Catalogue des descriptions compilées
Construction de la SDP et de la SOS
SDP initiale
Ordonnancement
Restructuration de réseaux
SDP ordonnée
Construction des blocs de simulation: génération de code
Construction de la SDS
Deuxième phase de compilation
SOS
Blocs de simulation
SDS
Modèle prêt pour la simulation
Commandes de simulation
Simulateur
Résultats

figure I.4

Le modèle construit peut être mono-niveau ou multi-niveaux. Le Catalogue pouvant contenir la description d'un même module plusieurs fois mais modélisé à des niveaux d'abstraction différents, c'est à ce stade du traitement que sont constituées les différentes "alternatives" d'un même modèle. On obtient ainsi des "sur-modèles" où chacun des modules composant le circuit peut apparaître plusieurs fois, à des niveaux de modélisation différents. C'est ultérieurement, avant de simuler effectivement le système, que l'utilisateur doit faire son choix et "couper" ainsi les branches inutiles du sur-arbre que constitue alors le modèle.

Parallèlement à la production de la SDP, une Structure Orientée Simulation (SOS) est générée (Rou.84b). Elle a pour rôle de permettre l'accès à tout objet du modèle sous sa forme externe et ainsi de connaître ou forcer la valeur de cet objet (Rou.84a). Nous reviendrons là-dessus au chapitre IV car cette SOS est une structure interface qui contribue à la communication entre l'utilisateur et le modèle en cours de simulation, par le biais du Superviseur.

Il faut maintenant générer un modèle simulable de telle façon que la simulation soit efficace.

A un instant donné de la simulation, la détermination du comportement du modèle se fait en évaluant toutes les expressions et en exécutant toutes les fonctions décrites à cet instant. Une propriété fondamentale des circuits est que l'on peut souvent trouver un ordre dans ces actions. Ce qui entraîne qu'on n'a à les exécuter qu'une seule fois. D'où une simulation particulièrement rapide puisqu'on évite le rebouclage sur les calculs.

Une deuxième propriété importante des circuits, vient encore permettre d'accélérer la simulation. Il s'agit du fait qu'à un instant donné de son fonctionnement, une partie du circuit seulement "bouge". On dit qu'elle est active, le reste étant dit inactif, par opposition. Il n'est alors pas utile d'évaluer le fonctionnement de cette dernière partie.

Pour profiter concrètement de ces propriétés, on s'appuie sur un découpage en blocs du circuit. Nous ne nous étendrons pas sur les critères à prendre en compte dans un tel découpage : choix des blocs, de leur taille, etc..., nous dirons seulement qu'à chaque niveau d'imbrication de la hiérarchie de départ, on cherche à se ramener à des blocs interconnectés à fonctionnement découplé. Ainsi, à chaque instant de simulation, en parcourant l'arbre de départ en partant du niveau le plus externe, il existe un ordre, fonction des liaisons inter-blocs, dans l'évaluation de ces blocs.

En conséquence de ce que nous venons dénoncer, nous avons été conduits à déterminer un algorithme d'ordonnancement, à fonctionnement récursif, travaillant couche par couche dans l'arbre d'imbrication des modules du modèle pris comme découpage en blocs de départ (Hum.84, Dec.84). A chaque niveau de l'arbre, cet algorithme fournit une liste ordonnée de blocs quand ceci est possible. Toutes les instructions ont été descendues au niveau d'imbrication le plus bas, sous forme de listes ordonnées. Bien entendu, à n'importe quel niveau, il peut arriver que certaines parties soient impossibles à ré-arranger, à cause d'un couplage mutuel fort. Ces parties sont alors automatiquement isolées et traitées par un algorithme de stabilisation.

Ainsi, au cours de son travail, l'algorithme d'ordonnancement peut être amené à modifier l'imbrication initiale des modules. Cette restructuration est effectuée à l'aide de deux opérations fondamentales (Gen.84) :

- ENVELOPPER, qui crée de nouveaux blocs à partir d'anciens et/ou de variables.

- ECLATER, qui a pour effet de faire disparaître des contours de modèles.

Après ce traitement, le modèle se présente sous la forme d'un arbre interprétable d'où sera tiré le modèle simulable final. Toute la partie fonctionnement a été descendue dans les feuilles, sous forme de séquences

ordonnées. Les noeuds non terminaux ne représentent plus que du structurel. A tous les noeuds sont associées les informations suivantes :

- activité / inactivité de la structure sous-jacente,
- pour les feuilles seulement, algorithme à utiliser pour l'évaluation de ce noeud (électrique, logique,...),
- descripteurs de fonctions d'interface, dans le cas où ces noeuds communiquent avec des noeuds voisins, modélisés à des niveaux d'abstraction différents (modèles mixtes).

Les séquences ordonnées d'instructions constituant les feuilles sont compilées et exécutées directement à la simulation. Elles sont appelées Blocs de Simulation (BS). Actuellement, elles sont générées par le module Générateur de Code, en FORTRAN, pris comme langage intermédiaire (BDG.84).

La structure finale obtenue est appelée Structure De Simulation (SDS) (Lef.84a).

Notons que suivant les niveaux de langage considérés, ce traitement et ses résultats peuvent présenter des particularités, nous n'en citerons que quelques unes :

- au niveau Portes Logiques (POLO) par exemple, il n'y a pas de partie fonctionnement, mais uniquement du structurel. Les feuilles de la SDS sont des circuits élémentaires prédéfinis, les portes, et le code généré associé se réduit à une exploitation de tables de vérité,
- au niveau Transfert de Registres (CASSANDRE et LASCAR), l'algorithme de simulation est lui aussi compilé dans les BLocs de Simulation, avec le fonctionnement,
- au niveau électrique (IMAG), les algorithmes de simulation — les méthodes de résolution des sytèmes algébro-différentiels contenus dans les BS — sont externes aux modèles et font partie intégrante de la base d'algorithmes du simulateur (partie statique),

  etc...

Nous donnons, en début de chapitre IV, un rapide aperçu du fonctionnement du simulateur sur les différents éléments produits par la phase $\phi 2$ de compilation que nous venons de voir.

I.3. LES OUTILS UTILISES POUR LA REALISATION

Un logiciel du volume et de la complexité de CASCADE peut être difficilement développé sans le recours à des outils d'aide à l'écriture de systèmes. C'est pourquoi, nous avons utilisé un logiciel d'aide à la conception de compilateurs dans les phases $\phi 1$ et $\phi 2$ de compilation, et un moniteur de communication homme/machine dans tous les modules de traitement.

I.3.1. GÉNÉRATEUR D'ANALYSEURS SYNTAXIQUES

Le but recherché ici est la simplification et la systématisation de l'écriture des programmes de traitement des structures de données. Par programme de traitement, nous entendons par exemple un passage de compilation, tout ou partie d'un module de traitement CASCADE. L'un des moyens d'y parvenir est de mettre en évidence, quand cela est possible, une grammaire régissant ces données, carle réalisateur peut alors utiliser l'expression de la syntaxe comme langage de programmation, particulièrement adapté pour diriger ses algorithmes. Concrètement, il suffit de disposer d'un analyseur spécialisé pour la syntaxe donnée, vérifiant d'une part la conformité de la structure d'entrée par rapport à la grammaire considérée et générant d'autre part un flot d'appels, contrôlé, à des fonctions sémantiques qui effectuent les transformations nécessaires sur la structure de données fournie en entrée (CGV.80).

Des outils permettent d'obtenir automatiquement l'analyseur correspondant à une grammaire donnée, il s'agit des Transformateurs de Grammaires. Les grammaires admises par ces outils sont souvent des grammaires "hors contexte" ("context free" en anglais), et les analyseurs produits sont des analyseurs déterministes, descendants ou ascendants. Les premiers correspondent aux grammaires LL(1), les autres aux grammaires LR(K).

Avec les grammaires LL(1), l'analyse est descendante, de gauche à droite, sans retour arrière, déterministe sur un symbole unique, c'est-à-dire que la connaissance d'un seul symbole permet de décider quelle est la production d'une règle qui doit être choisie.

Avec les grammaires LR(K), l'analyse est ascendante, avec K symboles "dans la fenêtre" (K est en fait souvent égal à 1). Les méthodes LR sont les plus puissantes car si un langage permet une analyse déterministe, alors il est possible de trouver une grammaire LR(1) pour ce langage. La méthode LL(1), par contre, ne s'applique qu'à un sous-ensemble des langages déterministes ; on prouve même que toute grammaire LL(1) est LR(1), l'inverse n'étant évidemment pas vrai. En pratique, pourtant, la différence de puissance a peu d'importance, comme en témoigne l'usage très courant des analyseurs LL(1). En effet, les méthodes LR sont réputées pour être d'une plus grande complexité de mise en oeuvre. L'insertion de fonctions sémantiques dans les règles, par exemple, est aisée avec les grammaires LL(1), où elles peuvent être insérées partout, à condition de respecter l'ordre naturel d'arrivée des symboles, alors qu'avec les grammaires LR, elles ne peuvent être insérées qu'aux positions de réduction, c'est-à-dire en fin de règle. Quand le programmeur veut insérer une fonction au milieu d'une règle, il doit transformer la grammaire en coupant la règle en deux.

Notre choix s'est porté sur les grammaires LL(1) vu que nous avions l'habitude d'écrire les syntaxes sous cette forme ou sous d'autres facilement transformables en forme LL(1).

Le générateur d'analyseurs de ce type que nous avons choisi est le TRGLL1, développé à l'IMAG et existant en version FORTRAN (DeG.80). Il était le seul disponible dans notre environnement proche, lorsque le projet CASCADE a démarré. D'autre part, nous avions une forte expérience de cette lignée d'outils (15 ans) et nous verrons au paragraphe suivant qu'il est aussi utilisé dans le Moniteur de Dialogue que nous avons adopté.

CASCADE étant actuellement développé sur VAX 11/780-VMS, on peut s'interroger sur le choix que nous avons fait ; pourquoi le TRGLL1 qui ne peut être exécuté que sous MULTICS, et pas YACC qui tourne sur VAX ?

Tout simplement parce qu'alors YACC générait un analyseur en langage C uniquement et était conçu pour tourner sous système UNIX. D'autre part, il nous a semblé que les messages d'erreurs n'étaient pas toujours très clairs et que les analyseurs syntaxiques étaient plus gros. Enfin et surtout, nous n'avions aucune expérience d'utilisation de YACC.

Terminons ce paragraphe en donnant le schéma d'usage général du TRGLL1, en distinguant bien ce qui est construction du programme de traitement de ce qui est structure de ce dernier (figure I.5).

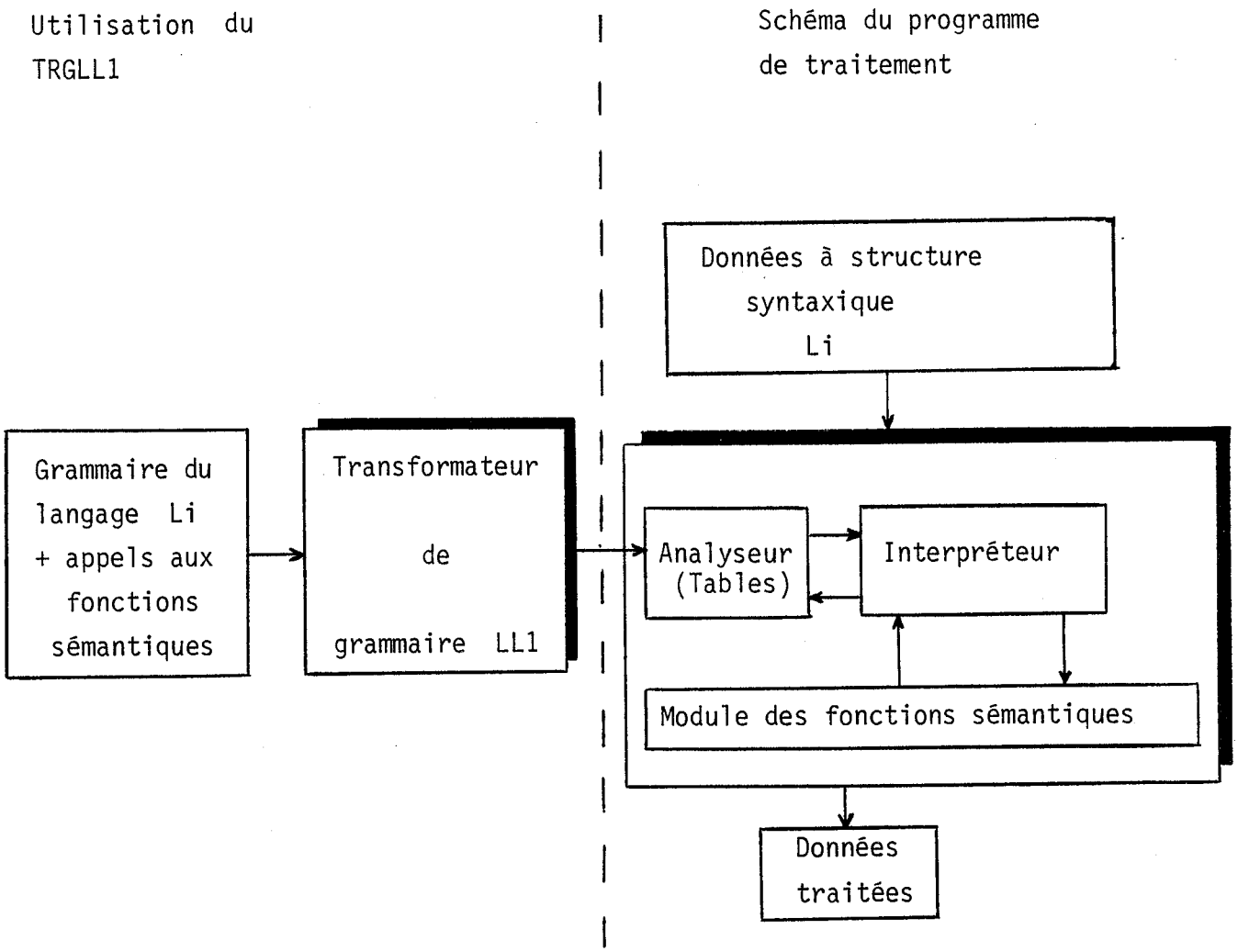


figure I.5

A tout traitement recherché, on fait correspondre une grammaire (langage L_i), qui, traitée par le TRGLL1 donne un analyseur. Cette phase des opérations relève de la conception du compilateur et est exécutée actuellement sous système MULTICS (CICG). L'analyseur est produit sous forme de tables interprétables, portables. Ces tables et l'interpréteur standard qui leur est associé (écrit en FORTRAN) constituent le squelette du programme de traitement ou de la partie du compilateur considéré. Un module constitué des fonctions sémantiques associées vient complèter le tout.

Ainsi, ce programme accepte des données à structure syntaxique L_i (des descriptions CASCADE, par exemple) et produit des données résultats, présentant éventuellement une structure syntaxique L_{i+1}, pouvant être traitées à leur tour par un autre programme, analogue, le passage suivant du compilateur par exemple.

I.3.2. OUTILS DE COMMUNICATION HOMME/MACHINE

Il s'agit d'un ensemble de deux moniteurs développés dans notre groupe (UIMS : User Interface Management System, DLU.84)

- un Moniteur de Communication pouvant assurer les entrées/sorties et la gestion des fichiers FORTRAN,
- un Moniteur de Dialogue assurant la mise en oeuvre des langages de commandes.

Le recours à ces outils permet une meilleure portabilité et intégration du produit.

Le Moniteur de Communication assure la correspondance entre les fichiers "logiques" créés et manipulés par les programmes de traitement, et les fichiers FORTRAN effectivement mis en oeuvre dans ce système programmé presqu'entièrement en FORTRAN.

Il est utilisé partout dans CASCADE.

Le Moniteur de Dialogue gère les intéractions. Il permet de remplacer la programmation des programmes intéractifs par leur spécification, suivie d'une programmation guidée des algorithmes. Ceci conduit à séparer l'aspect intéraction de l'aspect algorithmique, de spécifier le premier et de programmer manuellement le second. Puis de transformer les spécifications pour qu'elles puissent prendre place dans un cadre prédéfini, basé sur le principe du Moniteur de Dialogue, capable de gérer l'intéraction et de faire le lien avec les algorithmes.

Plus concrètement, disons que le concepteur de systèmes use d'un premier outil, le Langage de Spécification de Langages de Commande (LSLDC) pour spécifier de manière simple et agréable le dialogue qu'il entend mettre en oeuvre (syntaxe et appels à des fonctions de traitement). C'est la "mise en place de l'application". Le compilateur spécialisé du LSLDC, conçu, notons le au passage, à l'aide du TRGLL1 déjà évoqué au paragraphe précédent, produit alors la grammaire du dialogue de l'application. Cette dernière, passée au TRGLL1, produit à son tour l'analyseur du dialogue, sous forme de tables interprétables, qui joint à un interprêteur spécialisé pour le mode intéractif, et au module contenant les fonctions sémantiques de traitement, constitue le Programme Intéractif recherché.

Le schéma ci-après, figure I.6, montre la hiérarchie existant entre les différents programmes et données mis en jeu ici, et l'enchaînement des opérations effectuées lors de la mise en place d'une application.

Le Moniteur de Dialogue peut être utilisé pour la mise en oeuvre de l'ensemble du langage de commande de CASCADE. Il l'est actuellement pour le sous-langage de commande réduit au dialogue du Simulateur. Nous développerons ceci plus en détail au chapitre IV.

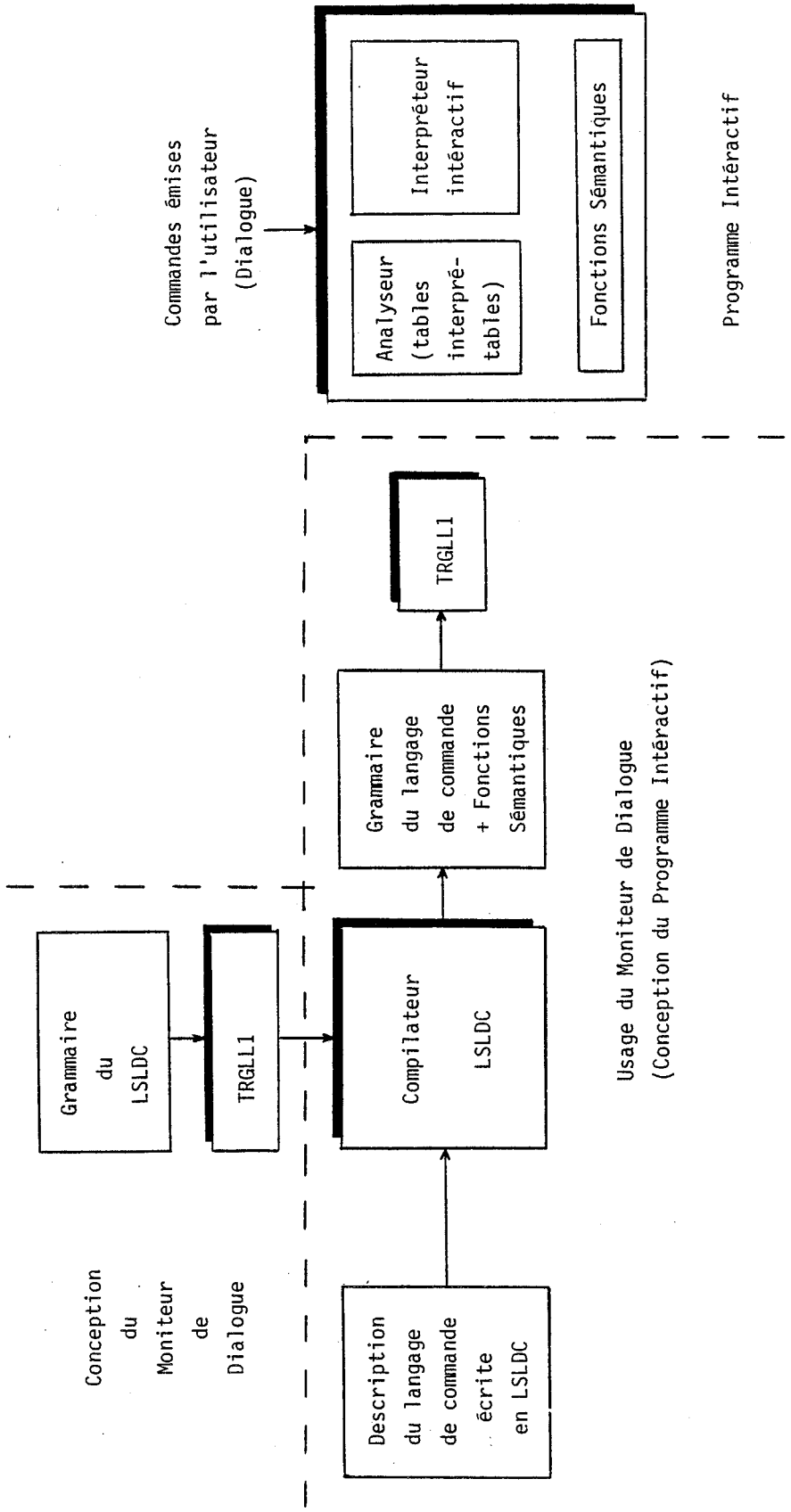
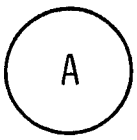

Commandes émises
par l'utilisateur
(Dialogue)
Programme Intéractif
Analyseur
(tables
interpré-
tables)
Interpréteur
intéractif
Fonctions Sémantiques
TRGLL1
Grammaire
du langage
de commande
+ Fonctions
Sémantiques
Grammaire
du
LSLDC
TRGLL1
Compilateur
LSLDC
Description
du langage
de commande
écrite
en LSLDC
Conception
du
Moniteur
de
Dialogue
Usage du Moniteur de Dialogue
(Conception du Programme Intéractif)

figure I.6



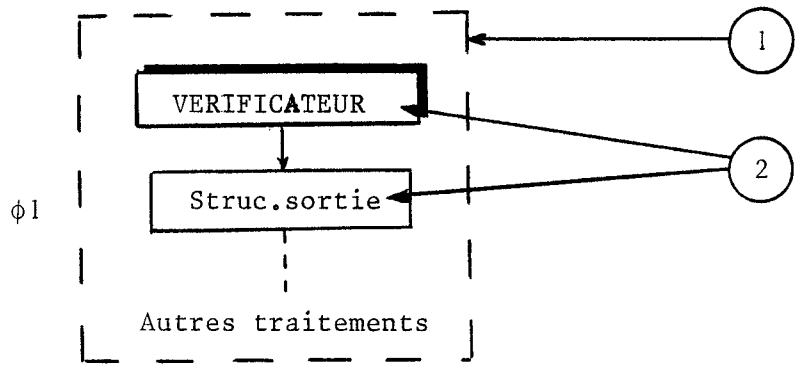
PREMIERE PHASE DE COMPILATION

Cette partie du document expose le travail personnel effectué dans la première phase de compilation CASCADE.

Le chapitre II consiste en l'étude des notions du langage de description et en la définition des mécanismes d'analyse des modèles en vue de leur mise en oeuvre.

Le chapitre III expose tout d'abord la définition des données mises en jeu dans $\phi 1$, les spécifications de leur traitement et l'organisation de $\phi 1$ qui en découle (1). On y trouve ensuite la mise en oeuvre de la tâche VERIFICATION : sa description et l'organisation des programmes qui la constituent, les spécifications des structures de données internes à cette tâche ou produites par elle, et les principes de fonctionnement associés (2).

Le schéma ci-dessous, situe le niveau des interventions (1) et (2) dans $\phi 1$.



Nous nous devons enfin de signaler ici que d'importants compléments à cette analyse ont été apportés par Monsieur Bernard CAUBET, et que la programmation de l'ensemble a été réalisée par ce dernier ainsi que par Madame Ginette BUISSON et Monsieur Jean-Paul GENEVOIS.

CHAPITRE II

ETUDE DES NOTIONS DU LANGAGE CASCADE, DÉFINITION DES MÉCANISMES D'ANALYSE DES MODÈLES EN VUE DE LEUR MISE EN OEUVRE

II.1. INTRODUCTION

Les traitements évoqués ici sont étroitements liés à la syntaxe et à la sémantique du langage et se situent au premier passage de la phase une de compilation, c'est-à-dire dans le VERIFICATEUR.

Nous accorderons une attention particulière aux mécanismes généraux, surtout à ceux qui permettent la génération des différents niveaux de modélisation (Bre.83c).

La description du langage donnée ci-après est orientée réalisation et n'est en rien le manuel de référence donnant la définition rigoureuse et complète de ce langage (Bor.84). D'autre part, s'il est fait état ici du premier prototype, les notions qui seront implantées ultérieurement seront tout de même évoquées pour des raisons de cohérence. Ce fait sera à chaque fois signalé.

II.2. STRUCTURE DU LANGAGE CASCADE

On dénombre en générale trois types de concepts dans les langages de programmation : les structures de contrôle, les structures de données et les opérations. Un langage peut offrir plus ou moins de possibilités dans chacune de ces trois catégories, en fonction du domaine d'application visé. Par exemple, un langage d'écriture de compilateurs est plus riche en outils de représentation ou de description de structures de données qu'un langage de programmation d'algorithmes numériques.

Un langage de description de circuits multi-niveaux tel que CASCADE n'est pas un langage de programmation mais offre de fortes possibilités suivant les trois concepts cités :

- les structures de contrôle, avec les automates, les graphes de contrôle, les instructions conditionnelles... et le TPDL, sous-langage pour la description d'évènements et de propriétés temporelles (assertions) (BMB.83),

- les structures de données, avec les types de porteuses, les types de valeurs... qui sont des constructions complexes,

- les opérations, qui sont très variées avec de nombreux opérateurs, généraux et spécialisés. La définition des expressions est très générale, avec une possibilité d'indexation puissante. Les instructions vont de l'équation mathématique, avec fonctions standards, à l'écriture explicite des chemins de données.

Le langage CASCADE est l'union de toutes les notions de description textuelle de circuits prises en compte dans le système. L'intérêt de CASCADE réside principalement dans le fait que la plupart de ces notions sont communes à tous ou à plusieurs niveaux. Défini par sa syntaxe et la sémantique associée, le langage est conçu autour d'une grammaire globale, unique, et tout niveau particulier est défini par restriction de cette syntaxe et de cette sémantique. On trouvera en annexe 1 la grammaire d'implantation du langage utilisée dans le Vérificateur. Cette dernière est munie de l'ensemble des appels aux fonctions sémantiques, appelées ici actions de compilation, définies à ce jour.

II.3. GRAMMAIRE CASCADE

Elle est écrite sous forme LL1 et est traitée par le TRGLL1 qui produit un analyseur déterministe descendant. Cet analyseur, couplé à un interprêteur standard, permet de vérifier la syntaxe de tout texte écrit en langage CASCADE. D'autre part, les actions de compilation, qui peuvent être appelées de n'importe où, dans les règles, effectuent les vérifications sémantiques nécessaires, ainsi que la génération de la structure de données et du code interne.

Notons que l'ordre suivi par l'exposé des notions du langage et de leur implantation , correspond à peu près à un parcours descendant de l'arbre associé.

II.4. ANALYSE LEXICALE

Un texte en langage CASCADE est écrit à l'aide de mots dont l'agencement, suivant la syntaxe définie par la grammaire, constitue des phrases. Ils représentent les symboles terminaux de cette grammaire, et nous les désignerons aussi par le terme "unités lexicales" car ils sont reconnus et codés par un sous-programme de l'interpréteur standard : l'analyseur lexicographique (ou lexical).

La lexicographie comporte donc les règles de formation des mots du texte CASCADE. Elle indique la manière de regrouper les caractères en mots.

Le choix du degré de complexité des unités lexicales, qui incombe aux concepteurs du compilateur, est important car il influe sur la lourdeur de la grammaire. En effet, ce qui est traité par lexicographie n'a pas à l'être par la syntaxe. Le compromis que nous avons adopté pour CASCADE est plutôt classique.

Nous nous sommes efforcés de concevoir les unités lexicales de telle façon qu'elles soient reconnaissables par simple étude locale, indépendamment du contexte syntaxique ou sémantique dans lequel elles se trouvent. Ce qui veut dire, en d'autres termes, que la fonction de reconnaissance, exécutée par l'analyseur, ne s'appuie jamais sur des paramètres ou indicateurs de contexte fournis à partir d'autres niveaux du compilateur (interpréteur standard, actions de compilation). L'analyse lexicographique pourrait ainsi fonctionner, si on le désirait, de manière autonome, complètement déconnectée du reste du compilateur et plus précisément de la syntaxe.

L'indépendance entre lexicographie et contexte est souhaitable afin d'éviter des difficultés (CGV.80). Considérons par exemple le fragment de texte :

```
A .= IF(X)...
```

Le mot IF y est ambigü. L'analyse LL1 ne permet pas de décider immédiatement de sa sémantique. Si ce texte est suivi du mot THEN, il s'agit alors d'une instruction conditionnelle, sinon c'est un identificateur de tableau.

Ceci implique que les mots clés doivent être identifiés par leur morphologie ou être des identificateurs réservés. Leur nombre relativement important, près de deux cents, nous a obligé à écarter la solution des identificateurs réservés. L'utilisateur aurait à mémoriser une trop grande liste de mots à éviter pour le choix de ses identificateurs. Malgré la lourdeur que cela introduit dans le graphisme, c'est donc par leur morphologie que les mots clés sont caractérisés.

Certains éléments prédéfinis du langage, des types de valeurs et de porteuses, en nombre réduit, sont représentés par des identificateurs prédéfinis. Nous justifierons ce choix lors de l'exposé de cette partie du langage.

II.5. UNITES LEXICALES, NOTATIONS

- Pour améliorer la lisibilité, les phrases du langage sont en format libre.

- Les CARACTERES RECONNUS sont :

• les lettres majuscules : A, ..., Z , et minuscules : a, ..., z , pour les machines qui les reconnaissent ;

• les chiffres : 0, ..., 9 ;

• la plupart des symboles spéciaux (suivant le niveau du langage) ;

• le blanc.

- Les BLANCS peuvent toujours apparaître entre les unités lexicales, et en quantité quelconque.

- Le BLANC est significatif dans certains cas, où il permet de détecter la fin d'une unité lexicale.

- Dans le texte d'entrée, la FIN DE LIGNE a la signification d'un blanc pour la lexicographie.

Voyons maintenant les différentes catégories d'unités lexicales.

II.5.1. IDENTIFICATEURS

Ce sont des suites de caractères alphanumériques de longueur limitée à la longueur de la ligne de texte (n paramétrable mais inférieur ou égal à 132 caractères) et commençant obligatoirement par une lettre. Le symbole _ (souligné) peut aussi être utilisé, non pas pour souligner un caractère alphanumérique, mais tout seul ("blanc souligné").

II.5.2. MOTS CLÉS

Ils sont prédéfinis pour chaque niveau de langage et s'écrivent avec les caractères alphanumériques, plus le souligné _ , le premier caractère étant alphabétique.

L'utilisation du souligné est ici la même que dans le cas des identificateurs.

DISTINCTION ENTRE MOTS CLÉS ET IDENTIFICATEURS

- Cas d'une machine admettant les majuscules et les minuscules :
  la partie alphabétique des identificateurs est en majuscules et celle des mots clés en minuscules.

  Exemple :

  description MULT (in HORLOGE H; in SIGBO EM[O : 15], DEPART
  out SIGBO SM[O : 31], OCCUPE)

- Cas d'une machine ne permettant pas l'utilisation des minuscules :
  dans ce cas, tout l'alphabet est en majuscules et les mots clés sont encadrés par le symbole " (guillemet).

  Exemple :

  "DESCRIPTION" MULT ("IN" HORLOGE H; "IN" SIGBO EM[O : 15], DEPART
  "OUT" SIGBO SM[O : 31], OCCUPE)

II.5.3. CONSTANTES LITTÉRALES

Ce sont des chaînes de caractères de longueur limitée à $n \leq 132$, encadrées par le symbole ' (apostrophe). Elles sont écrites avec n'importe quel caractère reconnu par la lexicographie, donc y compris le blanc.

Exemple :

'ZSn1748 +'

Il est aussi possible d'utiliser le délimiteur lui-même , à condition de le doubler.

Exemple :

'Ah ! quel beau temps aujourd''hui'

La chaîne vide '' est admise.

II.5.4. COMMENTAIRES

Ils sont placés entre deux délimiteurs : << et >> ("plus petit que" et "plus grand que" doublés) et sont écrits avec n'importe quel caractère reconnu par la lexicographie, donc y compris le blanc. Leur longueur est illimitée.

Exemple :

<<ceci est un commentaire>>

Un même commentaire peut tenir sur autant de lignes que l'on veut. Les commentaires ne sont pas significatifs en tant qu'éléments de description. Ils sont reconnus et traités par l'analyse lexicale qui les stocke dans le listing d'édition, mais ne les transmet pas au reste du compilateur qui les ignore donc.

II.5.5. CONSTANTES LOGIQUES

Elles se présentent comme un caractère alphanumérique, précédé du symbole "accent grave" ` . Le jeu de caractères permis est :

les chiffres : 0, 1, ..., 9
avec les lettres minuscules : a, b, ... , z , dans le cas des machines admettant les majuscules et les minuscules, et :
avec les lettres majuscules : A, B, ... , Z , dans le cas des machines n'admettant pas les minuscules.

Exemple :

`0, `1, `u, `z

Contrairement à tous les autres types de constantes, évoquées à la suite, ce n'est pas la constante logique qui constitue une unité lexicale, mais le vecteur de constantes logiques (éventuellement réduit à un seul élément).

Mis à part ce cas particulier, les tableaux de constantes, y compris ceux de constantes logiques à plus d'une dimension, sont des constructions syntaxiques élaborées à partir de ces unités lexicales et de certains symboles. Cette notion sera exposée plus loin.

Dans l'écriture des vecteurs de constantes logiques, il est possible, pour des raisons de simplification, d'omettre les accents graves autres que le premier.

. Exemple :

le vecteur `a`1`0 s'écrit `a10

La longueur de ces vecteurs est aussi limitée à la longueur de la ligne de texte ($n \leq 132$ caractères). De plus, aucun blanc n'étant permis dans leur écriture, il est à noter qu'ils ne peuvent jamais se situer à cheval sur deux lignes, puisque la fin de celles-ci est équivalente à un blanc.

II.5.6. CONSTANTES NUMÉRIQUES

Elles se divisent en deux classes : entières et réelles.

II.5.6.1. CONSTANTES ENTIÈRES

Elles se divisent à leur tour en deux sous-classes : celles avec base explicite et celles avec base par défaut.

BASE EXPLICITE

Il s'agit des constantes numériques entières binaires, octales ou hexadécimales. Elles se présentent comme une suite de chiffres existants dans la base en question, immédiatement précédée de # et de l'initiale de la base :

#b	pour binaire ;		#B	
#o	pour octal ;	ou	#O	suivant la machine
#h	pour hexadécimal ;		#H	

Exemple :

#b110101 ; #0670251 ; #HA10BF97B

BASE PAR DÉFAUT

Il s'agit des constantes numériques entières décimales. La base n'est pas mentionnée explicitement comme dans les cas précédents.

Exemple :

262635 1101

II.5.6.2. CONSTANTES RÉELLES

FORME SIMPLE

Il s'agit de deux constantes entières, décimales, séparées par un point, la première constituant la partie entière. Le point ne peut pas être utilisé comme premier ou dernier caractère de la constante.

Exemple :

12.345

OPTIONS

Les constantes réelles simples (ainsi que d'ailleurs les constantes entières), peuvent être complètées par deux options s'excluant mutuellement :

- OPTION PUISSANCE DE DIX :

  Un exposant de la forme E{+, -, rien} n vient complèter la constante où n est une constante entière.

  Exemple :

  123E-45 123.4E-5
  123.45E+06 123.45E6
  12.3E45 123E+4

  etc...

- OPTION FACTEUR D'ECHELLE (noté f.e.) :

  Ce facteur d'échelle permet une meilleure commodité d'écriture des constantes numériques. Le f.e. peut être :

le femto	noté F	(10 puissance - 15)
le pico	noté P	(10 puissance - 12)
le nano	noté N	(10 puissance - 9)
le micro	noté MC	(10 puissance - 6)
le milli	noté ML	(10 puissance - 3)
le kilo	noté K	(10 puissance 3)
le mega	noté MG	(10 puissance 6)
le giga	noté G	(10 puissance 9)

  Exemple :

  123P
  12.34ML

II.5.7. SYMBOLES SIMPLES ET DOUBLES

Comme les mots clés, ils sont définis pour chaque niveau de langage.

Exemple :

(,) ; + := ⇐ etc...

II.6. LE LANGAGE CASCADE ET SA MISE EN OEUVRE

II.6.1. GÉNÉRALITÉS

Les méthodes de modélisation mises en oeuvre ici sont basées sur le principe suivant : le système à modéliser doit pouvoir être vu comme un ensemble de modules interconnectés, formant une structure arborescente. Pour un modèle donné, le nombre de ces modules est fixe durant toute simulation.

Cette propriété de modularité se traduit par la notion de description, entité de programme descriptif d'un module (ou d'une famille de modules si la description est paramétrée par des attributs).

En plus de la possibilité d'écrire des descriptions, l'utilisateur peut définir et utiliser des "compléments de langage". Cette notion, détaillée ultérieurement, permet de rendre globales et implicites des définitions de descriptions, fonctions, procédures et types.

Une autre propriété importante du langage est la règle de "non référence avant". Pour éviter un traitement trop complexe, les références avant sont interdites : tout objet doit être déclaré avant d'être utilisé, tout segment doit être partiellement ou totalement défini avant d'être référencé. Cette contrainte nous facilite la réalisation de la vérification du langage en un seul passage.

Dans certains cas particuliers où le contexte est très fort, des déclarations implicites sont admises : les noms d'états d'automate, les indices de boucles "pour", etc...

Toutes ces caractéristiques du langage CASCADE ont une influence immédiate sur les principes de traitement dans la phase $\phi 1$ de compilation.

II.6.2. LA MODULARITÉ

Les systèmes traités avec CASCADE sont vus sous deux aspects, complémentaires l'un de l'autre : structurel et fonctionnel. La structure modulaire de ces systèmes peut être de complexité arbitraire et est prise en compte à tous les niveaux de modélisation. Chaque module correspond à un noeud de l'arbre associé au système.

La propriété principale qui en résulte est qu'à chaque niveau de l'arborescence, n'importe quel sous-arbre forme un tout structurellement et fonctionnellement, c'est-à-dire que c'est un modèle CASCADE à part entière, pouvant être, par exemple, simulé.

Chaque module du modèle constitue un élément de la structure (appelé encore ici élément de réseau, ou composant) et réalise une partie du fonctionnement global du système conformément au fractionnement induit par le découpage structurel. Ce comportement local est modélisé dans le module par des instructions, des équations, etc... et par les fonctions réalisées par d'autres modules éventuellement contenus et constituant un sous-réseau.

Un cas limite est à signaler, celui où le fonctionnement d'un module est traduit uniquement par le fonctionnement du sous-réseau, à l'exclusion de toute autre instruction. C'est-à-dire que le module ne contient que des modules.

Dans un modèle, les modules peuvent être de niveaux de modélisation différents. Il n'y a pas, a priori, de restriction dans les combinaisons, mais il faut que cette mixité ait une signification en simulation CASCADE.

EXEMPLE DE SYSTÈME MODULAIRE

Le système global COMPUTER est vu comme un module correspondant à la racine de l'arbre, c'est-à-dire au modèle total. Il contient quatre modules : MEM9 , CANAL1 , CANAL2 et CPU , et ce dernier contient à son tour ALU32 et CTL (figure II.1).

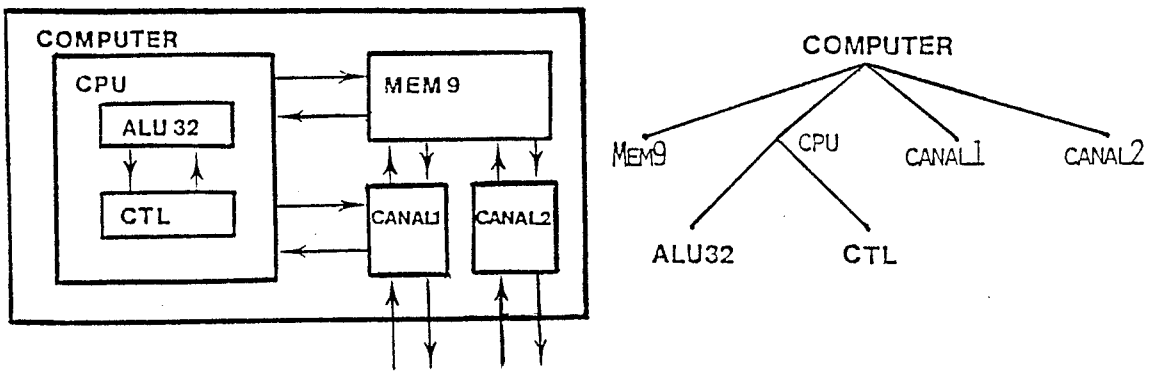


figure II.1

Les schémas ci-après illustrent le fait que tout module constitue lui-même un modèle CASCADE, CPU par exemple (qui devient le module le plus englobant de ce modèle) (figure II.2).

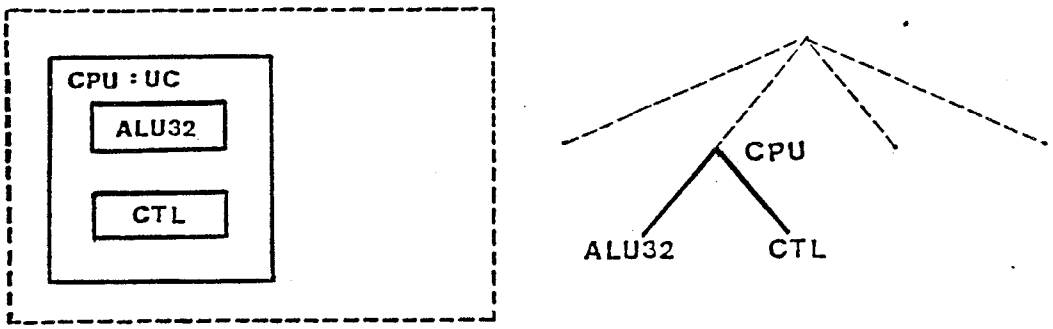


figure II.2

REMARQUE : La récursivité n'est pas permise pour la modularité car elle n'a pas de sens ici. Un module ne peut pas être contenu dans lui-même, directement ou transitivement.

Tout module intervenant dans l'arbre des imbrications est décrit par un texte (ou segment) appelé description, et le modèle total se présente comme un ensemble de telles descriptions.

II.6.3. AXIOME DE LA GRAMMAIRE CASCADE

Il dérive directement sur les deux types de données au plus haut niveau, ou "unité de traitement", que l'on peut présenter en entrée de $\phi 1$: une suite de descriptions ou un ensemble de définitions constituant un complément de langage.

Dans la réalisation actuelle, une unité de traitement constitue le contenu d'un fichier.

Le niveau de langage dans lequel sont écrits les descriptions ou les compléments de langage est spécifié dans un ordre lanref qui les précède (langage de base).

Exemple :

Unités de traitement

(1)

```
lanref CASSANDRE_S
  .
  .
  .    suite de descriptions
  .    décrites en CASSANDRE_S
  .
lanref IMAG_F
  .    suite de descriptions
  .    décrites en IMAG_F
```

(2)

```
lanref CASSANDRE-S
  .
  .
  .    définition d'un
  .    complément de lan-
  .    gage contenant des
  .    définitions décri-
  .    tes au niveau
  .    CASSANDRE_S
```

Notons qu'il est possible de mélanger dans une même unité de traitement, des descriptions écrites à des niveaux de langages différents.

II.6.4. LES COMPLÉMENTS DE LANGAGE ET L'ORDRE LANREF

II.6.4.1. DÉFINITION

La définition d'un complément de langage, appelée encore plus brièvement "définition de langage", est un segment qui permet de regrouper un ensemble de définitions de types, fonctions, procédures et descriptions, toutes décrites dans un même niveau de langage CASCADE, spécifié dans l'ordre lanref, en début de texte.

Exemple :

Définition du complément de langage B12 venant compléter CASSANDRE_S

```
lanref CASSANDRE_S
langage B12
  .
  .
  .
  . définitions
  .
  .
```

La défintion d'un type, d'une fonction, d'une procédure ou d'une description, peut être précédée, ici, du mot clé "privé". Le nom de ce segment est alors inconnu à l'extérieur de la définition de ce complément de langage. Ce mécanisme sert à cacher des définitions de segments intermédiaires servant à la construction de segments plus complexes.

II.6.4.2. UTILISATION

L'identificateur d'un complément de langage ne peut être référencé que dans un ordre lanref ; il a pour effet de compléter le langage de base avec l'ensemble des définitions constituant ce complément de langage.

Dans un texte CASCADE, tout segment utilisé doit avoir été défini ou déclaré avant d'être référencé. Le fait de nommer un complément de langage dans l'ordre lanref qui précède un texte, rend prédéfini pour tous

les segments sous la portée de ce lanref, l'ensemble des définitions contenus dans le complément de langage. D'où un allègement notable dans l'écriture des descriptions. Un même ordre lanref peut faire référence à plusieurs compléments de langage à la suite du langage de base.

Exemples :

1. Utilisations du complément de langage B12, déjà défini :

```
lanref CASSANDRE_S B12
  .
  .
  .
  . suite de descriptions
  .
  .
lanref CASSANDRE_S B12
langage B13
  .
  .
  .
  . définition du nouveau
  . complément de langage, B13
```

Notons que les compléments de langage doivent être utilisés dans le même niveau de langage que celui dans lequel ils sont définis (ici CASSANDRE_S pour B12 et B13) d'où les vérifications correspondantes à effectuer.

2. lanref CASSANDRE CLX1, CLX2

. unité de traitement

où CLX1 et CLX2 sont des compléments de langage, préalablement définis par l'utilisateur, venant complèter le niveau CASSANDRE.

Ces notions influent directement sur l'implémentation des règles de portée de noms. Elles correspondent à une extension du champ de recherche des définitions.

Dans une suite de descriptions, la portée d'un ordre lanref est l'ensemble des descriptions qui le suivent jusqu'au prochain lanref ou éventuellement jusqu'à la fin du fichier. Autrement dit, toute description est associée au dernier lanref explicite rencontré dansle fichier.

Exemple :

1. lanref CASSANDRE_S CL1 CL2
   . descriptions

2. lanref CASSANDRE_S CL2 CL3
   . descriptions

3. lanref LASSO CL4
   . descriptions

La liste CASSANDRE_S CL1 CL2 de (1) est remplacée à partir de (2) par la liste CASSANDRE_S CL2 CL3, qui est à son tour remplacée par LASSO CL4 à partir de (3).

Sur le plan traitement, ceci correspond à une gestion des accès aux langages de base et aux compléments de langage disponibles.

Exemple de complément de langage :

Le concept de complément de langage peut être utilisé pour spécialiser un projet dans le cadre d'une technologie. Une autre application importante peut être la description, par exemple au niveau LASCAR, d'un ensemble de circuits du catalogue d'un fabricant.

II.6.4.3. RÈGLES DE DÉFINITION DES COMPLÉMENTS DE LANGAGE

Un complément de langage est considéré comme un complément de l'union du langage de base et de tous les compléments de langage qui peuvent apparaître dans l'ordre lanref précédant sa définition.

Exemple :

Dans la définition de complément de langage suivante :

```
lanref IMAG Y1 Y2
langage Y3
  .
  .
  .
  .
```

Y3 est un complément de IMAG ∪ Y1 ∪ Y2.

On dit que Y3 est dérivé de IMAG ∪ Y1 ∪ Y2.

Tout complément de langage dérivé, étant défini à l'aide de segments appartenant aux compléments de langage dont il dérive, doit être utilisé précédé de ces compléments de langage. Ceci afin qu'il n'y ait pas de défaut de définition de segment.

Exemple :

Utilisation de Y3

```
lanref IMAG Y1 Y2 Y3
description MACHIN ...
  .
  .
  .
```

De plus, pour cette chaîne, par définition, il ne peut pas y avoir ambiguïté de nom, c'est-à-dire deux segments distincts désignés par un même identificateur. Cette erreur serait détectée par le Vérificateur en tant que double déclaration.

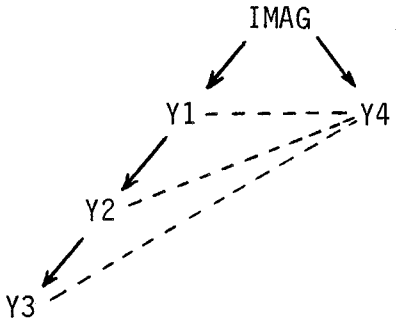
Par contre, si deux compléments de langage dérivés indépendamment apparaissent dans un même ordre lanref, il peut y avoir des ambiguïtés. Celles-ci ne sont pas détectées, pour des raisons de coût, par analyse systématique de tout le contenu de tous les compléments de langage utilisés dans l'ordre lanref, mais au cours des recherches normales de définition de segment.

Exemple :

Avec la définition de langage suivante :

```
lanref IMAG
langage Y4
  .
  .
  .
```

et compte tenu des exemples précédents, on obtient l'arborescence :



où l'on est certain qu'il n'y a pas ambiguïté de nom dans les chaînes IMAG, Y1, Y2, Y3 et IMAG, Y4, mais où l'on ne peut rien dire quant aux combinaisons de Y4 avec Y1, Y2 ou Y3 : lanref Y1 Y2 Y3 Y4, par exemple.

II.6.4.4. COMPLÉMENTS DE LANGAGE VUS DES CONCEPTEURS DU SYSTÈME CASCADE

Le mécanisme des compléments de langage est aussi utilisé par les concepteurs du système CASCADE, pour la mise en oeuvre du compilateur.

Ce dernier est constitué de l'union des mécanismes syntaxiques et sémantiques définis pour l'ensemble des niveaux de modélisation. La définition d'un niveau particulier est obtenue par :

- la restriction de cette union aux règles et à la sémantique permettant de décrire un modèle à ce niveau ;
- la définition des types, fonctions, procédures et éventuellement descriptions considérés comme primitifs pour ce niveau.

C'est à l'aide du mécanisme de complément de langage que sont faites ces définitions, en s'appuyant sur les programmes existants et sans nécessiter de nouvel effort de programmation. Ces compléments de langage, dits systèmes, sont transparents pour l'utilisateur.

L'identificateur qui suit immédiatement l'ordre lanref joue donc un double rôle :

- vis-à-vis de l'utilisateur, il spécifie le niveau de langage utilisé ;
- vis-à-vis du système, il spécifie en plus un complément de langage qui fait partie de la définition du niveau de langage standard.

D'où le schéma de structure suivant, pour un niveau de langage donné (figure II.3).

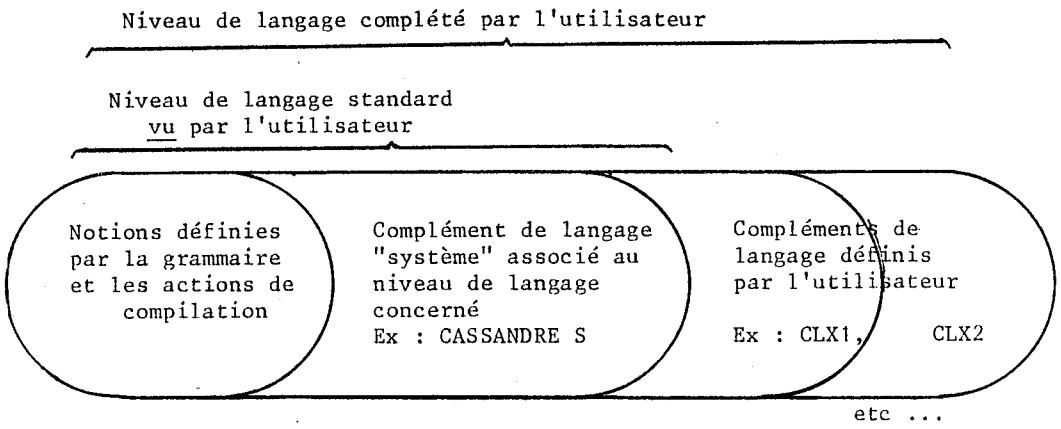


figure II.3

II.6.5. LES DESCRIPTIONS

On appelle description un texte en langage CASCADE, décrivant un module d'un système modélisé.

De manière générale, une description est composée des parties suivantes, dont seule la première n'est pas optionnelle :

- l'en-tête qui nomme la description et déclare ses attributs et son interface.
  Les attributs sont des paramètres de la description.
  L'interface est l'unique moyen de communication avec l'extérieur.

- Les assertions, qui ne font pas vraiment partie du langage de description et qui permettent à l'utilisateur de préciser des contraintes sur les attributs et les objets d'interface, ou bien des relations temporelles entre les entrées et les sorties.

- Le corps de la description proprement dit dans lequel on trouve :
  - des assertions portant sur les objets internes au module,
  - des déclarations d'externes faisant référence à des descriptions définies par ailleurs et qui seront invoquées dans ce module,
  - des définitions de fonctions, procédures, types et descriptions,
  - des déclarations d'objets internes à la description.

Ces déclarations, spécifications et définitions peuvent être écrites dans n'importe quel ordre et en nombre non limité, en respectant la règle de non référence avant.

On y trouve ensuite la partie fonctionnelle de la description. C'est un ensemble d'instructions décrivant les interrelations entre objets et modules internes ainsi que le fonctionnement du module décrit. Cette partie est spécifique à chacun des niveaux de langage CASCADE mais on y utilise tout de même des notions générales comme l'indexation, les expressions, les transferts, etc...

Une description constitue une "région" pour les identificateurs qui y sont définis. Ils ne sont connus qu'à l'intérieur de celle-ci et obéissent à la règle d'unicité.

Exemple :

Additionneur de n bits construit à l'aide de n additionneurs un bit avec propagation de la retenue (n est limité à 16).

```
description ADDER(ENT N)
     (in SIGBO A[1 : N], B[1 : N], CI; out SIGBO D[1 : N], CO)
     assertion N =< 16 ;
   corps
     externe ADD1BIT ;
     decl SIGBO X[1 : N+1] ;
     unites ADD1BIT ADD[1 : N] ;
   relations
     pour I de 1 à N repeter
       ADD[I] (A[I], B[I], X[I+1], D[I], X[I])
     fin,
       CO .= X[1], X[N+1] .= CI
fin
```

II.6.6. VALEURS ET PORTEUSES, TYPES

II.6.6.1. DÉFINITIONS

Le concept de porteuse est une généralisation de la notion de variable. En première approximation, une porteuse est un objet ayant son comportement propre dans le fonctionnement du modèle (un registre, un signal, un fil électrique,...), contenant une valeur (logique, réelle,...) et caractérisé par les opérations par lesquelles cette valeur peut être lue ou modifiée.

Les valeurs manipulées dans CASCADE peuvent être de huit types différents prédéfinis ; il s'agit des types de valeur, pour tous niveaux de langage réunis :

LOGIQUE	(logique)
BOOL	(booléen)
IMPULSION	(impulsion d'horloge)
ENT	(entier)
CAR	(chaîne de caractères)
ETAT	(nom d'état d'automate)
REEL	(réel)
NOMCONTROLE	(nom de variable de contrôle)

Les objets 'porteuses' sont déclarés à l'aide de types. Un type de porteuse est un type de porteuse primitif, paramétré par un type de valeur et une valeur par défaut. Cette dernière est associée à tout objet déclaré par ce type.

Les types de porteuses primitifs actuellement pris en compte dans CASCADE sont, pour tous niveaux de langage réunis :

REGISTRE	(registre)
SIGNAL	(signal)
COMPTEUR	(compteur)
VARIABLE	(variable)
CONTROLE	(contrôle)
FILELEC	(fil électrique)
NDELEC	(noeud électrique)
VARSYST	(variable système)
LATCH	(latch)

Sur le plan écriture du langage, et plus précisément lexicographie, les types de valeurs prédéfinis et les types de porteuses primitifs sont des identificateurs prédéfinis réservés.

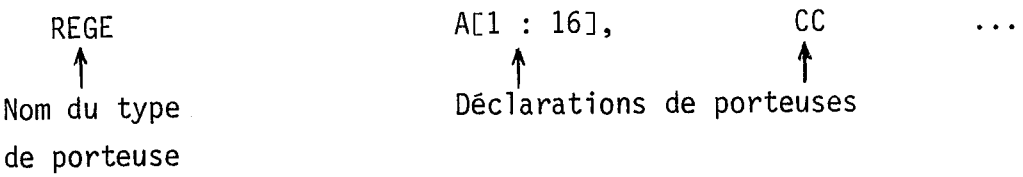
La valeur par défaut spécifiée dans un type de porteuse déclarant des objets est aussi la valeur initiale de ces objets, au début de toute séquence de simulation (initialisation des zones valeurs par le compilateur).

Exemples :

- Déclaration d'un type de porteuses

```
type REGE  =  REGISTRE (ENT,          1)
     ↑           ↑        ↑           ↑
  nom du      type de  type de     valeur
  type de     porteuse valeur      par
  porteuse    primitif (cas        défaut
                       prédéfini)
              └──────────────────────┘
                  type de porteuse
```

- Utilisation d'un type de porteuses (préalablement déclaré) comme déclarateur



On aurait pu tout aussi bien utiliser le triplet de définition complet, au lieu du nom :

REGISTRE (ENT, 1) A[1 : 16], CC ...

ce qui peut éviter de déclarer explicitement le type.

Dans la déclaration d'un type de porteuses, le nom du type peut ne pas être unique.

Exemple :

type CHAINE, TEXTE = VARIABLE (CAR, 'a'), où CHAINE et TEXTE sont deux noms différents désignant un même type. Ils peuvent être utilisés indistinctement l'un de l'autre comme déclarateur.

Lors de l'analyse de la définition d'un type de porteuse, il faut principalement vérifier que :

- les types de valeurs et les types de porteuses primitifs utilisés sont définis ;
- la valeur par défaut spécifiée appartient au type de valeur utilisé ;
- la combinaison type de porteuse primitif <-->.type de valeur est autorisée. Toutes ne le sont pas, et la matrice ci-après montre ces contraintes, d'ailleurs facilement modifiables (figure II.4).

Ces vérifications sont indépendantes du niveau de langage et effectuées systématiquement.

Types de valeurs prédéfinis / Types de porteuses primitifs	LOGIQUE	BOOLEEN	IMPULSION	ENTIER	CARACTERE	ETAT	REEL	NOMCONTROLE
REGISTRE	oui	oui	non	oui	oui	oui	non	non
SIGNAL	oui	oui	oui	oui	oui	oui	non	non
COMPTEUR	non	non	non	oui	non	non	non	non
VARIABLE	non	oui	non	oui	oui	non	oui	oui
CONTROLE	non	oui	non	non	non	non	non	non
FILELEC	non	non	non	non	non	non	oui	non
NDELEC	non	non	non	non	non	non	oui	non
VARSYT	non	non	non	non	non	non	oui	non
LATCH	oui	oui	non	oui	oui	non	non	non

figure II.4.

II.6.6.2. TYPES DE VALEURS DÉFINIS PAR L'UTILISATEUR

L'utilisateur a la possibilité de se définir, par déclaration, de nouveaux types de valeurs à partir de types déjà existants (type de base), c'est-à-dire des types prédéfinis ou des types déjà définis par l'utilisateur. Les opérations permises sur les valeurs de ce type sont celles héritées du type de base.

Les types de valeurs définis par l'utilisateur sont obligatoirement des sous-ensembles de types de valeurs prédéfinis. Dans une série de définitions successives, où un type de valeur est défini comme une restriction du précédent, le type de valeur de départ, obligatoirement prédéfini, est appelé type de valeur origine.

On distingue deux formes :

- Les types énumérés, où les différentes valeurs sont citées une à une sous forme de liste, comme en mathématiques.

  Exemple :

  type E1 = {-2, -3, 12, 5, 0, -9}

  Les éléments de l'ensemble appartiennent tous à un même type de base (règle d'homogénéité) qui est dans ce cas le type de valeurs prédéfini origine : ENTIER.

  Du point de vue traitement, signalons que c'est le premier élément qui détermine le type de base et on vérifie que les suivants appartiennent bien à ce dernier.

- Les types définis par propriété caractéristique.

  Cette forme correspond à la deuxième possibilité de définir des ensembles d'éléments en mathématiques.

  La forme est :

  type nom-de-type = {X.<Y/P(X)}

où :

- X est une variable muette, n'avant un sens et une existence qu'à l'intérieur de la définition du type ;
- .< est la notation du symbole d'appartenance mathématique d'un élément à un ensemble : $\in$;
- Y est le type de base à partir duquel est construit le nouveau type ; c'est soit un type de valeurs prédéfini, soit un type de valeur utilisateur déjà défini ;
- P(X) désigne un prédicat sur le domaine Y ; c'est la propriété caractéristique (expression booléenne) permettant de définir le nouveau type comme restriction du type de base.

Exemple :

```
type E3 = {X.<REEL/abs X =< 10}
```

E3 est l'ensemble des nombres réels dont la valeur absolue est inférieure ou égale à 10.

```
type E4 = {X.<E3/X>0}
```

E4 est l'ensemble des éléments de E3 (type de base) strictement positif. Le type de valeur origine de E4 est REEL.

Autres exemples :

```
type INDEX = {-3, -2, +2, 10}
type ZIP   = {V.<INDEX/1 =< V}
```

Le type origine de ZIP est entier et son type de base INDEX.

Un type de valeur défini par l'utilisateur peut servir à définir un type de porteuse.

Exemple :

type INDEXAD = {X. ENT/X =0 & X = 511}

type SIGAD = SIGNAL (INDEXAD, 0)

où INDEXAD est l'intervalle [0, 511] défini sur l'ensemble des entiers et où SIGAD est par exemple un type permettant de déclarer des signaux destinés à véhiculer des adresses comprises entre 0 et 511. L'envoi d'une valeur extérieure à ces bornes dans tout signal déclaré sous la portée du type SIGAD provoque la détection d'une erreur (cas habituellement dynamique).

Un type de valeurs, prédéfinis ou utilisateur, sert aussi à déclarer des constantes ou des attributs.

II.6.6.3. CORRESPONDANCE

TYPES DE VALEURS <--> CONSTANTES LEXICOGRAPHIQUES

Elle est donnée par le tableau ci-après (figure II.5)

Type de valeur	Constantes lexicographiques
LOGIQUE et BOOLEEN	Constante logique
IMPULSION	Constante entière décimale : 0 ou 1 pour "Bas" et "Haut"
ENTIER	. avec base explicite : binaires, octales, hexadécimales . sans base explicite : décimales
CHAINE DE CARACTERES	Constantes littérales
ETAT	Certains identificateurs, reconnus par syntaxe seulement
REEL	Constantes réelles . simples . avec exposant ou facteur d'échelle . avec exposant ou facteur d'échelle, sans partie décimale

Figure II.5

II.6.7. LES FONCTIONS ET PROCÉDURES

Les fonctions et procédures en CASCADE sont deux catégories de segments utilisés pour décrire le comportement d'une partie du modèle, de manière plus ou moins abstraite selon le niveau de langage utilisé. Par défaut, fonctions et procédures sont écrites en langage CASCADE mais elle peuvent l'être aussi dans un langage de programmation classique, FORTRAN ou PASCAL.

La définition de ces segments n'est que toute récente dans CASCADE. Nous en donnons cependant quelques brèves explications.

II.6.7.1. LES FONCTIONS

Une fonction est un segment qui calcule et renvoie un résultat, sans effet de bord sur son environnement d'appel. Les paramètres peuvent être des porteuses ou des valeurs. Le résultat provient de l'évaluation d'une expression, après exécution du corps de la fonction.

La forme générale d'une définition de fonction est la suivante (où les crochets entourent ce qui est optionnel, et les étoiles indiquent des listes de déclarations ou définitions en nombre arbitraire, dans un ordre quelconque).

```
fonction identificateur [dimensions]
   (paramètres formels) résultat type
[assertions liste d'assertions]
[corps
  *[définitions de types, procédures, fonctions]
  *[accès liste d'accès]
  *[déclare déclarations]
   [relations liste d'instructions]]
    retour expression
finfonction
```

II.6.7.2. LES PROCÉDURES

Une procédure est un segment qui ne renvoie aucun résultat, mais peut modifier le contenu d'une ou plusieurs porteuses de son environnement d'appel.

Les paramètres peuvent être des porteuses et des valeurs. Le corps de la procédure exprime les transformations que l'appel de la procédure effectue sur son environnement.

La forme générale d'une définition de procédure est la suivante :

```
procédure identificateur (paramètres formels)
   [assertions liste d'assertions]
   corps
   [définitions de types, fonctions, procédures]
   [accès liste d'accès]
   [déclare déclarations]
   [relations liste d'instructions]
finprocédure
```

II.6.7.3. PARAMÈTRES FORMELS, PARAMÈTRES EFFECTIFS

Les paramètres formels d'un segment de définition de fonction ou de procédure sont accessibles à l'intérieur du segment suivant deux modes :

- accès en lecture seule : c'est le mode d'accès par défaut, le seul autorisé pour les fonctions ;
- accès en lecture/écriture : pour une procédure seulement ; le paramètre est alors spécifié comme modifiable, par un mot clé déterminé, dans la définition de la procédure.

II.6.7.4. ACCÈS À DES OBJETS GLOBAUX, MÉCANISME D'IMPORTATION

Par défaut, dans le corps d'une fonction ou d'une procédure, on ne peut accéder qu'aux paramètres et aux objets déclarés localement, c'est-à-dire dans le corps de la fonction ou de la procédure. Tout accès à un objet de l'environnement d'appel doit donc être normalement effectué via le mécanisme de passage de paramètres. Pourtant, il est parfois intéressant de pouvoir accéder systématiquement à des objets de l'environnement d'appel, sans passer par ce mécanisme. Ceci est possible grâce à un ordre d'importation statique d'objets globaux, appelé "accès". Là encore, l'accès à ces objets est défini par défaut en lecture seulement, mais peut être aussi spécifié en lecture/écriture.

Toutes ces notions influent sur les règles de portée de noms et imbrications de segments et les vérifications qui en découlent. Elles font aussi largement appel aux vérifications de compatibilité de types, de compatibilité dimensionnelle, etc...

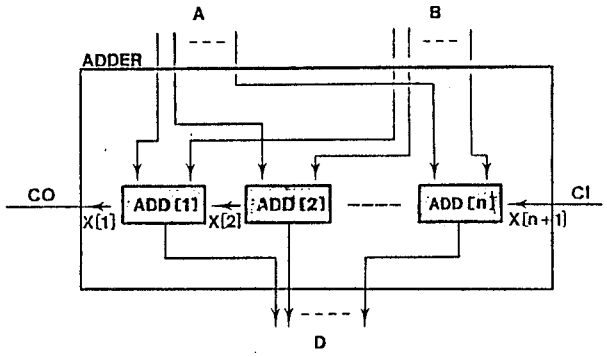
Il serait beaucoup trop long d'exposer ici, en détail et rigoureusement, ces règles de portées de noms. Se rapporter pour cela au manuel de référence CASCADE (Bor.84).

II.6.8. LES ATTRIBUTS

Lorsque des attributs sont associés à une description, cette description (texte unique) définit non pas un modèle mais une famille de modèles, de structure et de comportement analogues mais différent entre eux :

- soit par leurs dimensions ;
- soit par leurs durées ;
- soit par toute combinaison de ces deux facteurs.

L'additionneur ADDER, déjà montré au paragraphe II.6.5 est un exemple de paramétrisation dimensionnelle. C'est un additionneur n bits construit à l'aide de n additionneurs un bit, avec propagation de la retenue (figure II.6).




```
description ADDER(ENT N)
     (in SIGBO A[1 : N], B[1 : N], CI; out SIGBO D[1 : N], CO)
     assertion N =< 16 ;
   corps
     externe ADD1BIT ;
     decl SIGBO X[1 : N+1] ;
     unites ADD1BIT ADD[1 : N] ;
   relations
     pour I de 1 à N repeter
       ADD[I] (A[I], B[I], X[I+1], D[I], X[I])
     fin,
       CO .= X[1], X[N+1] .= CI
fin
```

figure II.6

L'entier n est déclaré comme attribut de la description ADDER. n sert à paramétrer les dimensions des entrées A et B, de la sortie D, du signal interne de propagation X, et le nombre d'exemplaires de l'additionneur un bit utilisé.

Les attributs sont des objets déclarés sous la portée d'un type de valeur, et sont référencés par leur nom pour toute utilisation. Pour la simulation, ce sont des constantes, mais celles-ci ne sont pas initialisées dans la description. Leur valeur est affectée par l'utilisateur au moment de la construction du modèle, dans la deuxième phase de compilation. Seuls les attributs du module le plus englobant sont alors à préciser. Les attributs des modules des niveaux inférieurs reçoivent leur valeur de proche en proche, par propagation descendante.

Le type de valeur déclarateur peut être prédéfini ou défini par l'utilisateur. Dans la version actuelle, les attributs sont des scalaires.

Exemple de déclaration d'attributs :

```
REEL J3, C, KZ ;
```

Dans une version ultérieure du Vérificateur, les attributs pourront être dimensionnés (voir plus loin les objets dimensionnés en CASCADE). Il pourront prendre une valeur par défaut dans le cas d'absence de valuation par propagation ou par l'utilisateur. Enfin, certains, n'influant pas sur la structure du modèle, pourront n'être valués qu'à la simulation. Suivant le traitement auquel ils seront soumis, nous les considèrerons comme des porteuses "statiques", avec une zone valeur, ou comme des constantes. Nous les appellerons "attributs fonctionnels".

II.6.9. LES INTERFACES

Dans un système modélisé en CASCADE, tout module décrit par un segment description est vu de l'extérieur comme une boîte noire avec laquelle on ne peut communiquer que par l'intermédiaire de son interface. Plus précisément, cela veut dire que de l'extérieur de ce module, seuls les éléments de son interface sont connus et accessibles. Ils correspondent, par exemple, aux éléments des broches d'entrées/sorties.

> Notons que dans le cas général, le module et son environnement sont modélisés à des niveaux différents, et la communication éléments d'interface-extérieur est établie via des fonctions de conversion.

L'interface est déclaré au début de la description, suivant la syntaxe :

description NOM-DESCRIPTION (déclaration d'interface)

et il est composé d'objets appelés éléments formels d'interface, éventuellement dimensionnés et déclarés avec leur type. Ces objets sont obligatoirement des porteuses, et à chacun d'eux est associé un sens, spécifié par les mots clés suivants :

- in pour les éléments "entrée" : leur valeur est modifiable de l'extérieur de la description, mais pas de l'intérieur ;
- out pour les éléments "sortie" : leur valeur est modifiable de l'intérieur de la description, mais pas de l'extérieur ;

- inout pour les éléments dont la direction peut varier selon les instants, en fonction de certaines conditions ; leur valeur est donc modifiable, suivant le moment, de l'intérieur ou de l'extérieur de la description ;

- nd pour les éléments non orientés ; on trouve ce type de liaison au niveau électrique par exemple.

Exemple de déclaration d'interface :

```
description MULT (in HORLOGE H1; SIGBO EM[1 : 16], START ;
                  out SIGBO OCCUPE, SM[1 : 32])
```

Les sens autorisés dans la définition d'un interface dépendent du niveau de langage auquel on se réfère et les types sous lesquels les éléments formels sont déclarés doivent être eux aussi reconnus à ce niveau-là, pour ce contexte.

II.6.10. .LES TABLEAUX

Dans CASCADE, la plupart des objets peuvent être dimensionnés : porteuses, constantes, attributs, fonctions, modules composant le système modélisé, ...

II.6.10.1. LES DÉCLARATIONS DE DIMENSIONS

Elles sont faites avec les conventions et les restrictions suivantes :

- le nombre de dimensions est quelconque ;
- une dimension est définie par sa borne gauche et sa borne droite, séparées par le caractère ":" ;
- les bornes sont des entiers positifs ou nuls;
- la borne gauche n'est pas forcément inférieure à la borne droite ;
- les dimensions sont écrites entre crochets, et séparées par des points virgules.

Exemples :

REGISTRE A[0:3; 1:4; 12:1] ;

Une borne peut être une expression, fonction d'attributs.

II.6.10.2. L'INDEXATION DES TABLEAUX

Tout objet déclaré avec des dimensions peut être indexé. L'indexation la plus générale est faite avec les conventions suivantes :

- On peut extraire d'un tableau un élément, plusieurs éléments non nécessairement consécutifs, plusieurs tranches non nécessairement contiguës.
- Avec une même indexation, on peut extraire d'un tableau plusieurs fois le même élément.
- Toutes les dimensions déclarées d'un tableau doivent être matérialisées lors d'une indexation qui ne référence pas le tableau tout entier.
- Si plusieurs éléments d'une même dimension sont référencés, leurs rangs dans cette dimension sont séparés par des virgules. Si des éléments sont consécutifs, on peut abréger cette écriture en celle d'un intervalle.
- Si tous les éléments d'une dimension sont référencés dans l'ordre de leur déclaration, il n'est pas nécessaire d'écrire l'intervalle ; mais cette dimension doit être matérialisée par le point virgule correspondant (sauf si c'est la dernière). De plus, on appelle dimension dégénérée, une dimension qui est réduite à un seul élément. On ne tient pas compte d'une telle dimension pour ce qui est de la notion de compatibilité dimensionnelle et de l'action des opérateurs.

Exemples :

Avec les déclarations :

```
A[0:3 ; 1:4 ; 12:1]
B[1:4 ; 2:2]
```

on peut écrire les indexations suivantes :

A[0;2;10] , B[2;2] , B[3;]	Extraction d'un élément
A[1,3;2;10,8,6]	Extraction de six éléments isolés
A[;2;5] , B[2:4;]	Extraction d'un vecteur
A[2,3;2,4;12:5]	Extraction de plusieurs vecteurs
A[3:0;1;10]	Extraction d'un vecteur retourné sur lui-même
A[;;10:1,12,11]	Décalage circulaire de deux éléments sur la troisième dimension
A[0,1,1,1,2;;11]	Extraction plusieurs fois du même vecteur

Nous verrons plus loin, dans le paragraphe sur les expressions, qu'un index peut être une expression, statique ou dynamique.

Les vérifications effectuées lors de la rencontre d'une indexation sur les suivantes :

- l'indexation a même nombre de dimensions que la déclaration de l'objet,
- les index utilisés sont toujours compris entre les bornes de la dimension correspondante, dans la déclaration.

Ces traitements sont effectués à l'aide de descripteurs manipulés en général sur la pile de vérification.

Toute indexation rencontrée donne lieu ensuite à la construction d'un descripteur normalisé (nombre et étendues des dimensions), servant aux vérifications ultérieures de compatibilité dimensionnelle.

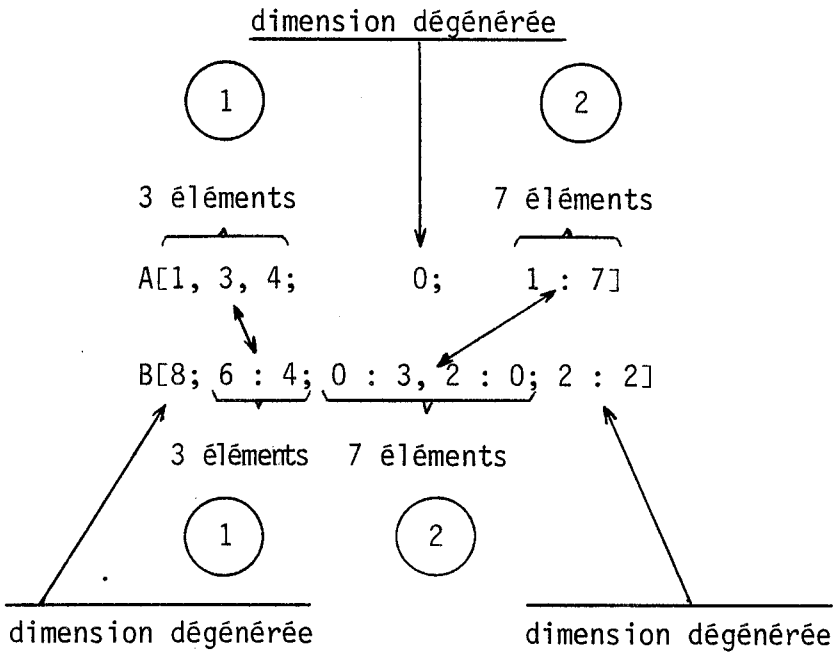
Le cas des vérifications attachées aux index dynamiques sera aussi vu dans le paragraphe sur les expressions.

II.6.10.3. DÉFINITION DE LA COMPATIBILITÉ DIMENSIONNELLE

- Nous appelons tableau, d'une manière générale, tout objet ou expression dimensionné, indexé ou pas (voir grammaire).
- Une dimension est dite dégénérée si elle ne contient qu'un élément.
- Nous dirons que deux tableaux sont dimensionnellement compatibles s'ils ont même nombre de dimensions non dégénérées et si, compte tenu des dimensions dégénérées, les dimensions correspondantes ont même nombre d'éléments.

Exemple :

Les indexations : A[1, 3, 4; 0; 1 : 7]
et : B[8; 6 : 4; 0 : 3, 2 : 0; 2 : 2]
sont compatibles.



- De plus, il y a évidemment compatibilité dimensionnelle entre deux scalaires et entre un scalaire et un élément de tableau (c'est-à-dire une indexation de tableau qui ne comporte que des dimensions dégénérées).

II.6.10.4. LES TABLEAUX DE CONSTANTES IMMÉDIATES, CAS DES CONSTANTES LOGIQUES

Dans le prototype actuel, sruls les vecteurs de constantes sont définis, avec la syntaxe suivante :

[cste 1, cste 2, ... , ...]

où les constantes peuvent être entières, réelles, logiques ou littérales.

Nous prévoyons d'implanter plus tard une forme générale à n dimensions, où les vecteurs de la première dimension sont séparés par le caractère ":" , entre la deuxième et la troisième, le séparateur est :: (on double le "deux points"), etc...

Exemples :

- d'un vecteur de réels :

  [3.14, 0.01E-3, 1.0]

- d'un tableau d'entiers à trois dimensions : 3 x 4 x 2 :

  [12,11,10:9,8,7:6,5,4:3,2,1::0,-1,-2:-3,-4,-5:-6,-7,-8:-9,-10,-11]

  3 — 4 — 2

CAS DES CONSTANTES LOGIQUES

Dans un souci d'allègement d'écriture, la virgule, séparateur d'éléments, est omise ici ; ceci est conforme à la définition de vecteurs logiques comme unités lexicales. De même, dans le cas de vecteurs logiques, l'accent grave, caractéristique dans l'écriture de ce type de valeur, peut être réduit au premier, et les crochets de parenthésage sont omis.

Exemples :

• de vecteurs de constantes logiques :

`a10 (identique à `a`1`0, ou à `a`10 ou à `a1`0)

• de tableau à deux dimensions : 8 x 2 :

[`10101010:`11110000]

Dans tous les cas, la rencontre d'un tableau de constantes immédiates (donc non déclarées) donne lieu à la construction d'un descripteur dimensionnel normalisé, en vue des vérifications de compatibilité dimensionnelle à venir.

II.6.11. LES DESCRIPTIONS EXTERNES

Lorsque des descriptions ne sont définies ni internes, ni dans un complément de langage, mais simplement au premier niveau dans l'unité de traitement "suite-de-descriptions" donnée en entrée du compilateur, elles sont dites externes et spécifiées comme telles par l'ordre correspondant.

Exemple :

```
externes HALFADD, OU ;
```

Dans une version ultérieure, les fonctions et procédures, pour le moment définies internes ou dans un complément de langage, pourront aussi être définies externes et apparaître au premier niveau dans la "suite-de-descriptions".

II.6.12. LES DÉCLARATIONS INTERNES

Ce sont les déclarations des éléments internes à un module du système modélisé. On y trouve :

- Sous la portée du mot clé "unité" ("use" en anglais) les déclarations, à partir de leur description, des autres modules contenus. Nous les appellerons aussi, indistinctement, unités ou exemplaires de description.
- Sous la portée du mot clé "declare", les porteuses ou constantes déclarées par leur type.

II.6.12.1. DÉCLARATIONS DES PORTEUSES

La forme générale est :

type-déclarateur liste-de-porteuses ;

où :

- type-déclarateur est un nom de type de porteuses préalablement déclaré, ou est explicitement le triplet de définition du type ;
- liste-de-porteuses est l'énumération des porteuses déclarées, c'est-à-dire leurs noms éventuellement suivis de leurs dimensions.

Exemple :

```
REGBO A[0:32], B,[C 1:4];
```

si REGBO est un type de porteuses préalablement déclaré par, par exemple : type REGBO = REGISTRE (BOOL, 0). On aurait pu tout aussi bien écrire : REGISTRE (BOOL, 0) A[0:32], B, C[1:4] ;

CAS DES PORTEUSES SYNONYMES

Il est possible, dans une déclaration de porteuse, de déclarer celle-ci synonyme :

- d'une autre porteuse déjà déclarée ;
- d'une partie seulement de celle-ci ;
- de la combinaison structurelle de plusieurs autres porteuses, déjà déclarées.

Sémantiquement, cela veut dire que tous ces objets ont bien des descripteurs internes distincts mais des zones valeurs, ou des parties de zones valeurs, communes.

Exemple :

Dans RR1 A[0:15], ..., B[1:16] = A,
C[0:7] = B[5,12] ;

la porteuse B est déclarée synonyme de A toute entière, tandis que C[0:7] est déclarée sur les éléments 5 ... 12 seulement de B.

REMARQUE :

Dans une relation de synonymie, les porteuses mises en jeu doivent être de même type.

A cette restriction près, l'expression de porteuses située en partie droite de la relation peut être générale du point de vue du Vérificateur. Pourtant, sa complexité est d'autant plus limitée que l'on rencontre de difficultés dans les traitements de compilation $\phi 2$ assurant l'implantation et l'exploitation des zones valeurs. En effet, une porteuse déclarée synonyme de la combinaison de plusieurs autres par exemple, a très souvent sa zone valeur constituée d'autres zones valeurs, ou parties de celles-ci, dispersées en mémoire, donc non consécutives. La gestion des liens présente alors une lourdeur rédhibitoire dans les mécanismes de compilation et de simulation du modèle.

Exemple :

Dans RR2 A ...

RR2 B, C, ..., D[1:3] = A\B\C ;

la porteuse D[1:3] est définie comme étant la concaténation, dans cet ordre, des trois objets scalaires A,B,C.

Le problème qui vient d'être évoqué peut déjà se présenter dans le cas où la partie droite de la relation est un objet seul, indexé. S'il est nécessaire que les éléments sélectionnés aient leurs zones valeurs consécutives en mémoire, compte tenu des conventions sur les sens de rangement des tableaux, cette indexation doit vérifier certaines contraintes que l'on trouvera exposées plus loin au paragraphe 6.13.6.

II.6.12.2. DÉCLARATION DES CONSTANTES

Le déclarateur est ici un nom de type de valeurs, prédéfini ou préalablement défini par l'utilisateur. La déclaration d'une constante, nom et dimensions éventuelles, doit aussi comporter sa valeur explicite.

Exemples :

```
ENTIER      C = 12 ;
CARACTERE   A[1:2] = [`TOTO`, `D2`] ;
INDEX       K = 3 ;
```

La sémantique de cette notion est la suivante : utiliser, dans une description, une constante déclarée, revient exactement à utiliser la valeur qu'elle représente (facilité en cas de modification de cette valeur).

Les mécanismes que l'on trouve derrière cette notion ne sont pas les mêmes que dans le cas des porteuses. S'il y a une expression en partie droite, celle-ci est évaluée statiquement : ce qui implique qu'elle ne peut être constituée que d'autres constantes préalablement déclarées, d'attributs ou de constantes immédiates combinées par les opérateurs permis pour les types de valeurs auxquels elles appartiennent. A la simulation, c'est la valeur de la constante, et non l'expression, qui est manipulée directement. La synonymie n'est donc pas vue ici de la même façon.

II.6.13. DESCRIPTION DES RÉSEAUX : LES EXEMPLAIRES DE DESCRIPTIONS ET L'ORDRE "UNITÉ" (USE)

II.6.13.1. LES EXEMPLAIRES DE DESCRIPTIONS DANS L'ORDRE UNITÉ

Nous avons déjà vu que tout module intervenant dans l'arbre des imbrications est décrit par un texte appelé description. Plus précisément, dans un modèle CASCADE, chaque module est un exemplaire de sa description ; utiliser un module , c'est utiliser un tel exemplaire.

Une description peut ainsi créer plusieurs exemplaires car plusieurs modules, fonctionnellement et structurellement identiques mais distincts, peuvent être décrits par une seule description.

C'est dans l'ordre "unités" que sont spécifiées ces déclarations.

Exemple :

Dans

```
        .
        .
        .
    externes NAND2, NAND3 ;
    unites NAND2 N1, N2, N4, N5, N6 ;
           NAND3 N3 ;
        .
        .
        .
```

N1, N2, N4, N5 et N6 sont les déclarations de cinq exemplaires de la description NAND2, et N3 est la déclaration d'un exemplaire de la description NAND3.

Il est à remarquer qu'afin de pouvoir effectuer un certain nombre de vérifications, exposées plus loin dans ce paragraphe, l'interface des descriptions de référence nommées dans l'ordre unité, doit avoir été déjà compilé par le Vérificateur lors du traitement de cet ordre.

II.6.13.2. PRINCIPE DE DESCRIPTION DES RÉSEAUX

Dans le langage CASCADE, une structure arborescente s'exprime de la manière suivante :

- Le modèle global, défini comme un module, racine de l'arbre, est décrit par une description.
- Pour les autres niveaux, chaque module est déclaré par son nom, comme exemplaire de sa description, dans la description du module qui l'englobe. Ces deux descriptions peuvent être décrites indépendamment l'une de l'autre, au problème d'interface près, déjà évoqué.

Les descriptions utilisées pour déclarer des exemplaires sont spécifiées externes ou définies internes dans les descriptions qui les référencent. Elles peuvent aussi être connues implicitement au niveau de langage CASCADE concerné, ou définies par l'utilisateur au moyen des compléments de langage.

Exemple :

```
description COMPUTER ...
    :
  externes UC, MEMORY, CHANNEL ;
  unites UC CPU ;
         MEMORY MEM9 ;
         CHANNEL CANAL1, CANAL2 ;
    :
fin
```

```
description UC ...
    :
description MEMORY ...
    :
description CHANNEL ...
    :
```

Ici, le modèle COMPUTER est défini par la description de même nom. Cette dernière, par l'ordre "externes", fait référence aux descriptions UC, MEMORY et CHANNEL. Dans l'arbre, cela correspond au niveau directement inférieur à COMPUTER. Par l'ordre "unités", CPU est déclaré comme exemplaire de UC, MEM9 comme exemplaire de MEMORY, et CANAL1 et CANAL2 comme exemplaires de CHANNEL.

Même principe pour UC, dont l'exemplaire CPU contient les modules ALU32 et CTL, exemplaires respectifs des descriptions ALU et CONTROLE.

```
description UC ...
    :
externes ALU CONTROLE ;
unites ALU ALU32 ;
       CONTROLE CTL ;
    :
fin
```

```
description CONTROLE ...
    :
description ALU ...
    :
```

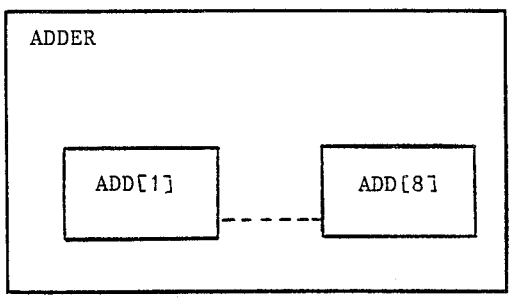
L'ensemble des descriptions COMPUTER, UC, MEMORY, CHANNEL, CONTROLE et ALU constitue la forme textuelle d'entrée du modèle.

II.6.13.3. FACILITÉ D'ÉCRITURE : STRUCTURATION EN TABLEAUX

Les exemplaires d'une description peuvent être structurés en tableaux. Il est par exemple possible de déclarer des vecteurs ou des matrices de modules, comme pour tout autre objet dimensionné.

Exemple :

Soit un additionneur 8 bits, ADDER, constitué de 8 additionneurs 1 bit. Nous pouvons déclarer ces 8 modules, exemplaires de la description ADD1BIT, sous forme d'un vecteur ADD[1:8]. Les unités ainsi générées peuvent être référencées par indexation, comme pour tout objet dimensionné (figure II.7), avec toutefois quelques contraintes vues ultérieurement.




```
description ADDER ...
   :
  externe ADD1BIT ;
  unités ADD1BIT ADD[1:8] ;
   :
fin
description ADD1BIT ...
   .
   .
   .
```

figure II.7

II.6.13.4. RÉFÉRENCE AUX PORTEUSES D'INTERFACE D'UN MODULE ENGLOBÉ

L'ordre unité déclarant des modules dans une description déclare aussi, implicitement, leur interface. Les éléments de ces interfaces sont donc accessibles et référençables depuis cette description. Compte tenu des définitionsdes segments dans CASCADE et des règles de portée des noms qui leur sont associées, des modules distincts peuvent avoir des éléments d'interface de même nom, et il y a alors ambiguïté Pour lever cet inconvénient, la référence à l'élément, constituée du nom formel de celui-ci (déclaré dans la description du module qui le contient), éventuellement indexé, est préfixée par la référence à ce module, c'est-à-dire son nom, éventuellement indexé.

Cette forme d'écriture est appelée "référence par notation pointée".

Soit BDM[15:0] une broche de connexion d'un banc de mémoire MEM[3] pour un bus bi-directionnel de données de dimension 16. Les 12 bits de poids fort de cet élément d'interface, par exemple, numérotés de 15 à 4 sont référencés dans la description MEMOIRE par MEM[3].BDM[15:4] (figure II.8).

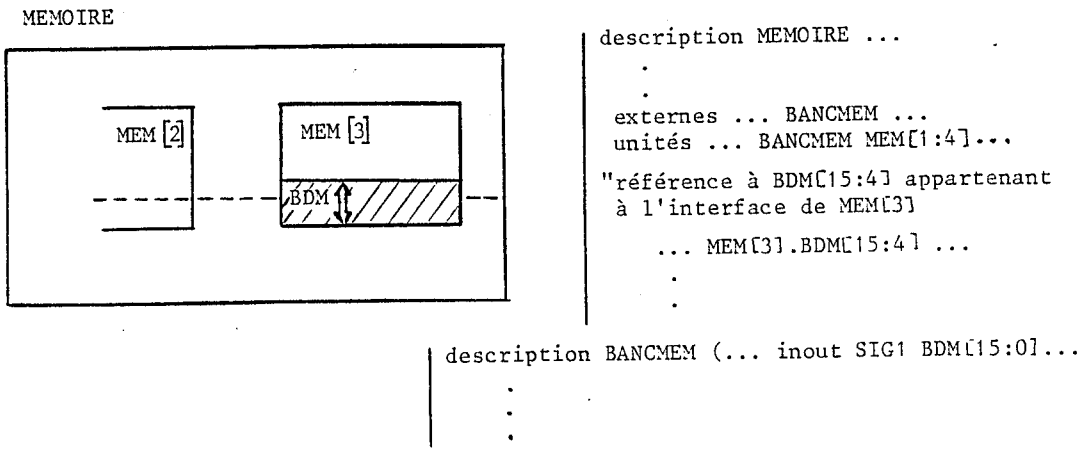


figure II.8

II.6.13.5. LES COMMUNICATIONS ENTRE UNITÉS

Dans le cas général (modèles hybrides), les modules sont décrits à des niveaux de modélisation différents et leur connexion s'effectue via des fonctions de liaison. Plus précisément, lorsque les éléments d'interface et les objets qui leur sont connectés ne sont pas définis au même niveau de langage, les valeurs qu'ils transportent sont le plus souvent de types différents, et une conversion adéquate de ces valeurs doit alors intervenir. C'est le rôle des fonctions de liaison.

Dans le cas particulier où les modules qui communiquent sont décrits au même niveau de modélisation, la sémantique de la liaison est celle de la connexion directe (définie pour ce niveau) entre les objets de types auxquels appartiennent les éléments d'interface. Les types des valeurs mises en jeu doivent alors être toujours compatibles, conformément aux propriétés de la connexion. Aucune fonction de conversion n'est utilisée ici.

L'interconnexion des modules constituant l'arborescence se fait par l'intermédiaire des interfaces.

Exemple :

Le module X est raccordé par son interface au module M qui l'englobe directement au moyen d'objets α , internes à ce dernier, et aux modules T et U au moyen de leurs éléments d'interface respectifs β et γ , accessibles à partir de M (figure II.9).

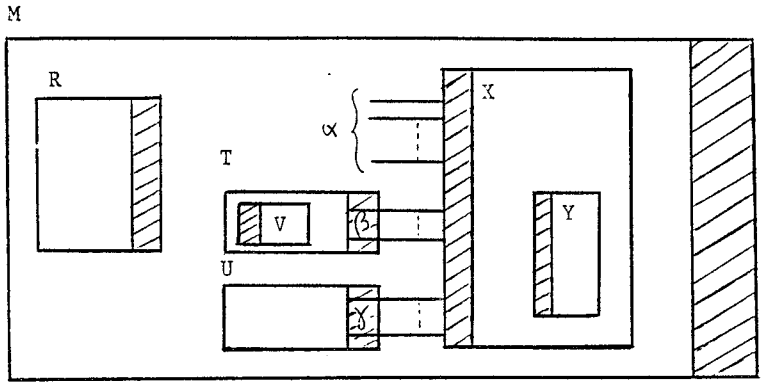


figure II.9

Le module X communique alors directement avec les modules T et U, sans passer par l'intermédiaire d'objets locaux au module M.

La notion de connexion d'interface traduit bien ainsi l'utilisation qui est faite de chacun des modules.

Enfin, le module le plus englobant communique comme les autres, par son interface, avec son environnement qui est, dans ce cas, le monde extérieur. Il peut à son tour être connecté ultérieurement dans un autre module (modularité).

REMARQUE : UTILISATION PARTIELLE D'UN MODULE :

Des éléments de l'interface d'un module peuvent ne pas être connectés ("pattes en l'air"). Ceci correspond à une utilisation partielle du module comme, par exemple la non-utilisation de certaines fonctions d'une Unité Arithmétique et Logique.

Du point de vue langage de description, l'interconnexion des modules peut être spécifiée de deux endroits, dans un segment description :

- soit dans l'ordre unité ;
- soit dans la partie fonctionnement (voir plus loin, la partie relations).

En fait, pour un même module, des éléments d'interface peuvent être connectés de la première façon, et d'autres de la deuxième. Il y a effet cumulatif.

D'une manière générale, dans toute interconnexion de modules, la compatibilité en nombre et en dimension doit être assurée entre les éléments d'interface de l'unité raccordée et les objets auxquels ils sont liés.

II.6.13.6. LA CONNEXION DANS L'ORDRE "UNITÉ"

La déclaration d'un module peut être suivie de la connexion globale de son interface. Le dimensionnement est alors interdit. Ces formes sont en effet exclusives car on ne peut connecter qu'un seul interface à la fois, pour un même nom de module.

On aura donc :

nom-de-module dimensions

ou nom-de-module connexion-d'interface.

Nous laissons volontairement de côté la connexion d'interfaces avec fonctions de conversion explicites, dans le cas de modèles multi-niveaux, cette forme devant être implantée ultérieurement. Nous nous plaçons donc dansle cas de modèles décrits à un seul niveau de modélisation, ou à plusieurs mais alors avec fonctions de conversion implicites, directement programmées dans le simulateur.

La modélisation de la connexion présuppose l'existence d'au moins deux objets : un élément d'interface, connecté à un objet accessible du module englobant. C'est exactement la notion de connexion d'un module dans la partie fonctionnelle d'une description. Dans l'ordre unité, c'est une autre modélisation qui est en vigueur. On considère que les deux objets sont synonymes, c'est-à-dire qu'ils ont une zone valeur commune.

Cette vision n'apporte rien au comportement externe du modèle, mais s'avère intéressante, pour les structures complexes, sur le plan de la simulation, car on économise alors les transferts d'informations entre les deux objets. Ceci n'est bien sûr valable que pour les modèles mono-niveau.

Exemple d'écriture :

Le schéma ci-dessous illustre une bascule D réalisée par l'interconnexion de six portes NAND (figure II.10).

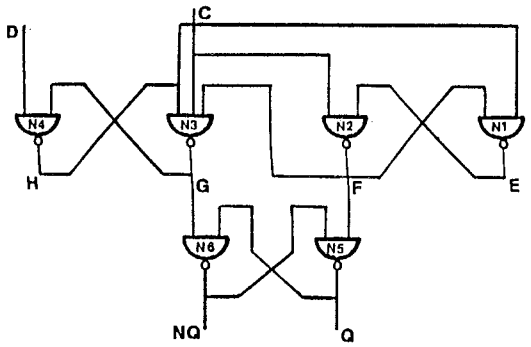


figure II.10

La description correspondante est la suivante :

```
description DFF(in SIGB1 C, D; out SIGB1 Q ; out SIGBO NQ)
   corps
      externe NAND2, NAND3;
      declarations
         SIGBO E, F, G, H;
         unites NAND2 N1(H, F, E),
                      N2(E, C, F),
                      N4(G, D, H),
                      N5(F, NQ, Q),
                      N6(Q, G, NQ);
                NAND3 N3(F, C, H, G);
   fin
```

Les objets connectés à l'interface d'un module, dans l'ordre unité, peuvent être :

- des porteuses ;
- des constantes générales .

La connexion d'une porteuse à un élément d'interface correspond au cas le plus fréquent. La porteuse est locale au module englobant ou appartient à l'interface d'un module englobé, de même niveau que le module connecté (porteuse référencée en notation pointée).

Dans le cas où l'objet connecté est une constante générale, ce peut être une constante déclarée, un attribut ou une constante explicite numérique, logique ou littérale.

Tous ces objets, porteuses ou constantes, peuvent être des tableaux. Ce sont donc soit des indexations de porteuses, attributs ou constantes déclarées, dimensionnés soit des tableaux de constantes immédiates.

Si l'objet connecté à l'interface est indexé, rappelons que pour des raisons de difficultés de compilation de la synonymie, cette indexation ne peut alors sélectionner que des éléments à zones valeurs contiguës en mémoire. En plus des contraintes de compatibilité dimensionnelle classiques, l'indexation doit donc, compte tenu de l'ordre de rangement des éléments en mémoire, respecter la règle suivante : sur p dimensions, les n premières ($0 \leq n \leq p$) doivent être prises en totalité. La dimension $n + 1$ doit être un intervalle ou être dégénérée. Toute les suivantes doivent être dégénérées.

Les autres contrôles à effectuer dans la connexion sont relatifs à la cohérence sur les sens et à la compatibilité des types des objets utilisés.

VÉRIFICATION DE LA COHÉRENCE DES SENS

Une porteuse peut être connectée à un élément d'interface de n'importe quel sens, autorisé dans le niveau de langage considéré.

Une constante, par contre, ne peut être connectée qu'à une entrée.

VÉRIFICATION DE LA COMPATIBILITÉ DES TYPES

TYPES DE PORTEUSES :

Si l'élément d'interface est connecté à une porteuse, ils doivent avoir tous deux même types de porteuses primitif. Il y a pourtant deux exceptions à cette règle, dans le niveau électrique IMAG_Fin où l'on peut trouver un élément d'interface "fil électrique" connecté à un "noeud électrique", et dans les niveaux Transfert de Registres où un élément d'interface Signal peut être connecté à la sortie d'un Registre.

TYPES DE VALEURS :

Si l'élément d'interface est connecté à une porteuse ils doivent avoir tous deux même type de valeur. Dans le cas d'une constante connectée à une entrée, la constante doit appartenir au type de valeur de l'entrée.

VALEUR PAR DEFAUT :

Dans tous les cas, elle n'a pas d'importance.

II.6.13.7. LA PROPAGATION DES VALEURS D'ATTRIBUTS PAR L'ORDRE UNITÉ

Une description paramétrable par attributs doit avoir la valeur de ces derniers fixée dans l'ordre unité la référençant, avant de générer les exemplaires de modules nécessaires.

D'où la forme, maintenant complète, de l'ordre unité :

unité nom-de-description (liste-de-valeurs-d'attributs) liste-d'exemplaires

La même description peut, bien sûr, être paramétrée plusieurs fois, avec des listes de valeurs d'attributs différentes, dans le même ordre unité.

Une valeur d'attribut peut être une expression, à condition que celle-ci soit évaluable statiquement, avant la simulation. Elle ne peut donc être constituée que de constantes ou d'autres attributs.

Sur le plan des vérifications à effectuer, signalons que la cohérence dimensionnelle doit être respectée :

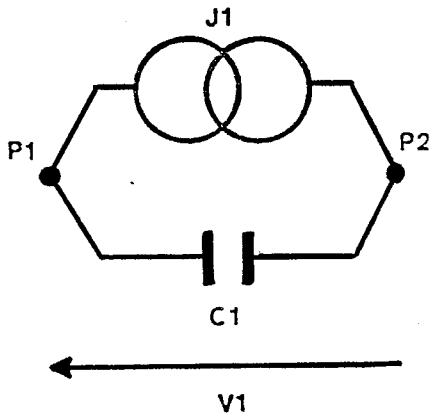
- Même nombre d'éléments dans la liste des valeurs que dans la déclaration de la liste des attributs formels dans l'en-tête de la description correspondante.
- Compatibilité dimensionnelle entre valeurs et attributs formels correspondants.

Sur le plan des vérifications de type, il doit y avoir compatibilité totale entre le type des valeurs et celui des attributs formels correspondants.

Enfin, dans une version ultérieure du Vérificateur, des emplacements de la liste des valeurs pourront être laissés vides. Les attributs formels correspondants prendront alors une valeur par défaut, spécifiée dans leur déclaration, ou dans le système même pour un niveau de langage donné.

Exemple de valuation d'attributs par l'ordre unité :

Considérons une diode ayant comme schéma équivalent une source de courant en parallèle avec une capacité (figure II.11) :

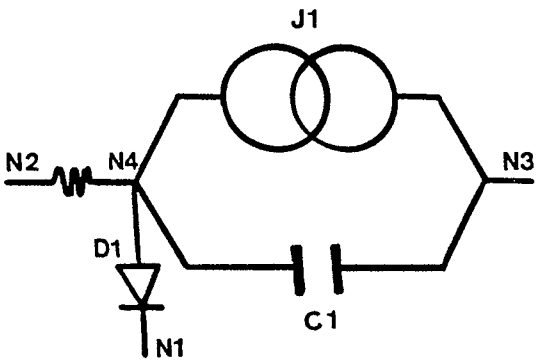



```
description D(REEL IDO, VALC1, QKT)
            (nd FIL P1, P2)
   corps
      decl VAR V1 ;
      unites J J1(P1, P2) ;
             C C1(P1, P2, VALC1) ;
      relations
         V1 := V(P1, P2) ;
         J1.VAL := IDO (exp(QKT*V1)-1)
fin
```

figure II.11

Les attributs à valeur réelle IDO, VALC1 et QKT représentent les paramètres technologiques de la diode.

Un exemplaire de cette description peut, par la suite, être utilisé pour définir un schéma équivalent de transistor (figure II.12).




```
description T(REEL VALR2, ALPHA, VC1)
             (nd FIL N1, N2, N3)
   corps
      decl VAR 15, NDEL N4 ;
      externe D ;
      unites R R2(N2,,VALR2) ;
             C C1( ,  N3, VC1) ;
             J J1(N3 , ,) ;
             D(1E-10, VC1, 40) D1( , N1)
   relations
      I5 := IP(D1.P1),
      J1.VAL    := ALPHA*I5,
      R2.N2 .= N4,
      C1.N1 .= N4,
      J1.N2 .= N4,
      D1.N1 .= N4
fin
```

figure II.12

Le transistor possède lui aussi des attributs technologiques : VALR2, ALPHA et VC1. Sa description T utilise la description D, paramétrée par les trois valeurs effectives 1E-10, VC1 et 40, où VC1 (valeur d'une capacité) est déjà lui-même attribut.

II.6.14. LES ASSERTIONS

Elles sont relatives à l'interface ou au corps. Dans une description, les assertions permettent de caractériser formellement le fonctionnement souhaité du modèle, dans un langage autre que le langage de description ; en particulier, les assertions, écrites sous forme de prédicats, peuvent servir à délimiter le domaine de validité du modèle, ou les conditions que doit satisfaire son environnement pour qu'un fonctionnement correct soit assuré.

En d'autres termes, elles permettent d'exprimer :

- des contraintes sur les attributs pour que la description ait un sens ;
- des vérifications sur le fonctionnement interne du modèle ;

- des contraintes sur l'utilisation de la description, en indiquant des combinaisons de valeurs interdites ou au contraire imposées sur les signaux d'interface pour assurer qu'une description est utilisée dans l'environnement pour lequel elle a été conçue ;
- des relations entre entrées et sorties.

Sous ces deux derniers aspects, les assertions d'une description écrite en CASCADE sont très semblables aux spécifications qu'un constructeur de circuits intégrés indique dans son catalogue.

Syntaxiquement, les assertions se présentent sous la forme d'expressions booléennes (conditions), évaluées durant la simulation, à la fin de chaque pas ou cycle de calcul. On dit que le modèle en cours de simulation est conforme aux spécifications tant que ces conditions sont toutes vraies. Si l'une d'entre elles au moins devient fausse, le modèle ne fonctionne plus comme prévu et la simulation est interrompue.

De plus, on peut désirer effectuer ces contrôles, non pas tout le temps, mais à des instants précis de la simulation seulement. Un langage de description d'évènements permet de définir ces plages d'observabilité (TPDL).

II.6.15. LA PARTIE FONCTIONNELLE D'UNE DESCRIPTION

Elle décrit les interrelations des différents composants de la description. Bien que spécifique à chacun des niveaux de langage, nous allons tout de même exposer ici l'ensemble des notions générales que l'on peut trouver dans cette partie.

II.6.16. LES FORMES CONDITIONNELLES "SI" ET "CAS" (INSTRUCTIONS ET EXPRESSIONS)

- "Si" :

si condition alors item_1 sinon item_2 finsi

- condition est une expression booléenne scalaire ;
- dans le cas d'une instruction conditionnelle, item_1 et item_2 sont des blocs de n'importe quelles instructions de la partie relations. La partie sinon est alors optionnelle.

Dans le cas d'une expression conditionnelle, item_1 et item_2 sont deux expressions. La partie sinon est ici obligatoire, et la condition valide une expression ou l'autre suivant sa valeur.

Exemple :

```
Si C[0:3] \ K = `10111 alors A.= B + C
                       sinon D.= B + C, R.= `0 fsi

CLOP.= si D alors   U   sinon   V   fsi
```

- "Cas" :

```
cas expression est
liste etiq1 : item_1 ;
liste etiq2 : item_2 ;
      :
liste etiqn : item_n ;
   sinon item_n+1    fincas
```

- Expression a pour résultat un entier, un littéral, une valeur logique ou un identificateur (étiqi).

- Chacun des cas correspond à une ou plusieurs valeurs de l'expression. Si cette dernière prend une valeur non prévue dans les différents cas, c'est le sinon, éventuel, qui est valide.

- Dans le cas d'une instruction "cas", les item_i sont des blocs de n'importe quelles instructions de la partie relations.

- Dans le cas d'une expression "cas", les item_i sont des expressions. C'est l'une d'elles qui est validée et la partie "sinon" est ici obligatoire.

Exemple :

Validation de certains groupes d'actions numérotés de 1 à 4, et codés dans une variable à valeur entière CODE.
cas CODE est :

1 : groupe_1
2 : groupe_2
3 : groupe_3
4 : groupe_4

Toutes ces formes conditonnelles peuvent être imbriquées entre elles, sans limitation de niveau.

II.6.17. INSTRUCTION RÉPÉTITIVE : BOUCLE "POUR"

C'est une facilité d'écriture permettant de macro-générer des instructions de la partie relations.
Sa forme est :

pour idf ensemble-parcouru repeter liste-d'instructions fin

L'ensemble parcouru par l'index idf est :

- Soit un intervalle donné par ses bornes (from ... step (optionnel) ... to ...), les bornes étant de type entier.
- Soit un type énuméré : in idf (nom du type).
- Soit un ensemble énuméré : in ensemble-fini (in {-2, 0, 253}), où les éléments sont des constantes signées.

REMARQUES :

Pour des raisons de difficulté de compilation et de délais, nous n'avons admis, pour le moment, qu'un nombre limité d'imbrications. Pour la même raison, les opérateurs puissance et division, non monotones, sont provisoirement interdits dans les expressions index contenant des indices de boucles "pour".

II.6.18. LES CONNEXIONS D'UNITÉS

Il s'agit ici de connexions de modules dans la partie relations, à l'aide d'objets appartenant à l'unité englobante ou accessibles de celle ci. Cette possibilité n'est pas prévue dans le cas général des modèles multi-niveaux. On ne peut l'utiliser que dans le cas particulier de modèles dont les composants sont décrits à un même niveau de modélisation. Ainsi, nous n'avons pas à utiliser ici de fonctions de liaison pour effectuer les raccordements. Les types de valeurs mis en jeu doivent être directement compatibles.

La forme d'écriture de cette connexion est la suivante :

nom-d'exemplaire connexion-d'interface

Contrairement à la connexion dans l'ordre unité, qui, elle, est permanente et ne peut jamais être invalidée, celle-là peut être conditionnée par des instructions "si", "cas", etc...

Exemple :

si COND alors W(A, B, C) sinon W(A,S,) fsi

Parallèlement à cette forme globale, il est possible de connecter individuellement les éléments d'interface d'un module par l'instruction normale de connexion du niveau de langage considéré. Ces liaisons peuvent aussi être soumises à condition. Nous développerons cette autre possibilité en fin de paragraphe.

On ne peut connecter qu'un seul interface par instruction de connexion. Dans le cas d'un tableau de modules, chacun doit donc être connecté individuellement et l'indexation qui le sélectionne doit être de dimension scalaire.

Contrairement à celle pratiquée dans l'ordre unité, la connexion traitée ici fait bien intervenir deux objets distincts non synonymes, c'est-à-dire ayant chacun sa zone valeur propre. Cette modélisation est caractéristisée par la notion de transfert de valeur d'un objet à l'autre, même dans le cas d'une connexion nd.

Exemple :

Reprenons le modèle DFF de bascule D déjà vu, et dont une nouvelle description est donnée ci-dessous. Dans cette version, les connexions des différentes portes sont situées dans la partie relations.

```
description DFF (in SIGB1 C, D ; out SIGB1 Q ; out SIGBO NQ)
   corps
      externe NAND2, NAND3 ;
      declarations
         SIGBO E, F, G, H ;
         unites NAND2 N1 ,
                      N2 ,
                      N4 ,
                      N5 ,
                      N6 ;
                NAND3 N3 ;
   relations
                      N1 (H, F, E) ,
                      N2 (E, C, F) ,
                      N4 (G, D, H) ,
                      N5 (F, NQ, Q) ,
                      N6 (Q, G, NQ) ,
                      N3 (F, C, H, G)
fin
```

Les objets pouvant être connectés aux éléments d'interface, dans ce contexte, sont les mêmes que ceux permis dans l'ordre unité, à savoir des porteuses ou des constantes. Nous ne détaillons pas à nouveau ces possibilités. Mais, dans le cas où l'élément d'interface correspond à une entrée, l'objet connecté peut être généralisé et consister en une expression dont les opérandes sont des objets connus et accessibles de la description du module englobant.

Exemple :

Aux niveaux Transfert de Registres, un réseau combinatoire peut être directement connecté à une entrée de boîte (in) sans que l'on soit contraint de réprésenter la sortie de ce réseau par un objet intermédiaire, connecté à son tour à l'élément d'interface.

Des emplacements peuvent aussi être laissés vides dans ce type de connexion. Cette forme d'écriture peut signifier la non-connexion définitive de l'élément d'interface correspondant (pattes en l'air). Elle peut aussi correspondre seulement à une liaison établie par ailleurs. Il est effectivement possible de connecter un interface en plusieurs fois, à l'aide de plusieurs instructions de connexion, y compris celle éventuelle de l'ordre unité, ayant des emplacements vides. Il y a effet cumulatif des connexions.

Exemple :

```
description PAT ...
   :
unité DTOTP TOTP (A[1:4], , 1, , ,) ;
   :
relations
   :
TOTP (, B & C, , , , M) ;
   :
si COND alors TOTP (, , , U, ,) sinon TOTP (, , , V, ,)
   :
fin
```

```
description DTOTP (in SIGBO P[1:4], E, F ;
                   out SIGBO G, H, I) ;
   :
```

Nous pouvons constater sur cet exemple que :

- l'unité TOTP est déjà partiellement connectée dans l'ordre unité ;
- l'avant-dernière sortie H de l'interface de TOTP n'est pas connectée ;

- l'entrée E est connectée à la sortie de l'expression B & C, locale à la description PAT ;
- l'entrée F est constamment maintenue à la valeur 1 ;
- la sortie G est connecté à U ou à V suivant la valeur de la condition COND.

etc...

Les objets ou expressions connectés à des éléments d'interface peuvent être des tableaux, indexés normalement. La sémantique de la liaison étant celle de la connexion normale dans le niveau de langage considéré, les vérifications dimensionnelles à effectuer sont celles attachées à cette notion, c'est-à-dire compatibilité dimensionnelle entre les deux éléments reliés.

Il en est de même pour la compatibilité des types de valeur mis en jeu.

Le tableau ci-après indique, pour certains niveaux de langage donnés en exemple et en fonction des sens autorisés des connexions, les compatibilités entre les types de porteuses primitifs attachés aux éléments d'interface et ceux éventuellement attachés aux objets connectés.

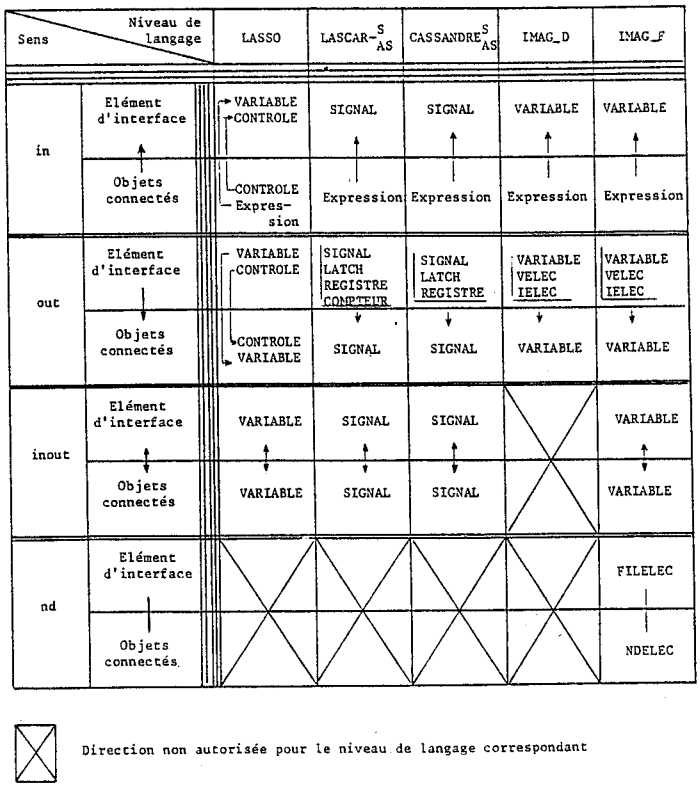


figure II.13

Par expression, on entend ici expression générale, pouvant être constituée de porteuses de tous types primitifs autorisés pour le niveau de langage correspondant, de fonctions, de constantes générales, d'attributs, etc...

Enfin, comme nous l'avons déjà indiqué, il est possible de connecter chaque élément d'interface par l'instruction de connexion explicite en vigueur dans le niveau de langage considéré. C'est le raccordement de l'élément d'interface, référencé par notation pointée, à un objet connu et accessible du module englobant. Une écriture mixte avec la forme connexion globale est possible, et il y a toujours effet cumulatif. Les règles de vérification définies précédemment restent valables.

Exemple :

Soit la configuration (figure II.14).

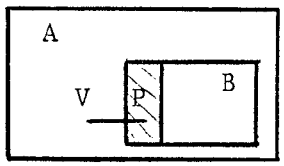


figure II.14

Dans la description du module A, l'écriture B (V, ...) et la connexion explicite de V et B.P sont parfaitement équivalentes.

II.6.19. LES EXPRESSIONS, OPÉRANDES ET OPÉRATEURS

Les expressions apparaissent fréquemment dans les textes CASCADE et peuvent présenter de nombreuses significations. Leur structure est unique et certaines de leurs caractéristiques sont paramétrées suivant les niveaux de langage considérés et suivant le contexte dans lequel elles sont utilisées. Elles sont vérifiées au cours de $\phi 1$. La vérification syntaxique est faite par l'analyseur issu de la grammaire CASCADE ; les vérifications de compatibilité dimensionnelle et de compatibilité de types sont faites en parallèle, par les actions de compilation.

Il n'est pas toujours possible, au cours de $\phi 1$, d'effectuer toutes les vérifications nécessaires (dimensionnelles ou autres), à cause de la présence éventuelle d'attributs dans le texte analysé, notamment dans la déclaration ou l'indexation de dimensions. Il en est de même pour les indexations dynamiques, évaluées à la simulation.

La voie qui s'ouvrait à nous pour traiter ces problèmes consistait à générer des relations sous forme d'expressions booléennes, à contrôler en différé dès que possible :

a) dans $\phi 2$ pour les expressions statiques dès que les attributs sont valués,

b) en cours de simulation pour les expressions dynamiques, chaque fois que le modèle est calculé.

Cette solution, rigoureuse mais difficile à mettre en oeuvre car complexe, a dûe être abandonnée compte tenu des délais de réalisation dont nous disposions. Ceci du moins dans le premier cas, car il n'y avait pas moyen de faire autrement dans le second. C'est ainsi que pour a) , une nouvelle version source de la description CASCADE du modèle est générée après substitution des attributs par leur valeur au cours d'une pré-phase 2 de préparation du modèle. Ce nouveau texte source, dans lequel n'apparait plus aucun attribut, est traité à nouveau par $\phi 1$, de façon totalement transparente pour l'utilisateur. Pour b) , par contre, le mécanisme de vérification dynamique est assuré par le code généré.

II.6.19.1. STRUCTURE DES EXPRESSIONS

La syntaxe des expressions présente les caractéristiques suivantes :

Une expressions est constituée d'opérandes et d'opérateurs. Un opérande est un objet représentant une valeur. C'est soit un objet défini dans le langage, soit une expression (générale ou sous-expression) contenant à son tour des objets. La notion d'expression est donc récursive.

Si leur définition ou leur construction le permet, les opérandes sont indexables : indexation d'objet, indexation d'expression. D'autre part, l'indexation peut, elle-même, contenir des expressions dont les opérandes peuvent être à leur tour indexés, etc... La notion d'indexation est aussi récursive. En fait, on a une double récursivité imbriquée (figure II.15).

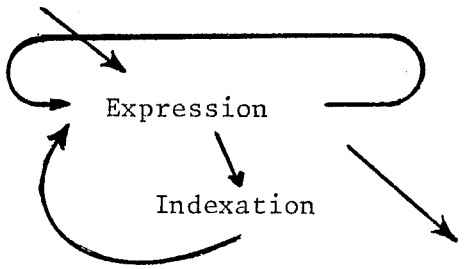


figure II.15

Tout le traitement des expressions doit constamment tenir compte de ces propriétés.

NOTE SUR LES INDEXATIONS D'EXPRESSIONS

Les expressions sont des constructions non déclarées. Pour pouvoir les indexer quand elles présentent des dimensions, on considère ces dernières comme normalisées, c'est-à-dire comme si elles étaient déclarées avec une numérotation de 1 à n de leurs éléments.

Exemple :

(A⌊0:7⌋ + B⌊12:5⌋) ⌈2:4⌉

représente A⌊1:3⌋ + B⌊11:9⌋

II.6.19.2. EVALUATION DES EXPRESSIONS

Pour des raisons d'optimisation du temps d'exécution, les expressions doivent être évaluées dès que possible dans la chaîne de traitement, et le minimum de fois.

Pratiquement, une expression est évaluée dès que les opérandes qui la constituent ont tous une valeur.
La connaissance de la valeur d'un opérande dépend de deux facteurs :

- la nature de l'objet ou des objets qui composent l'opérande (constantes, porteuses, etc...) ;
- la connaissance de la valeur de l'éventuelle indexation de l'opérande (récursivité).

Dans ce qui suit, nous classons les opérandes en différentes catégories suivant le moment à partir duquel leur valeur est connue, en ne tenant compte que du premier facteur. Pour cela, nous distinguons trois stades de traitement dans la chaîne des programmes : $\phi 1$, $\phi 2$ et la simulation. La plupart des objets susceptibles de servir d'opérandes dans une expression sont connus dans l'une de ces étapes seulement, toujours la même.

- Les constantes explicites sont connues dans $\phi 1$ (elles sont dites immédiates).
- Les attributs sont connus dans $\phi 2$.
- Les porteuses, variables muettes et fonctions sont connues à la simulation.

D'autres objets, par contre, ont leur valeur qui peut être connue à des moments différents suivant leur définition. Il s'agit des constantes déclarées et des indices de boucle "pour". En effet, il faut distinguer deux cas : celui où ces objets sont dépendants eux-mêmes directement ou indirectement d'attributs, et le cas contraire. Dans le premier, la valeur des opérandes n'est connue qu'au niveau de $\phi 2$, alors que dans l'autre elle l'est dans $\phi 1$.

Exemple :

Soit K un attribut et la déclaration de constantes:

```
ENT  C = 12, D = C - 3, E = 8 - K, F = 3 * (E - 1) ;
```

C et D sont connues dans $\phi 1$.
E dépendant directement de l'attribut K, n'est connu que dans $\phi 2$ et il en est de même de F qui dépend de E.

Le problème est le même pour les indices de boucle "pour" car les bornes de l'intervalle parcouru peuvent être fonction d'attributs. D'où, en résumé, la classification suivante, en trois familles :

- la classe 1 associée à $\phi 1$, comprenant les constantes explicites, et les constantes déclarées et indices de boucles "pour" indépendants d'attributs ;
- la classe 2 associée à la phase $\phi 2$ comprenant les attributs et les constantes déclarées et indices de boucle "pour" dépendants d'attributs ;
- la classe 3 associée à la simulation et comprenant donc tout ce qui est dynamique au sens de la trajectoire du modèle dans le temps, à savoir : porteuses, fonctions et variables muettes (voir les définitions de types de valeurs à propriété caractéristique).

Les attributs sont toujours de classe 2. Les porteuses et les appels de fonction sont toujours de classe 3. Les constantes explicites sont toujours de classe 1. Pour ces catégories, la classe n'est pas indiquée dans la structure de données, mais intégrée directement dans les programmes. Par contre, les constantes déclarées et les indices de boucle peuvent être de classe 1 ou 2 ; leur descripteur, dans la structure de données doit donc contenir cette information.

La notion de classe s'étend aux expressions. Le critère qui permet de déterminer la classe d'une expression est le suivant :

La classe d'une expression (1, 2 ou 3) est le maximum des classes de ses opérandes.

Exemple :

Soit l'expression A + B + C + D
où A et C sont de classe 2
B de classe 3
D de classe 1
L'expression est de classe 3 à cause de B.

Les expressions de classe 1 sont évaluées dès leur analyse dans $\phi 1$ et "remplacées par leur valeur". Celles des autres classes doivent être marquées 2 ou 3 en vue de leur évaluation ultérieure.

Les classifications ci-dessus ne tiennent pas compte de l'éventuelle indexation des opérandes. La classe d'un opérande indexé est la maximum entre la classe de l'opérande seul et celle de l'indexation.

Une indexation étant dans le cas général constituée d'expressions, la classe de cette dernière est le maximum des classes des différentes expressions, etc..., conformément à la double récursivité.

AUTRES MARQUAGES DES EXPRESSIONS

En vue de traitements ultérieurs, à savoir l'ordonnancement des actions dans $\phi 2$, il faut déterminer une nouvelle partition de certaines expressions : une expression associée à une condition "choix" ou "si" doit être marquée "instantannée" si et seulement si elle contient au moins un opérande à valeur immédiate (au sens de la simulation). Ce qui, pour CASSANDRE et LASCAR par exemple, est une porteuse non retardée de type SIGNAL.

II.6.19.3. PRINCIPE DE TRAITEMENT DES EXPRESSIONS DANS LE PASSAGE 1 DE $\phi 1$

Le traitement des expressions sous-entend le traitement des opérandes. Ces derniers étant ce que l'on peut appeler des objets utilisés, le traitement des opérandes d'expression n'est pas particulier, c'est-à-dire attaché au contexte "expression", mais est celui attaché à la notion d'utilisation d'objets en général.

De ce point de vue, les objets utilisés se répartissent en deux familles :

- ceux qui sont référencés par leur nom et donc habituellement préalablement déclarés ;
- les constantes explicites (ou immédiates).

Précisons seulement les principaux paramètres dont on doit disposer après ce traitement pour pouvoir effectuer celui des expressions :

- indicateurs d'existence ou de non existence de la déclaration de l'objet ;
- identification de la liste à laquelle il appartient (s'il est déclaré) ;
- sa localisation dans la liste.

De plus, on a vérifié que leur indexation éventuelle est correcte. La vérification syntaxique de l'expression est effectuée par l'analyseur de la grammaire CASCADE qui pilote aussi la vérification sémantique et dimensionnelle et la génération de structure et code.

La figure A2.1, en annnexe 2, montre le traitement général d'une expression CASCADE.

La figure A2.2, en annexe 2, exprime les différentes caractéristiques des expressions suivant le contexte dans lequel elles sont utilisées.

REMARQUES :

- Les indices de boucles "pour" sont inaccessibles dans la partie déclarations.
- Les variables muettes de propriétés caractéristiques sont inaccessibles dans la partie relation.
- Lorsqu'aucun opérande n'est obligatoire, cela veut dire que l'expression est composée d'au moins un opérande permis.

II.6.19.4. OPÉRATEURS

Le tableau suivant regroupe les différents opérateurs des niveaux du langage CASCADE du premier prototype, et résume leurs propriétés (figure II.16).

REMARQUES :

- La n-arité des opérateurs (leur nobmre d'opérandes) est notée de la façon suivante :

U pour unaire
B pour binaire
T pour ternaire
N pour N-aire

- Les types de valeurs prédéfinis sont notés en abrégé de la façon suivante :

L pour LOGIQUE
B pour BOOLEEN
Imp pour IMPULSION
E pour ENTIER
Car pour CARACTERES
Vet pour ETAT
R pour REEL
N pour NOMCONTROLE

- La référence à l'ensemble de ces types de valeurs est notée t (pour "tout").
- A niveau égal de priorité, les opérations sont effectuées de la gauche vers la droite de l'expression (ordre décroissant).
- Les mots clés donnés dans ce tableau pour représenter typographiquement certains opérateurs ne sont pas en général uniques. Ils existent souvent en français et en anglais et avec des synonymes.
- Rappelons que le type de valeurs BOOLEEN est exactement équivalent au type LOGIQUE, énuméré, restreint aux deux valeurs `0,`1.
- Le tableau donne seulement un aperçu des propriétés des opérateurs en ce qui concerne les dimensions des opérandes. Les spécifications rigoureuses de ces propriétés sont données à la suite.

A cette liste doivent être ajoutés les opérateurs :

- v (tension) et i (courant), pour les niveaux électriques IMAG ;
- pl (plus un) et ml (moins un) pour les niveaux LASCAR.

Leur emploi n'est pas aussi général que celui des opérateurs précédemment vus. Ils se comportent et sont traités plutôt comme des fonctions standards que des opérateurs, et ne sont utilisables que dans des contextes bien particuliers. Il faut leur ajouter une douzaine de fonctions arithmétiques classiques (log, sin, cos, tan, max, min, etc...).

Les opérateurs CASCADE

Terminal de la grammaire	Symbole typo. (lex)	Priorité (+ faible vers + forte)	n-arité	Signification	Fonctionnalité : Type	Fonctionnalité : Dimension
cou	|	1	B	ou logique	B × B → B	tab.T x tab.T —> tableau T
cnou	nor			non ou	L × L → L	
cnor	xor	2		Disjonction (ou exclusif)	B × L → L L × B → L	
ceqv	eqv			équivalence (non xor)	Imp × Imp → Imp	
cet	&	3		et logique	Imp × E → Imp } Si E=0 E × Imp → Imp } ou 1	
cnet	nand			nand (non et)		
cinf	<	4		inférieur	E × E, R × R, E × R, R × E, B × B, L × L → B ; Imp × Imp, L × B, B × L, E × Imp, Imp × E → B	
csup	>			supérieur		
cegalinf	=<			inférieur ou égal		
csupegal	>=			supérieur ou égal		
cegal	=			égal	t × t → B (même type)	tab.T x tab.T —> scalaire
cnonegal	¬= ou ~ =			différent (non égal)		
cplus	+	5		plus	E × E → E R × R → R E × R → R R × E → R (α)	tab.T x tab.T —> tableau T
cmoins	-			moins		
cmult	*	6		multiplication		
cslash	/			division		
cmod	mod			modulo	E × E → E	
cpuissance	^ ou ↑	7		puissance	idem α	
cconcat	\	8		concaténation	t × t → t (même type) L × B → L B × L → L Ex Imp → Imp } Si E=0 Imp × E → Imp } ou 1	accole suivant 1° dimension
cpuissance	^ ou ↑		U	détection front montant (dérivation logique)	B → Imp L → Imp	tab. T —> tab.T
cdescend	~ ou ↑¬			détec. front descendant		
cplus	+			plus unaire	E → E R → R	
cmoins	-			moins unaire		
cnon	~ ou ¬			non logique	B → B L → L B si 0 ou 1 Imp → Imp E → Imp si E=0 ou 1	
cdollard	$	9	B	conversion en entier positif	L → E, B → E } ∈ N	tab.T —> tab.T' enlève une dimension
cintval	intval			conversion en entier relatif (codage en complément à 2)	L → E, B → E } ∈ Z	
cabs	abs			valeur absolue	E → E R → R	tab.T —> tab.T
cpourcent	%			retard n unités de temps	E × t → t, ∈N+	scal.×tableau T → tableau T
cfan	fan			fan-out		ajoute une dim.
cconvert	convert			conversion d'entier positif sur n bits	E × E → B, ∈N	scal.×tableau T → tableau T' ajoute une dimension
cbinval	binval			conversion d'entier relatif sur n bits (codage en complément à 2)	E × E → B, ∈N ∈Z	
creduc	reduc		N	réduction (par rap. aux opérateurs binaires de priorité 1,2,3,5,6)	vecteur t → t tableau t → tableau t (dim. n) (dim.n-1) (même type) où t = type autorisés pour les opérateurs binaires 1,2,3,5,6	supprime la première dimension non dégénérée
ctransp	transp		T	transposition:permutation des deux dim. spécifiées	E × E, t → t, ∈N+	permute deux dim. non dégénérées
cderiv	d	10	U	dérivation réelle	R → R	tab. T → tab. T

Figure II.16

D'autre part, signalons que les formes conditionnelles si et cas des expressions, bien que non répertoriées dans le tableau, sont aussi des opérateurs.

DÉTAIL DES PROPRIÉTÉS DIMENSIONNELLES DES OPÉRATEURS

a) OPERATIONS BINAIRES ENTRE TABLEAUX ET PORTANT SUR ELEMENTS CORRESPONDANTS

Ce sont les opérations dont la fonctionnalité dimensionnelle est notée : Tableau T x Tableau T → table T ou scalaire.

Il s'agit des opérations de comparaison, logiques binaires et algébriques binaires.

- Les deux opérandes doivent être des tableaux dimensionnellement compatibles.
- Les opérations s'effectuent entre éléments correspondants.
- Elles ont ici leur signification habituelle et le résultat est un tableau dimensionnellement compatible avec les opérandes, sauf dans le cas de l'égalité et de la non-égalité où la comparaison porte globalement sur l'ensemble de chaque tableau opérande et où le résultat est un booléen scalaire.

b) OPERATIONS UNAIRES PORTANT SUR LES ELEMENTS D'UN TABLEAU

Ce sont les opérations : "non" logique, valeur absolue, détection de front, plus et moins unaires et conversion booléen → entier sans paramètre de longueur.

- Les opérations s'effectuent sur chacun des éléments du tableau avec leur signification habituelle (voir les langages IMAG, CASSANDRE, LASCAR et LASSO).
- Le résultat est un tableau dimensionnellement compatible avec le premier sauf dans le cas de la conversion où l'on obtient un tableau avec une dimension en moins, celle qui correspond à la transformation d'un vecteur de booléens en un nombre entier.

c) OPERATIONS BINAIRES ENTRE UN TABLEAU ET UN SCALAIRE

Il s'agit du retard, des conversions entier → booléen avec paramètre de longueur et de l'opérateur fan.

- Ces opérations s'effectuent entre le scalaire et chacun des éléments du tableau (distributivité).
- Avec le premier opérateur, on obtient un tableau d'éléments, tous retardés de n unités de temps, dimensionnellement compatible avec l'opérande tableau.
- Avec les autres, on obtient un tableau ayant une dimension supplémentaire :
  - celle qui correspond, dans le cas des conversions, à la transformation d'un nombre entier en un vecteur de booléens ;
  - celle qui correspond, dans le cas de l'opérateur fan, à la transformation de chacun des éléments de la première dimension non dégénérée en une composante constituée de n fois cet élément.

Un scalaire devient alors un vecteur, un vecteur une matrice, etc...

Notons que l'opérateur fan n'agit pas sur la valeur des éléments du tableau opérande mais sur la structure de ce dernier. Il en est de même pour les opérateurs qui vont suivre.

Sur le plan réalisation, signalons que ces opérateurs ont été implantés en tant qu'opérateurs unaires "qualifiés".

d) CONCATENATION

Elle accole deux tableaux suivant la première dimension non dégénérée. Les dimensions autres que la première doivent être compatibles.

Cette opération ne rajoute pas de dimension. La dimension du résultat est la même que celles des opérandes mais avec une première dimension qui a pour nombre d'éléments la somme des nombres d'éléments des premières dimensions non dégénérées des opérandes.

Exemples :

Soit les tableaux :

A[2 : 7 ; 0 : 7 ; 1 : 36]

et B[1 : 10 ; 1 : 8 ; 3 ; 11 : 46]

A\B est un tableau 16 x 8 x 36 obtenu en juxtaposant A et B parallèlement à la première direction.

Pour deux vecteurs, R[1 : 5]\S[3 : 1] est un vecteur à 8 éléments.

e) REDUCTION

Elle s'applique avec certains opérateurs binaires (voir tableau des opérateurs) et agit suivant la première dimension non dégénérée, qui le devient après l'opération.

Le résultat a donc une dimension non dégénérée de moins que l'opérande.

Le résultat de la réduction d'un vecteur est un scalaire, celui de la réduction d'une matrice un vecteur, etc...

Exemple :

L'expression "reduc + A[1 : 4 ; 1 : 2]" a pour résultat un vecteur à deux éléments dont chacun est la somme des quatre éléments de la ligne correspondante du tableau A.

f) TRANSPOSITION

Elle permute les dimensions spécifiées, en ne tenant pas compte de celles qui sont dégénérées. L'opérande doit donc avoir au moins deux dimensions non dégénérées. Le résultat est un tableau ayant le même nombre d'éléments, mais répartis différemment.

Exemple :

Le résultat de l'expression

transp!1,3! A[1 : 4 ; 2 ; 0 : 7 ; 8 : 0]

est un tableau 9 x 8 x 4.

II.6.20. LES TRANSFERTS

La notion de transfert de valeur, sous la forme générale "réceptrice← expression" , est présente dans la majorité des niveaux de langage CASCADE.

Exemple :

- Les connexions de signaux CASSANDRE et LASCAR.
- Les chargements de registres dans les mêmes niveaux.
- L'affectation LASSO.
- L'affectation IMAG.
- La connexion d'éléments d'interfaces de sens "in" à un objet, pouvant être éventuellement une expression (dans la partie "relations" seulement).
- etc...

Si certains de ces transferts présentent des aspects spécifiques, ils obéissent cependant tous à un traitement général que nous allons exposer ici.

Les vérifications attachées à ces transferts sont les suivantes :

COHERENCE DIMENSIONNELLE :

La compatibilité doit être assurée entre l'expression et la réceptrice.

CONTRAINTES SUR LES TYPES DE PORTEUSES :

Celles attachées aux opérandes de l'expression dépendent du contexte et ont été déjà vues figure A2.2, en annexe 2. Celles attachées à la réceptrice dépendent aussi de la sémantique du transfert ; voir chacun des niveaux de langage pour cela.

Exemple :

Dans un chargement, en CASSANDRE, la réceptrice doit être un objet de type REGISTRE.

COHERENCE ENTRE TYPES DE VALEURS :

Dans le cas où le type de valeur de la réceptrice est prédéfini, il doit y avoir identité entre ce type et le type de valeur résultant de l'expression. A quelques exceptions près toutefois car les conversions implicites suivantes sont admises :

LOGIQUE ← BOOLEEN
ENTIER ← IMPULSION
REEL ← ENTIER

Dans le cas où le type de valeur de la réceptrice est défini par l'utilisateur, il doit y avoir compatiblité au sens précédent du terme entre son type origine et le type de valeur résultant de l'expression. Un contrôle supplémentaire, à la simulation, est alors nécessaire pour vérifier, à chaque envoi dans la réceptrice, que la valeur de l'expression appartient bien au type de valeur de celle-ci.

Dans le cas très spécial de liaisons bi-directionnelles comme par exemple dans les équations ou les connexions d'éléments d'interface de sens "nd" ou "inout", il doit y avoir identité totale entre les types de valeur résultants des deux membres, qu'ils soient prédéfinis ou définis par l'utilisateur.

REMARQUES :

Relativement à ces notions, un certain nombre de compléments de vérifications statiques peuvent être envisagés :

- Le contrôle des conflits de transfert, c'est-à-dire la vérification qu'une réceptrice ne reçoit pas simultanément plus d'une valeur, est effectué d'une façon systématique à la simulation. Et ceci vaut tant pour les transferts explicites que pour les connexions d'interfaces. Il serait très intéressant de détecter le maximum de ces conflits dès la vérification, quand cela est possible.
- Le contrôle de l'appartenance de la valeur d'une expression au type de valeur non prédéfini de sa réceptrice peut être statique et effectué dès la vérification si la valeur de l'expression est constante vis à vis de la simulation.

II.7. LA REALISATION DES NIVEAUX DE LANGAGE DANS CASCADE

Un niveau est défini par sa grammaire propre (syntaxe), et par la sémantique qui l'accompagne. Cette grammaire est une sous-grammaire de la grammaire globale CASCADE. Elle est constituée de règles spécifiques et en grande partie de règles communes à tous ou à plusieurs niveaux.

Des actions de compilation placées au début des ensembles de règles spécifiques vérifient s'il est légitime d'aller exécuter ces règles compte tenu du niveau de langage courant. On a ainsi une grammaire unique, paramétrée pour traiter la syntaxe de chacun des niveaux.

Les notions sémantiques les plus importantes sont communes, et correspondent à une paramétrisation du noyau. Ce sont, pour un niveau donné, les définitions :

- des types de valeurs prédéfinis ;
- des types de porteuses primitifs ;
- des combinaisons possibles entre ces types de valeurs et de porteuses ;
- du complément de langage "système" contenant les segments prédéfinis pour ce niveau : types, descriptions, fonctions, procédures.

Les actions de compilation de traitement et de vérification correspondantes sont aussi paramétrées par niveau.

A chaque niveau de langage correspond un ensemble de types de porteuses primitifs et de types de valeurs prédéfinis, significatifs pour ce niveau. En compilation, un premier contrôle est effectué en vérifiant, pour le niveau de langage concerné, que le type de porteuse ou de valeur utilisé est bien autorisé (voir figure II.17 ci-après). On vérifie ensuite, sur un tableau unique, que les combinaisons types de porteuses <—> types de valeurs rencontrées dans ce niveau sont permises (voir figure II.4 dans II.6.6.1).

Les niveaux de langage générés ainsi dans CASCADE sont IMAG (Lef.82), CASTOR (Mer.84), POLO (BoM.84), CASSANDRE (Bre.84a), LASCAR (Bre.84a) et LASSO (Bre.84b, Bor.85a).

On trouvera en annexe 3, en illustration de ces notions, la réalisation des niveaux CASSANDRE et LASCAR de CASCADE.

	NIVEAUX DE LANGAGE				
Types de porteuses	LASSO	LASCAR-$^{A}_{AS}$	CASSANDRE-$^{A}_{AS}$	POLO	IMAG
Registre		X	X		
Signal	X	X	X	X	
Latch		X	X		
Compteur		X			
Variable	X				X
Contrôle	X				
Filelec					X
Ndelec					X
Varsyst					X
////////////	////////////	////////////	////////////	////////////	////////////
Types de valeurs					
Logique		X	X	X	
Booléen	X	X	X	X	
Impulsion		X	X		
Entier	X	X	X		
Caractères	X	X			
Etat		X	X		
Réel				X	X
Nomcontrôle	X				

Figure II.17

CHAPITRE III

ORGANISATION DE LA PREMIÈRE PHASE DE COMPILATION CASCADE ET TRAITEMENTS DE VERIFICATION

La gestion des données CASCADE n'entrant pas dans le cadre de ce mémoire, signalons que le présent chapitre, s'il se réfère aux principales données manipulées dans $\phi 1$, ne tient pas compte des notions de projets, modèles, versions, niveaux de langage, "points de vues", etc. Nous évoquerons tout au plus, pour fixer les idées, une structure théorique d'accueil des données de $\phi 1$ que nous dénommerons "catalogue". Au fur et à mesure des besoins, nous subdiviserons ce dernier en "sous-catalogues" correspondantsaux différentes catégories de données rencontrées.

III.1. ORGANISATION DE Φ1, DEFINITION ET PRINCIPE DE TRAITEMENT DES DONNEES

III.1.1. PRINCIPES DE COMPILATION DES MODÈLES ÉCRITS EN LANGUAGE CASCADE

Deux importantes propriétés du langage influent sur le traitement, il s'agit de la prise en compte, par le langage, de la modularité des systèmes décrits et de la règle de "non référence avant".

III.1.1.1. MODULARITÉ

Il est intéressant d'exploiter le plus longtemps possible, dans la logique des traitements, la propriété de modularité de la modélisation des systèmes électroniques. D'où une compilation segmentée en deux parties : phase 1 (Φ1) et phase 2 (Φ2).

Pour Φ1, la notion de modèle, en tant qu'ensemble de modules générés à partir de descriptions, n'existe pas. Cette phase permet à l'utilisateur de vérifier et compiler ses descriptions une à une, indépendamment les unes des autres, sans préjuger du fait que telle et telle descriptions soient référencéees dans le même modèle (ou pas). La construction du modèle complet et la compilation finale sont dévolues à Φ2.

Ce choix de réalisation offre les avantages suivants :

- Souplesse maximum pour l'utilisateur, sur le plan de la méthode, car il n'a pas l'obligation d'écrire tout son modèle avant d'en effectuer la compilation. Il peut ainsi vérifier ce dernier par morceaux (au sens de Φ1) et même s'intéresser à plusieurs modèles en même temps.

- Minimisation du temps d'exécution nécessaire à la mise au point du modèle, car la compilation globale de ce dernier (Φ2) n'est entreprise que lorsque toutes les descriptions nécessaires sont correctes (au sens de Φ1). De plus, en cas de modification locale du modèle, par exemple après détection d'une erreur de conception en cours de simulation, seules les descriptions modifiées doivent à nouveau être traitées par Φ1.

Cette structuration du compilateur en deux phases devrait être d'autant plus efficace qu'un maximum de vérifications est effectué sur les descriptions, indépendamment les unes des autres.

III.1.1.2. "NON RÉFÉRENCE AVANT"

Cette règle introduit une contrainte d'ordre, non seulement sur les déclarations et utilisations de notions dans un texte écrit en langage CASCADE (a), mais aussi entre les compilations de notions dépendantes les unes des autres (b et c).

Les règles de précédence sont les suivantes :

a) Les objets locaux (porteuses, valeurs, exemplaires de descriptions) et les descriptions, les types, les fonctions et les procédures (appelés segments) doivent avoir leur déclaration ou définition qui précède leur utilisation dans le texte que l'on veut compiler (sauf dans les cas de déclarations implicites).

b) Si l'on veut compiler un texte faisant référence à des compléments de langage, ces derniers doivent avoir été préalablement compilés.

c) Les descriptions, fonctions et procédures doivent avoir subi la compilation de leur interface ou en-tête avant d'être utilisées. Ceci afin de pouvoir procéder, dans leur connexion ou leur appel, à des vérifications de compatibilité et de cohérence fortement souhaitables.

III.1.2. ORGANISATION DE LA PHASE Φ1 DE COMPILATION (Bre.83a)

III.1.2.1. DONNÉES D'ENTRÉE

Une "unité de traitement", pour Φ1, est constituée d'un "paquet" de descriptions, en nombre quelconque, sans liens obligatoires entre elles. Dans la mise en oeuvre du compilateur CASCADE, cette solution est un compromis entre la souplesse offerte à l'utilisateur et la complexité du traitement.

Dans une unité de traitement, les descriptions doivent être écrites dans un ordre satisfaisant la contrainte de non référence avant. De même, dans un complément de langage, constitué d'un ensemble de définitions de types, fonctions, procédures et descriptions, la seule restriction concerne le respect de la règle de non référence avant.

Pour des raisons d'homogénéité et d'optimisation, c'est le même compilateur qui réalise Φ1 pour les descriptions et les compléments de langage. Un complément de langage est en général assez volumineux pour constituer à lui seul une unité de traitement.

Pour des raisons de commodité de réalisation du compilateur, la grammaire CASCADE représente directement les unités de traitement telles qu'on vient de les définir. D'où son aspect au niveau de la racine :

axiome → ref-langage segment
segment → suite-de-descriptions
def-de-langage

III.1.2.2. TRAITEMENT GÉNÉRAL DANS Φ1 DES DESCRIPTIONS, FONCTIONS, PROCÉDURES ET COMPLÉMENTS DE LANGAGE

Φ1 traite de façon indépendante les unes des autres les descriptions de boîtes, paramétrables ou pas, écrites par l'utilisateur en langage CASCADE. Elles sont traduites en structures internes et stockées dans leur catalogue.

La compilation des modèles CASCADE est d'autant plus efficace que les descriptions composantes sont mises sous une forme optimum pour alléger Φ2. Il faut donc que le maximum du travail soit effectué par Φ1.

Ce dernier s'effectue en deux traitements, largement dirigés par la structure syntaxique des données traitées :

- Le premier a pour rôle de vérifier les descriptions sources écrites en langage CASCADE, d'en éditer un listing et d'en effectuer un premier codage interne. Le module correspondant à ce traitement de vérification est appelé "VERIFICATEUR".

- Le deuxième a pour but de transformer la structure produite par le Vérificateur en une structure synthèse destinée au catalogue. Le module effectuant cette opération est appelé "CATALOGUEUR" (Cau.84).

Φ1 offre à son tour une efficacité maximale si le vérificateur, sur lequel on boucle tant qu'il y a des erreurs dans le texte traité, effectue tout le travail de vérification possible à ce moment là, à l'exclusion de tout autre traitement. La structuration et le codage du texte sont évidemment nécessaires mais ils restent le plus proche possible du langage d'entrée, afin de ne pas alourdir ce passage déjà suffisamment complexe. Le reste des transformations doit donc être effectué par le catalogueur, lorsque le texte analysé est considéré comme correct.

Vu ainsi, le Vérificateur produit donc une forme abstraite générale, intermédiaire. N'effectuant que la Vérification, il n'a connaissance que de la sémantique suffisante pour assurer cette tâche. Le reste des traitements sémantiques de Φ1 est dévolu au Catalogueur, qui produit une forme abstraite spécialisée.

COMPILATION DE L'UNITÉ DE TRAITEMENT

a) Cas d'une suite-de-descriptions dont aucune ne contient de description, fonction ou procédure définie interne

Les deux modules de Φ1 traitent séquentiellement et entièrement la suite-de-descriptions en une seule exécution : les descriptions sont d'abord toutes traitées lors de l'exécution du premier module, puis toutes lors de l'exécution du second. Les données sont transmises d'un module à l'autre à l'aide d'un fichier intermédiaire. Ce fichier est standard et constitué d'enregistrements de longueur variable. Chaque fois que le Vérificateur effectue une exécution, il y stocke ses résultats en écrasant ceux qui y avaient été rangés lors de l'exécution précédente. Ces données constituent l'entrée du Catalogueur.

Enfin, ce dernier range les descriptions produites dans le catalogue associé à Φ1 (voir figure III.1).

On a alors perdu toute trace de la notion d'unité de traitement, c'est-à-dire d'organisation des descriptions sources.

Les descriptions sont stockées dans le fichier intermédiaire et dans le catalogue, dans l'ordre suivant lequel elles ont été écrites dans la suite-de-descriptions.

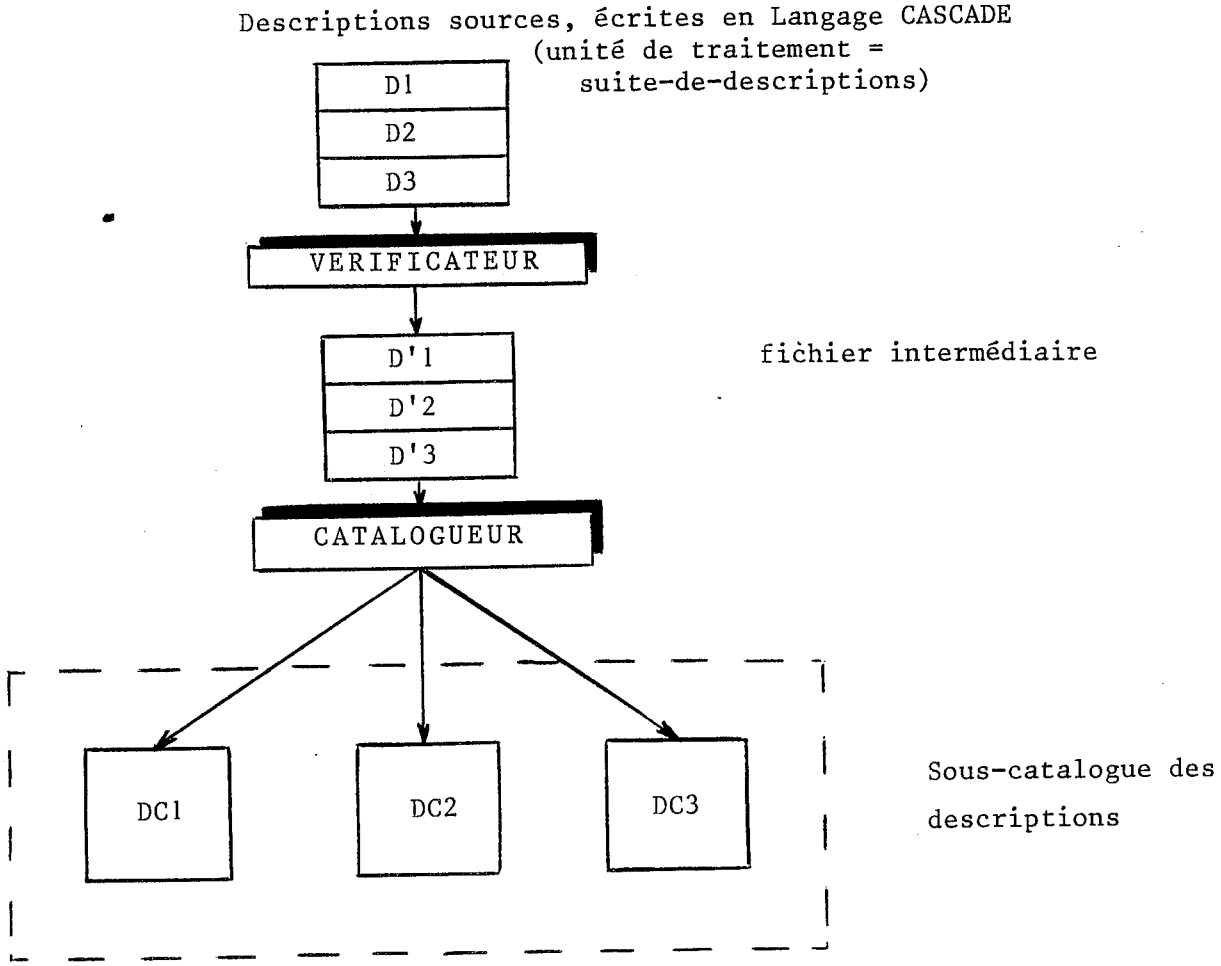


Figure III.1

b) Cas général d'une suite-de-descriptions, c'est-à-dire contenant des descriptions, fonctions, ou procédures définies internes (imbriquées)

Dans le passage de vérification, les inclusions de définition sont traitées par empilement dans un espace de travail. Quand le traitement de l'un des segments est terminé, on vide celui-ci dans le fichier de sortie. On obtient ainsi une "mise à plat" de ces inclusions. Le fichier intermédiaire est organisé suivant le même principe que précédemment et le cataloguer n'a plus à prendre en compte ces imbrications de définitions (voir exemple figure III.2). Les ambiguïtés de noms peuvent être évitées par qualification du nom du segment extrait par ceux de ses contenants.

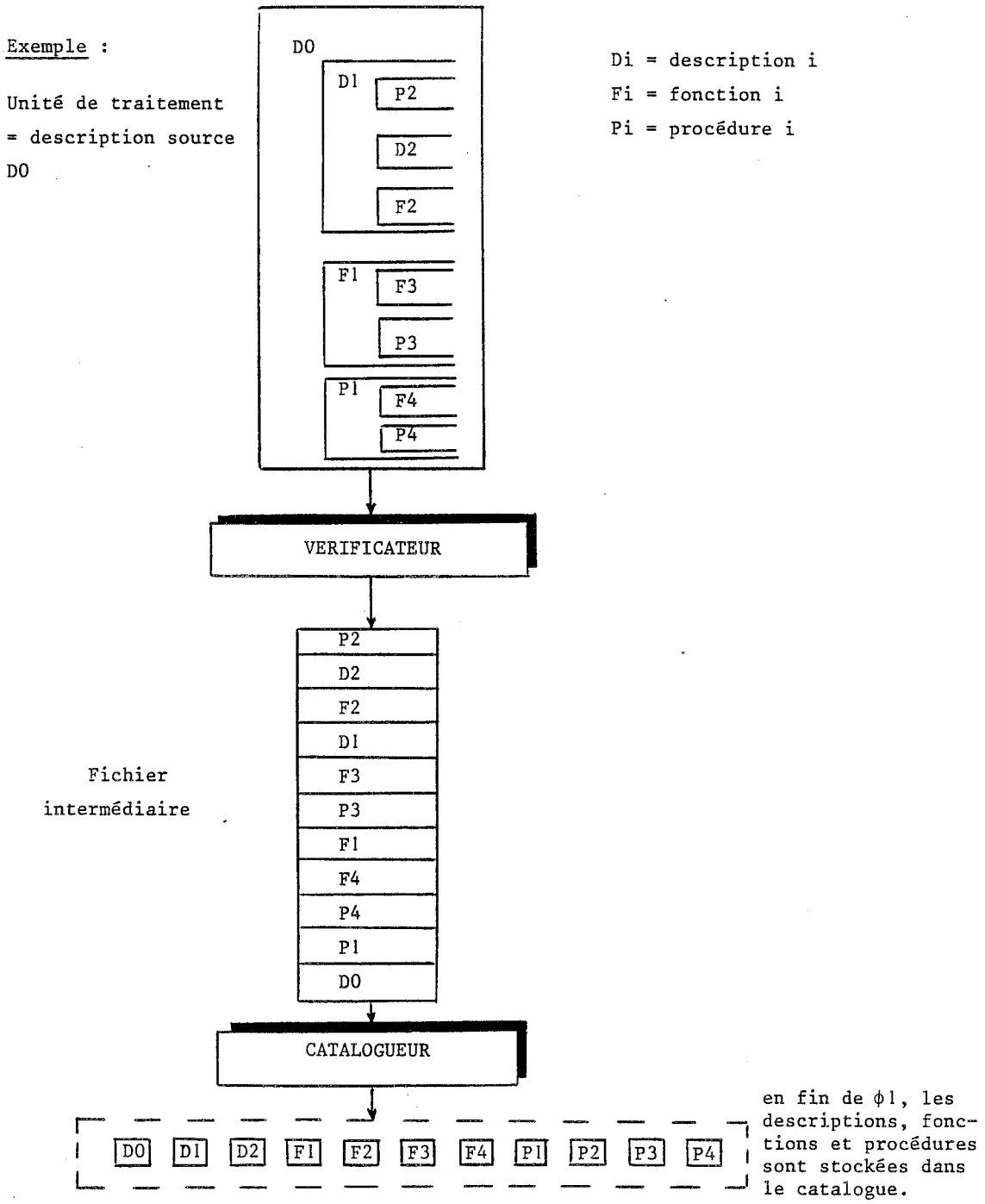

Exemple :
Unité de traitement
= description source
DO
DO
D1
P2
D2
F2
F1
F3
P3
P1
F4
P4
Di = description i
Fi = fonction i
Pi = procédure i
VERIFICATEUR
P2
D2
F2
D1
F3
P3
F1
F4
P4
P1
DO
Fichier
intermédiaire
CATALOGUEUR
DO
D1
D2
F1
F2
F3
F4
P1
P2
P3
P4
en fin de ϕ1, les descriptions, fonctions et procédures sont stockées dans le catalogue.

Figure III.2

III.1.2.3. TRAITEMENT DES INTERFACES ET EN-TÊTES

Durant la compilation d'un segment, lorsqu'on traite l'utilisation d'un autre segment : description, fonction ou procédure, il est nécessaire de connaître l'interface ou l'en-tête de cet autre segment.

L'accès à ces informations est indispensable au vérificateur si l'on désire qu'il effectue les vérifications de compatibilité lors du traitement des appels de la fonction ou de la procédure et des connexions des unités générées par la description. De plus, dans le cas d'une utilisation de description et des unités générées par cette dernière, le vérificateur et le catalogueur complètent la structure du segment en cours de traitement avec des informations relatives à l'interface de cette description.

Pour des raisons de cohérences évidentes, au moment de leur consultation, les interfaces ou les en-têtes doivent être corrects au sens "vérificateur" du terme et sous une forme aisément exploitable. Ceci est réalisé de la façon suivante : l'utilisateur décrit et compile d'abord par Φ1, les segments contenant les interfaces ou en-têtes en question. Lors de l'analyse par le vérificateur, de l'interface ou de l'en-tête, on génère, en plus de la structure interne du segment, un descripteur qui est ensuite stocké dans le sous-catalogue prévu à cet effet. Ensuite, au cours de chacun des deux passages de compilation du segment "utilisateur", dès la rencontre de la première référence au segment utilisé, le descripteur d'interface ou d'en-tête de ce dernier est lu dans le sous-catalogue et stocké dans une zone de travail accessible. Il y est conservé jusqu'à la fin du traitement de l'unité de traitement courante par le passage de compilation, pour y être consulté autant de fois que nécessaire (voir figure III.3).

Sur le plan de la méthode, cette façon de procéder amène deux remarques :

- Soit un segment S2 utilisant un autre segment S1. S1 doit donc être compilé avant S2. Si ces segments se trouvent dans deux unités de traitement

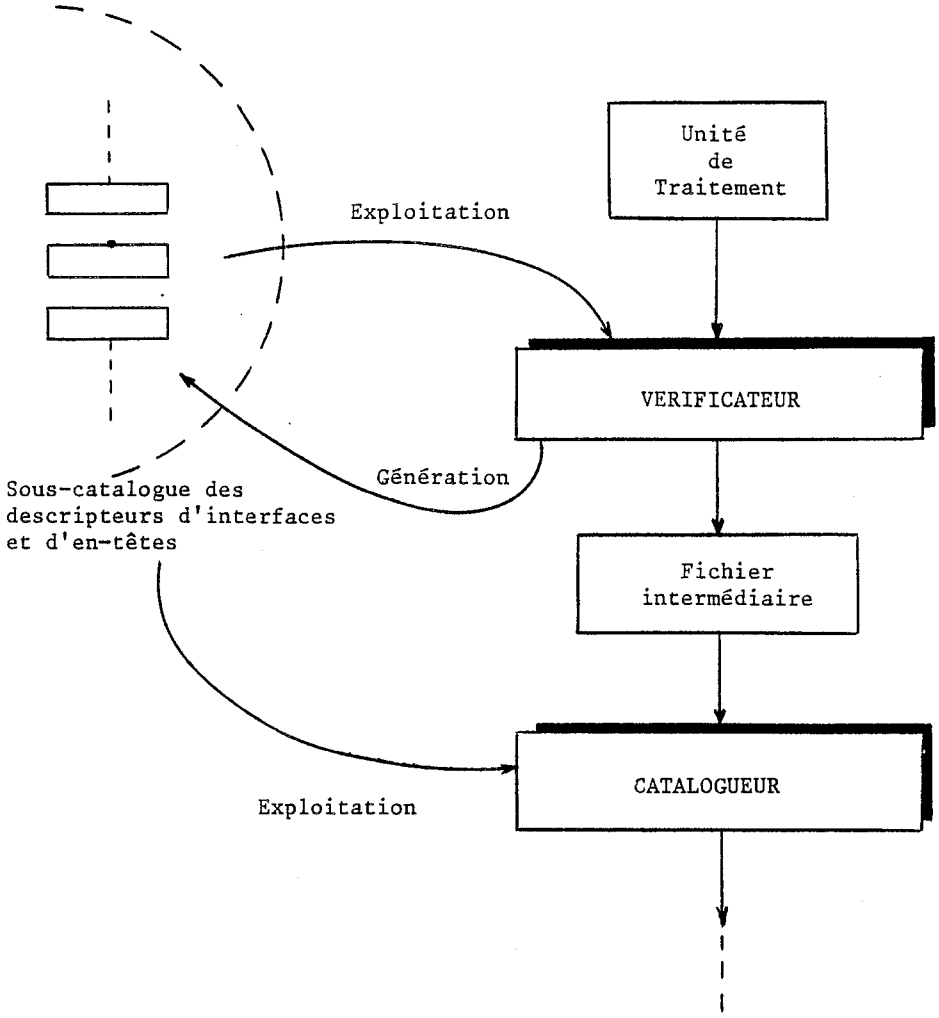


Figure III.3

différentes (respectivement U1 et U2), alors S1 et S2 sont traitées par deux exécutions différentes et U1 doit être compilée avant U2. Dans le cas contraire, S1 et S2 sont contenus dans la même unité de traitement (et éventuellement dans un même segment). Il faut et il suffit, alors, que S1 soit placé avant S2. Ici, les deux segments sont traités au cours de la même exécution.

- Lorsque l'utilisateur conçoit un modèle, il peut désirer compiler immédiatement certaines descriptions déjà écrites,

Supposons qu'il porte son attention sur une description particulière D, contenant des connexions d'unités ou des appels de fonctions ou procédures. Les segments référencés peuvent n'être pas entièrement écrits ; seule est indispensable la description de l'interface ou de l'en-tête. Leur compilation préalable permet de compiler correctement à son tour la description D. Ces segments "creux" devront ultérieurement être complétés et recompilés. Si leur en-tête ou interface n'est pas modifié, il est inutile de recompiler la description D.

NOTE : Une facilité a été introduite tout récemment par les réalisateurs de Φ1, afin que l'utilisateur puisse s'affranchir de l'ordre obligatoire dans la compilation des descriptions. Il s'agit d'une extension de l'ordre "externe", dans lequel on peut spécifier la définition des interfaces des descriptions externes utilisées. Comme l'ordre en question est toujours placé avant le "use", la contrainte de non référence avant est respectée.
Mais cette solution ne règle pas la question de la vérification de l'identité de l'interface spécifié dans l'ordre externe et de celui défini dans le texte de la description déclarée externe correspondante, et définie par ailleurs.

III.1.2.4. TRAITEMENT DES COMPLÉMENTS DE LANGAGE

Lorsque l'unité de traitement est un complément de langage, l'effet de Φ1 est le suivant :

- Les descriptions, fonctions et procédures définies dans le complément de langage subissent le traitement normal et sont stockés dans le catalogue.

- Le complément de langage proprement dit donne lieu à la génération d'un descripteur, construit au cours de la vérification dans l'espace de travail, puis copié dans le sous-catalogue des compléments de langage. Ce

descripteur est constitué de la définition des types et des listes des noms des descriptions, fonctions et procédures définis dans le complément de langage.

La figure III.4 montre le traitement d'un exemple.

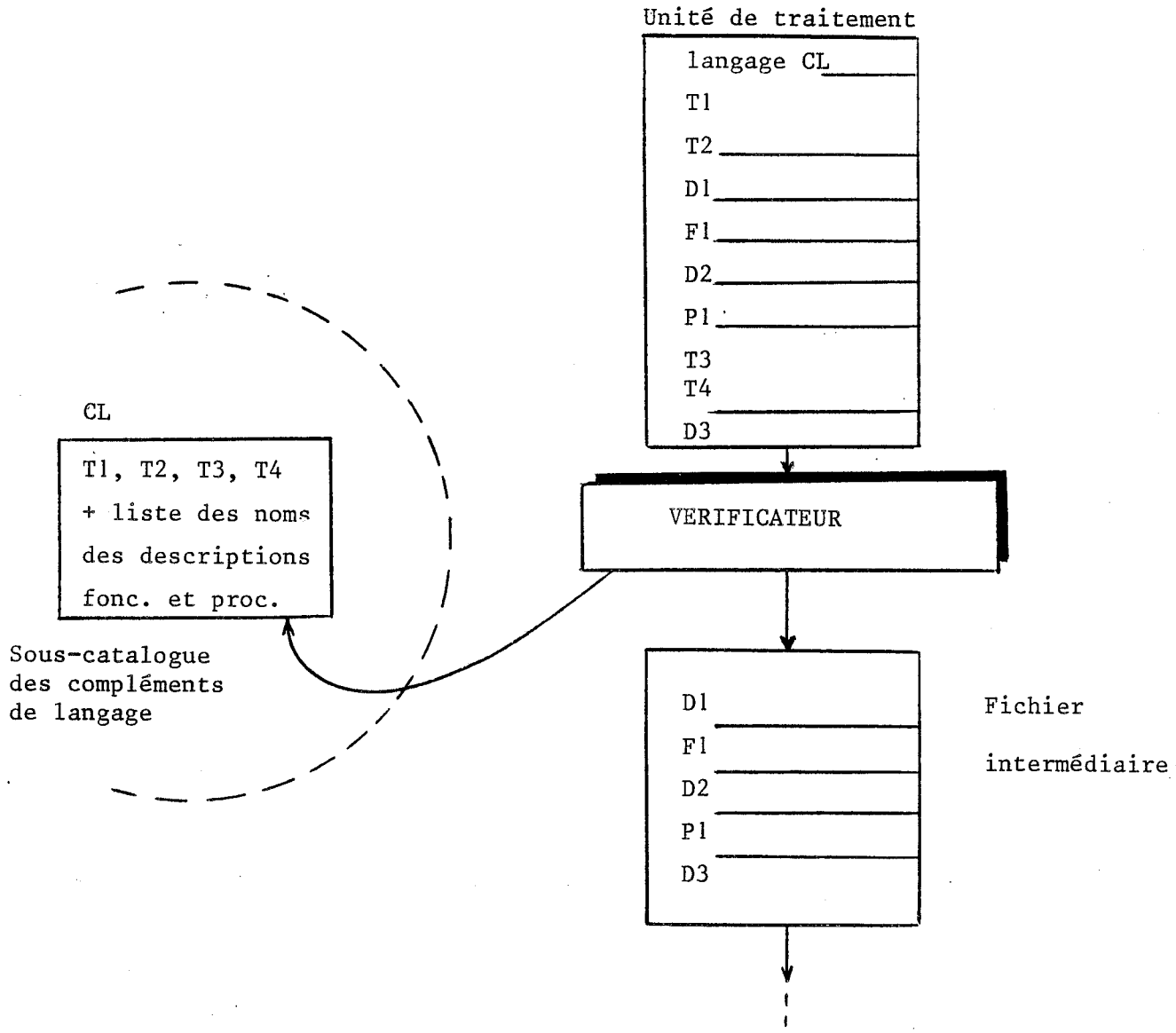


Figure III.4

Un complément de langage est utilisé lors de la définition d'une suite-de-description ou d'un autre complément de langage si son nom apparaît dans l'ordre reflan de l'unité de traitement considérée.

Lors de l'analyse de l'ordre reflan, on va chercher dans le sous-catalogue les compléments de langage spécifiés, y compris le complément de langage "système" qui précise le niveau de langage considéré. On les stocke dans une zone où ils sont accessibles durant toute la vérification de l'unité de traitement. En pratique, ces compléments de langage représentent une extension de la zone dans laquelle sont conservés les noms des segments accessibles dans cette unité de traitement.

La figure III.5 montre le traitement d'un exemple.

Remarque : Le catalogueur n'utilise pas les compléments de langage.

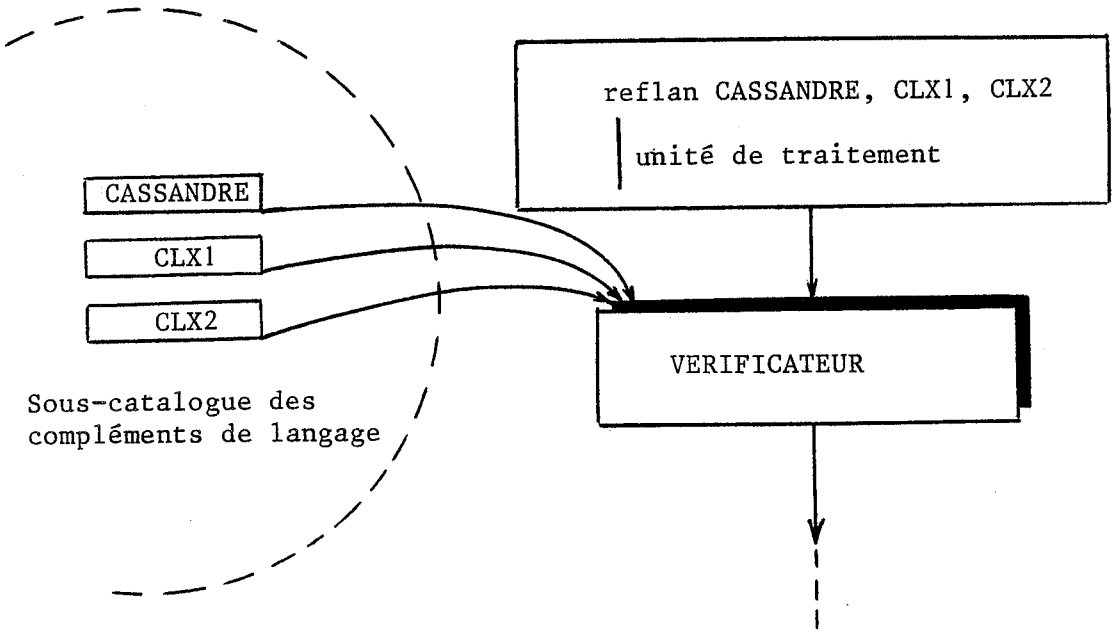


Figure III.5

III.1.2.5. RÉCAPITULATION DES PRINCIPALES DONNÉES ASSOCIÉES À Φ1

Vis-à-vis de Φ1, toutes les données en entrées/sorties relèvent du catalogue, sauf celles qui sont stockées dans le fichier intermédiaire où elles sont gérées par programme. Ces données sont des segments à différents stades de leur traitement :

- Les unités de traitement : suites-de-descriptions et compléments de langage source, en entrée du vérificateur

- Les descriptions, fonctions et procédures compilées par Φ1

- Les interfaces de descriptions, les en-têtes de fonctions et procédures, et les compléments de langage, tous compilés par le vérificateur et repris par lui-même et par le catalogueur.

REMARQUE :

Les fonctions et procédures écrites dans un langage de programmation (par exemple FORTRAN ou PASCAL) ont leur définition scindée en deux parties :

- L'en-tête est écrit selon la syntaxe de CASCADE. Son écriture apparaît en position de définition normale.

- Le corps du segment est un sous-programme destiné à être compilé par le compilateur du langage de programmation. Son texte est écrit séparément.

La liaison entre les deux parties est établie par l'identificateur de la fonction ou de la procédure.

La figure III.6 récapitule l'ensemble des principales catégories de données mises en jeu dans Φ1.

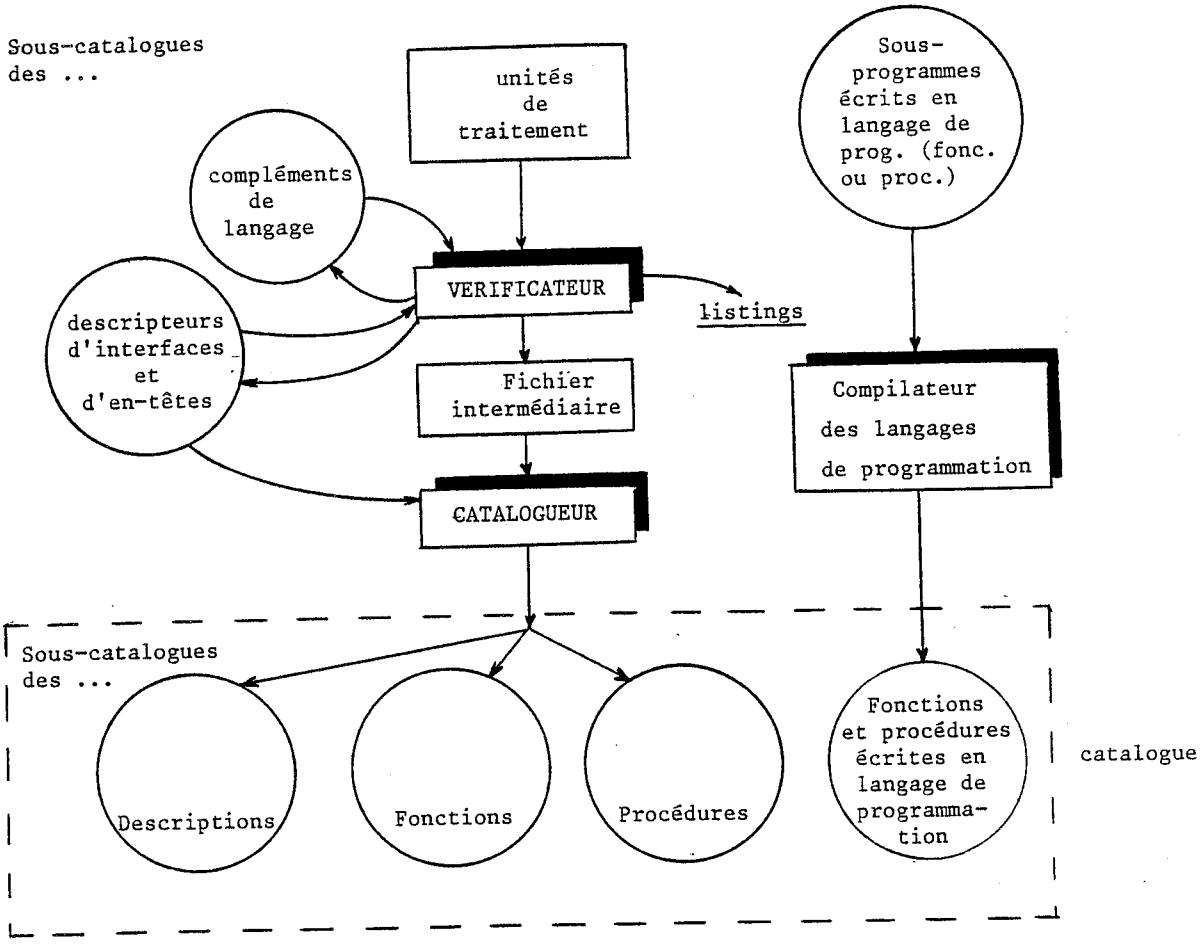


Figure III.6

Notons enfin qu'aux niveaux Electrique (IMAG) et Portes Logiques (POLO), les composants primitifs sont modélisés dans CASCADE par des descriptions prédéfinies dont l'interface seulement est déclaré dans les compléments de langage système IMAG et POLO, tandis que leur fonctionnement est interne au système CASCADE.

III.2. LE VERIFICATEUR

III.2.1. MISE EN OEUVRE DE LA TÂCHE

Il s'agit de développer ici les caractéristiques, la description, la structure et l'organisation du Vérificateur.

III.2.1.1. LES CARACTÉRISTIQUES DE LA VÉRIFICATION CASCADE

La Vérification statique des modèles CASCADE se ramène à l'exécution de deux analyses, bien classiques dans le domaine de la compilation des langages, l'analyse lexicographique et l'analyse syntaxique.

a) L'analyse lexicographique a pour fonction de reconnaître et coder les "mots" du langage de description. C'est l'analyse dite morphologique des données d'entrée. Dans CASCADE, le module qui remplit ce rôle assure aussi l'édition du texte traité et celle des éventuels diagnostics d'erreurs.

b) L'analyse syntaxique effectue les vérifications syntaxiques proprement dites, suivant les règles de la construction des phrases du langage (voir grammaire), ainsi que les vérifications de sémantique statique : contrôles de cohérence des types dans l'utilisation des objets, contrôles relatifs à la déclaration et à l'utilisation des objets, contrôles des compatibilités dimensionnelles, etc.

La description est traduite en un ensemble de listes correspondant aux déclarations des objets et en une chaîne codée correspondant à la partie fonctionnement.

Rappelons qu'à ce stade-là des traitements il n'est pas question de comprendre toute la sémantique des descriptions traitées mais seulement celle qu'il est nécessaire de connaître pour effectuer les vérifications statiques.

La Vérification CASCADE est effectuée en un seul passage, piloté par la syntaxe du texte d'entrée. Les opérations a) ainsi que celles citées dans b) sont donc menées de front. Les motivations qui nous ont amenés à adopter cette voie sont les suivantes :

- Si l'on considère que dans un compilateur structuré en plusieurs passages, on ne procède généralement au passage suivant que lorsque tous les précédents n'ont détecté aucune erreur, cette solution permet théoriquement de détecter le maximum d'erreurs en le minimum de bouclages sur le Vérificateur, lors de la correction des modèles. D'où une meilleure efficacité pour ce qui est de la durée de mise au point des modèles par l'utilisateur, et aussi sur le plan de la consommation de temps machine.

- La lexicographie étant menée en parallèle avec tout le reste de l'analyse et des vérifications, la ligne source courante reste en permanence disponible et il est alors aisé de marquer calligraphiquement l'emplacement des erreurs détectées à signaler. Dans le cas d'un Vérificateur structuré en plusieurs passages, comme il est hors de question, pour des raisons de coût évidentes, de recommencer à chaque fois la lexicographie, la connaissance, par le programme, de la forme externe du texte d'entrée est perdue au moins dès après le premier passage. Le marquage direct des erreurs dans le texte édité peut être effectué tout au long de cette première étape mais devient impossible au-delà, lors des passages ultérieurs. A moins que l'on accepte de traîner, au cours de ces passages

successifs, tous les liens nécessaires entre chaque unité syntaxique du texte analysé et son codage interne. Ceci permettant de garder en permanence la correspondance entre forme interne et forme externe et de répercuter ainsi un repérage précis. Mais au prix de quelle lourdeur de structure et de traitement.

- Enfin, CASCADE, qui lors de la définition de ses principes de base, se présentait comme un langage "déclaratif" et sans "références avant" (tout objet utilisé devait, au préalable, être obligatoirement déclaré) se prêtait très bien à la technique du passage unique.

III.2.1.2. PRINCIPE D'UTILISATION DU TRGLL1 POUR LE PASSAGE DE VÉRIFICATION CASCADE

La figure III.7 montre le schéma de principe du Vérificateur.

Le module d'analyse lexicographique, à la demande de l'interpréteur standard, accepte les caractères du code source et les regroupe afin de former des symboles pour l'analyseur syntaxique. Chacun de ces symboles est codé, ce qui rend le texte à analyser plus compact et plus simple à manipuler. L'analyseur syntaxique compare ces codes avec ceux des symboles terminaux présents dans les représentations des règles de grammaire, et vérifie ainsi la conformité du texte analysé.

Les actions de compilation et de vérification sémantique sont automatiquement appelées au cours de ce processus.

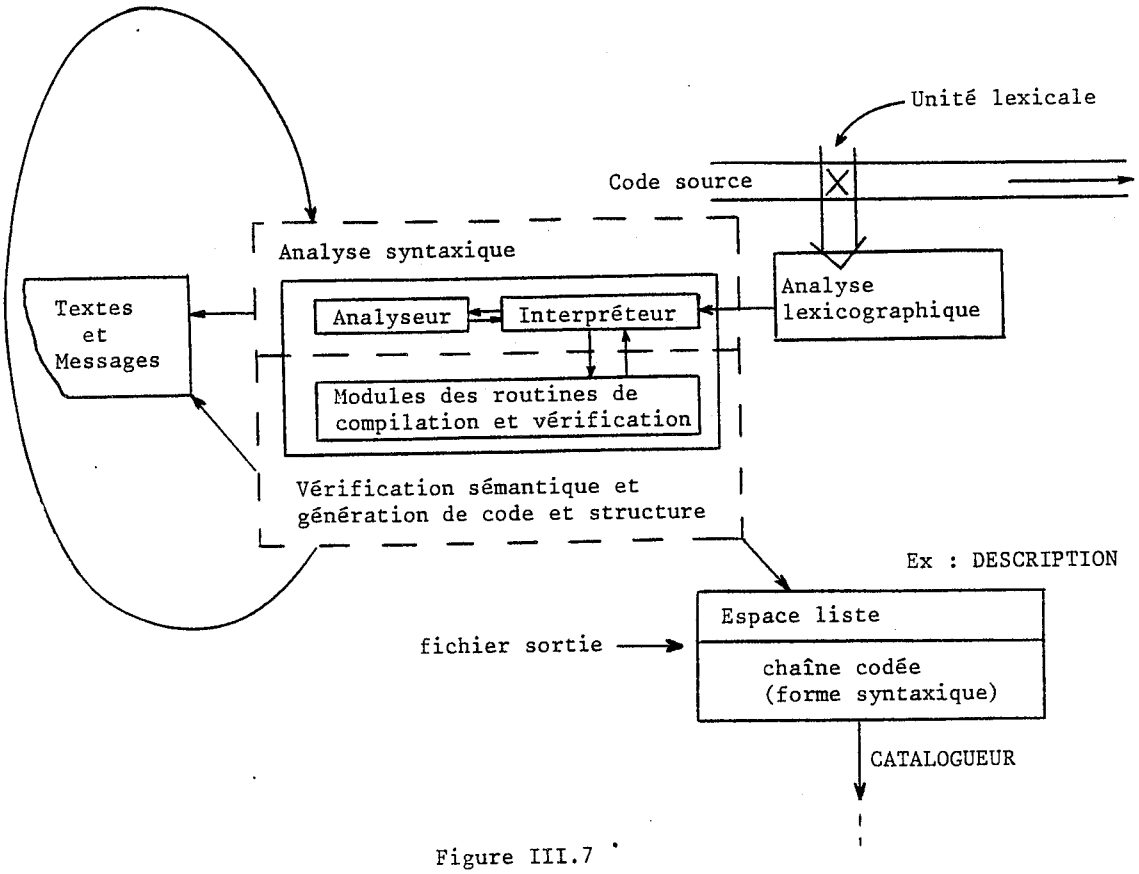


Figure III.7

Exemple :

Soient les règles suivantes relatives à la déclaration d'objets, sous la portée de leur type déclarateur, et à l'utilisation de ces objets.

```
déclaration_objets ---> nom_type objet F1 suite_liste_objets
suite_liste_objets ---> (  )
                        objet F1 suite_liste_objets
        .
        .
        .
affectation        ---> objet_affecté := ... objet_utilisé F2...
        .
        .
        .
```

F1 et F2 sont par exemple des appels à des fonctions respectivement de traitement de la déclaration d'un objet (mise en table ...) et de traitement de l'utilisation de cet objet (vérification de son type ...).

Contrairement à d'autres systèmes utilisant le même outil d'aide au développement de compilateurs, comme IRENE-C décrit dans Bou.74 par exemple, nous ne permettons pas à l'utilisateur de modifier à son gré le code de l'analyseur, produit par le TRGLL1 à partir de la grammaire, afin d'appeler des fonctions sémantiques de son choix. Nous estimons en effet que de telles opérations ne doivent pas être de son ressort.

III.2.1.3. LES SORTIES, LES OPTIONS

L'exemple vérifié ci-après (figure III.8) montre comment est présenté le texte édité. La colonne U.L. est constituée des numéros d'ordre de la première unité lexicale de chaque ligne ; une unité lexicale étant, rappelons-le, un mot-clé, un identificateur, une constante ou un symbole simple ou double.

L'utilisateur a tout loisir pour demander l'édition de ce listing sur disque ou au terminal. Il en est de même pour les diagnostics d'erreurs.

Notons que le Vérificateur pourrait être rendu très facilement paramétrable sur ces points, afin que l'utilisateur puisse fixer lui-même les cas par défaut de son choix.

Enfin, suivant le matériel utilisé, le Vérificateur fonctionne avec ou sans les caractères minuscules et avec un code de représentation interne ASCII ou EBCDIC.

```
                                VERIFICATION DE : AUTO16V1.CAS
                                LE : 28-FEB-84  A : 13:58:21
                                NIVEAU DE LANGAGE : LASCAR+S [3]

LIGNE   U.L.

   1      1   lanref LASCAR+S
   2      3    description AUTO16V1
   3      5     (in HORLOGE H ;
   4     10      in SIGB0 EM[0:15],DEPART,S[0:15],RS ;
   5     30      out REGB0 A[0:15],B[0:15] ;
   6     46      out SIGB0 SM[0:31],OCCUPE)
   7     57    corps
   8     58    declaration
   9     59     SIGB0 RC0,RC1,RC2,RC3 ;
  10     68     REGB0 BUSY,C[0:3],A1[16:31],R ;
  11     87    relation
  12     88     OCCUPE .= BUSY ,
  13     92     RC3    .= `1 ,
  14     96     RC2    .= RC3 & C[3] ,
  15    105     RC1    .= RC2 & C[2] ,
  16    114     RC0    .= RC1 & C[1]
  17    122     :REPOS: !H! B <= EM , si DEPART alors charger INIT ,
  18    138                                          BUSY <= `0 fin ;
  19    143     :INIT:  !H! A  <=  fan 16!`0 ,
  20    156                 A1 <= B ,
  21    160                 B  <= EM ,
  22    164                 C  <= `0000 ,
  23    168                 charger ADDITION ,
  24    171                 BUSY <= `1 ;
  25    175     :ADDITION: !H! si A1[31] alors  A <= S ,
  26    191                                     R <= RS
  27    194                                sinon  R <= `0  fin ,
  28    200                    C <= (RC0 \ RC1 \ RC2 \ RC3) xor C ,
  29    214                    charger DECAL ;
  30    217     :DECAL: !H! A  <= R \ A[0:14] ,
  31    234                 A1 <= A[15] \ A1[16:30] ,
  32    248                 si reduc!C alors  charger ADDITION
  33    255                              sinon  charger FIN  fin ;
  34    260     :FIN: SM .= A\A1 ,
  35    269           !H! charger REPOS ;
  36    275   findescription


                        +-+- AUCUNE ERREUR DE VERIFICATION -+-+
```

Figure III.8

III.2.1.4. LES ERREURS, DÉTECTION ET RÉCUPÉRATION

On distingue trois catégories d'erreurs :

a) les erreurs lexicales, qui correspondent à la mauvaise construction d'une unité lexicale du langage ;

b) les erreurs de syntaxe, qui correspondent à une phrase dont la construction grammaticale est incorrecte ;

c) les erreurs sémantiques, qui correspondent à une impossibilité d'attribuer un sens à une phrase ou à un élément de phrase syntaxiquement correct.

La première catégorie d'erreurs est traitée par l'analyseur lexicographique. Une erreur lexicale provoque souvent une erreur de syntaxe car le codage de l'unité lexicale incorrecte est souvent impossible.

La deuxième catégorie d'erreurs est détectée automatiquement par l'interpréteur standard, lorsque le code courant n'est pas l'un de ceux permis, à cet instant-là, par l'analyseur. Il y a erreur de syntaxe et émission d'un diagnostic approprié. L'interpréteur peut ensuite essayer de se récupérer, tout d'abord en tentant lui-même une correction :

- par substitutions successives du symbole incriminé par d'autres présumés corrects (correction de la classique "faute de frappe" par exemple) ;

- par insertions successives de ces symboles avant le symbole incriminé (cas de l'oubli d'un symbole).

Si ces essais s'avèrent infructueux, l'interpréteur peut alors commander le saut d'une partie du texte qui suit l'occurrence de l'erreur, jusqu'à ce qu'il puisse se recadrer convenablement. On utilise pour cela des unités lexicales servant de repères, appelées "symboles importants". C'est le "mode panique".

Cette récupération n'est pas automatiquement entreprise par l'interpréteur lors de l'occurrence d'une erreur de syntaxe. Il demande tout d'abord à l'écrivain du compilateur, par le biais de fonctions de communication, quelle(s) action(s) parmi celles citées il doit exécuter. Ce qui permet de traiter chacune des erreurs individuellement, cas par cas si on le désire.

Ces mécanismes se sont avérés très délicats d'emploi pour le langage CASCADE, nous ferons le point là-dessus un peu plus loin.

Enfin, les erreurs sémantiques sont traitées directement par les actions de compilation. Les erreurs de cette catégories peuvent être classées, très grossièrement, suivant deux niveaux :

- celles portant sur les objets mis en jeu dans le modèle :

  . un identificateur a plusieurs sens car il est déclaré plusieurs fois ;

  . un identificateur n'a pas de sens car il est utilisé sans avoir été déclaré ;

  . l'indexation d'un objet sort des bornes définies par la déclaration de ses dimensions.

  etc.

- celles portant sur les phrases :

  . cohérence dans la cohabitation de deux objets dans une phrase ;

  . cohérence entre un objet et le contexte dans lequel il est utilisé.

  etc.

L'exemple donné ci-après, figure III.9, montre la présentation d'un texte CASCADE incorrect, avec ses diagnostics d'erreurs correspondants.

Les erreurs sont localisées dans le texte par le symbole $.

Les messages sont constitués des éléments suivants :

- Le type de l'erreur : c'est en fait son origine, pour aider éventuellement l'utilisateur à mieux comprendre ce qui ne va pas

  . LEX pour une erreur d'origine lexicographique
  . SYN pour une erreur syntaxique
  . SEM pour une erreur sémantique
  . SYS pour une erreur système.

- Le numéro d'identification du message, afin d'établir une correspondance avec les explications détaillées de l'erreur, dans la documentation utilisateur.

- Le numéro de la ligne du texte dans laquelle l'erreur a été détectée.

- Le numéro de l'unité syntaxique incriminée (en plus du symbole $ la repérant).

- Le texte bref du message.

Il serait bon de préciser aussi le degré de sévérité de l'erreur commise : erreur utilisateur ou système, avertissement, etc.

Enfin, dans le texte des messages, les unités lexicales doivent apparaître en clair, lorsqu'on y fait référence.

```
                                VERIFICATION DE : PLA2.CAS
                                LE : 23-APR-85  A : 09:12:46
                                NIVEAU DE LANGAGE : LASCAR+AS [5]

 LIGNE   U.L.

     1     1   reflan LASCAR+AS
     2     3   description PLA2 (in BTM0 RES;in BREG0 S[1:3];in BTM0 RDY,DEC;
     3    25                     out BTM0 W[1:3],STROBE,INC,SDR,RW)
     4    42   body
     5    43        declare BTM0 ROW[1:12];
     6    52        relations
     7    53
     8    53           << and plane >>
     9    54
    10    54               ROW[1] .= ~S[1] & ~S[2] & ~S[3] & RES,
    11    79               ROW[2] .= ~S[1] & ~S[2] & ~S[3] & ~RES,
    12   105               ROW[3] .= ~S[1] & ~S[5] & S[3] & RDY,
***  ERREUR                                        $
    13   129               ROW[4] .= ~S[1] & ~S[2] & S[3] & ~RDY,
    14   154               ROW[5] .= ~S[1] & S[2] & ~S[3],
    15   176               ROW[6] .= ~S[1] & S[2] & S[3] & ~DEC,
    16   200               ROW[7] .= ~S[1] & S[2] & S[3] & DEC,
    17   223               ROW[8] .= S[1] & ~S[2] & ~S[3] ,
    18   245               ROW[9] { S[1] & ~S[2] & S[3] & ~DEC,
***  ERREUR                          $
    19   269               ROW[10] .= S[1] & ~S[2] & S[3] & DEC,
    20   292               ROW[11] .= S[1] & S[2] & ~S[3] & ~RES,
    21   316               ROW[12] .= S[1] & S[2] & ~S[3] & RES,
    22   339
    23   339                  << or plane >>
    24   340
    25   340              W[1] .= reduc ! ROW[7:11],
    26   354              W[2] .= reduc ! ROW[3,5,10:11],
    27   372              W[3] .= reduc ! ROW[2,4:6,8],
    28   390              RW   .= reduc ! ROW[1:2,4:12],
    29   405              SDR  .= reduc ! ROW[2,4,6],
    30   418              INC .= ROW[5] ! ROW[9] ,
    31   430   STROBE .= ROW [9]
    32   436   enddscription << PLA >>
***  ERREUR                 $
    32   438

       MESSAGE(S)  DE  VERIFICATION DE : PLA2
                                LE : 23-APR-85  A : 09:12:46

COD G  NM   NL   NU :  TEXTE DU MESSAGE

    DESCRIPTION PLA2

SEM    33   12  120 :  LES BORNES D"INDEXATION SONT HORS DES DECLARATIONS

SYN    47   18  248 :  ERREUR SYNTAXIQUE

LEX     7   32  435 :  LE MOT-CLEF EST INCONNU (HORS SPECIFICATIONS)

SYN    47   32  435 :  ERREUR SYNTAXIQUE
```

Figure III.9

III.2.1.5. ORGANISATION DES PROGRAMMES

Le schéma suivant, figure III.10, emprunté à Madame Ginette BUISSON, montre la position des sous-programmes, des fichiers et des données partagées, assurant l'interface entre l'interpréteur et le reste du Vérificateur.

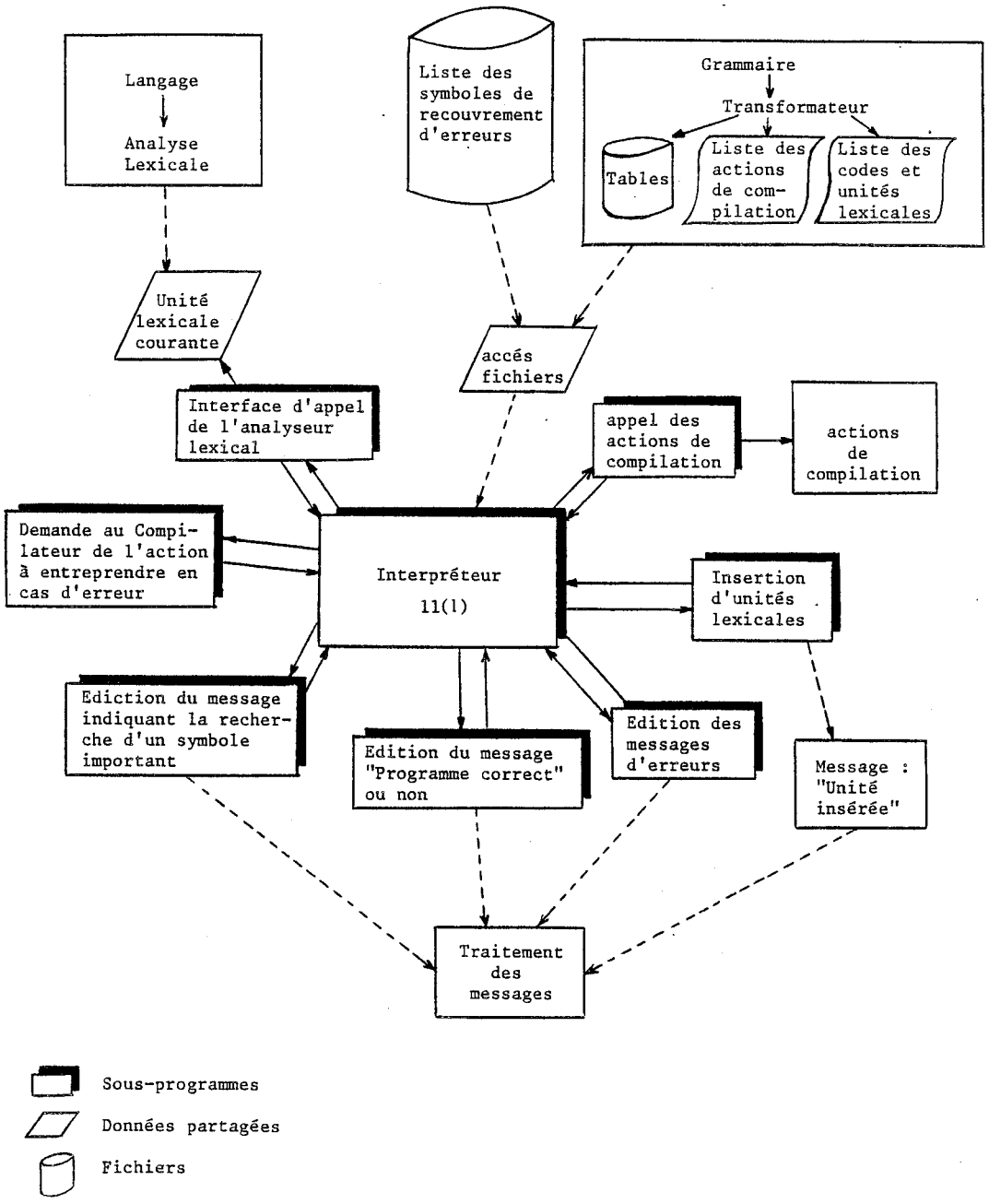


Figure III.10

III.2.1.6. REMARQUES SUR LA MISE EN OEUVRE DU VÉRIFICATEUR

Nous pouvons citer au moins deux critères, parmi les plus importants, qui permettent de juger cette partie des compilateurs qu'est la Vérification des textes d'entrée. Il s'agit de la rapidité et de la fiabilité de la détection et du traitement des erreurs.

Pour le premier, il semble que la complexité du langage soit l'une des causes principales de la consommation importante de temps CPU actuelle, surtout depuis que de nombreux cas particuliers ont été ajoutés aux notions de base du langage de départ ; ces entorses aux règles primitives se justifiant, évidemment, sur le plan langage de description.

Une possibilité de simplification du traitement de cette complexité consisterait à structurer cette partie du compilateur à raison d'une grammaire par niveau de langage ou par famille de niveaux proches. La diminution du nombre de tests qui en découle devrait améliorer les performances ainsi que la compréhension des programmes. Par contre, cette solution aurait pour inconvénient d'alourdir notablement la gestion du logiciel, en nécessitant la répercussion fréquente des modifications sur les n grammaires.

Le deuxième point évoqué dans ce paragraphe est le plus important. Il est relatif aux erreurs de syntaxe, les erreurs lexicographiques et de sémantique ne présentant pas de difficultés particulières.

L'interpréteur standard détecte à coup sûr, en principe, toute erreur syntaxique, mais la récupération, par utilisation des mécanismes de substitution et d'insertion, s'avère parfois peu maîtrisable. Notons, au passage, que, d'une façon générale, les techniques utilisées pour le rattrapage d'erreurs dans les compilateurs disponibles sont très variées. Il n'existe pas de standard en la matière. (Bac.79)

Il est nécessaire, dans notre application, d'apporter des soins constants et tous particuliers aux traitements d'erreurs portant sur chaque terminal de la grammaire. De plus, nous avons souvent recours au rattrapage sur symboles importants : fin de bloc ou séparateur d'éléments de liste par exemple (mode panique), Une gestion des contextes doit, pour cela, être faite par l'écrivain du compilateur.

Dans cette optique, notons un problème important qui se pose lorsque l'analyse d'une partie du texte est abandonnée. Comme la syntaxe pilote aussi la construction de la structure interne, lors de la reprise de l'analyse on risque de trouver des incohérences telles que des listes non correctement terminées, des descripteurs dimensionnels à moitié construits etc... La structure interne étant partiellement utilisée par la Vérification sémantique, cela peut provoquer des erreurs dans le fonctionnement d'actions de compilation appelées ultérieurement, travaillant sur cette mauvaise partie. Il faut donc savoir, dans ce cas là, invalider proprement cette dernière. Cela provoquera sans doute des erreurs sémantiques ultérieures "non justifiées", mais c'est normal.

Ce problème de cohérence de structure ne se pose plus si l'on conçoit un vérificateur avec plusieurs passages, le premier étant exclusivement réservé à la vérification de la syntaxe. Les vérifications sémantiques ne sont effectuées que lors du ou des passages ultérieurs (toujours pilotés par un analyseur) à partir de la chaîne syntaxique, codée, correcte, produite par le premier passage. Mais on a vu que cela présentait d'autres inconvénients, comme la perte du texte d'entrée, alors qu'il est bon de disposer de celui-ci tout au long de la vérification, pour le marquage des erreurs.

Nous terminerons ces remarques en faisant état de certaines incompatibilités caractéristiques, rencontrées dans ce travail. Il s'agit de l'opposition entre le fait d'éviter de trop alourdir un langage devant être utilisé par des non informaticiens, peu désireux de faire un effort important d'apprentissage, et l'adoption de formes syntaxiques rigoureuses mais contraignantes dans leur écriture.

Exemple :

La plupart des terminaisons de blocs personnalisées, de la forme "end-machin" dans

```
machin
  .
  .
  .
end-machin
```

ont été considérées comme inacceptables pour l'utilisateur potentiel. Elles ont donc été banalisées et remplacées par le caractère ";". Il est évident que cet allègement de l'écriture, nécessaire d'un certain point de vue, complique d'autant plus la récupération d'erreurs.

III.2.2. STRUCTURES DE DONNÉES : DÉFINITION ET PRINCIPES DE FONCTIONNEMENT

La structure issue du traitement d'un segment CASCADE, de n'importe quel niveau de language, est construite en mémoire, dans un espace de travail unique qui lui est attribué.

Les données élémentaires manipulées dans cette opération sont de deux types :

- "informations", éventuellement précédées de leur longueur si le contexte, ou une connaissance insuffisante de ce dernier, ne permet pas une interprétation correcte ;

- "pointeurs", définis par rapport aux extrémités de l'espace de travail (pointeurs absolus).

Les notions structurelles utilisées, composées de données élémentaires, sont essentiellement des descripteurs, des chaînes codées et des listes.

- Les descripteurs sont des zones de stockage de données. Ces dernières sont en en nombre fixe pour des descripteurs de même type.

- Les chaînes codées sont des suites ininterrompues de données, formées de codes et de paramètres. Elles se terminent par une marque "fin de chaîne".

- Les listes, à chaînage descendant, peuvent contenir une quantité quelconque d'éléments. Un élément de liste est constitué classiquement d'une partie "pointeur" et d'une partie "données élémentaires" quelconque. Le format utilisé pour ces listes est donnée ci-après, figure III.11.

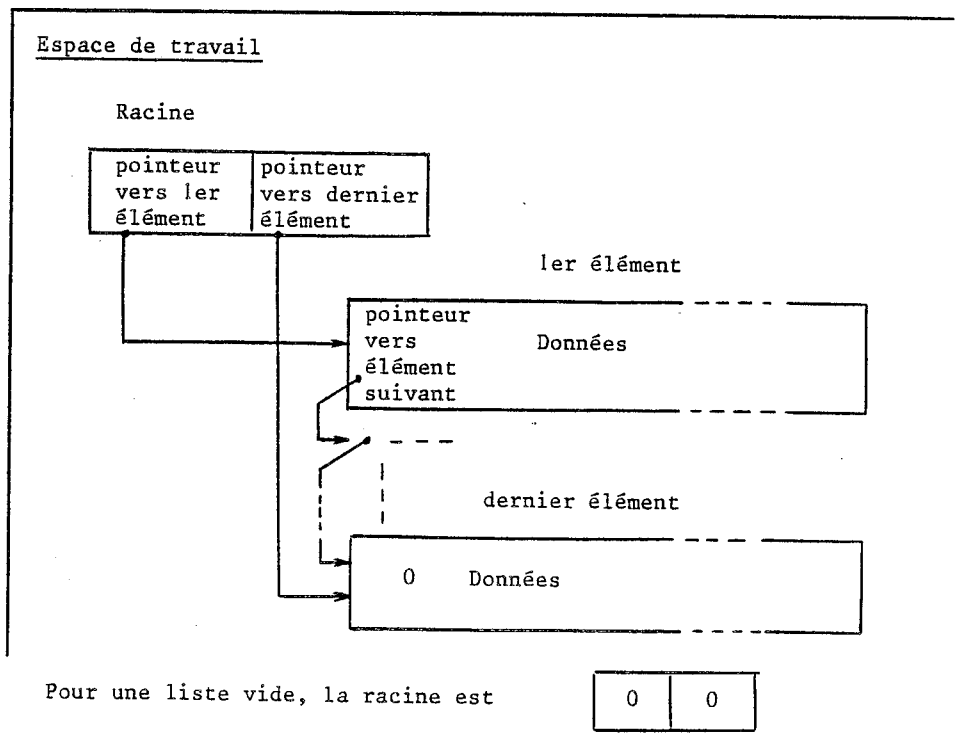


Figure III.11

III.2.2.1. CODAGE DES UNITÉS LEXICALES

L'analyseur lexical, appelé au coup par coup, reconnaît et code les mots et symboles de base du langage. Pour chacune des unités lexicales traitées, il rend des résultats structurés comme ci-après

Code lexical	accès à la chaîne de caractères	longueur 1	longueur 2	longueur 3
1	2	3	4	5

Chaque champ est un entier et son utilisation dépend du type d'unité lexicale considéré.

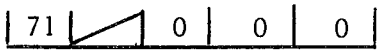
Champ 1 : Le code lexical est celui d'une unité lexicale précise, symbole ou mot-clé, ou celui de la famille de cette dernière dans le cas d'un identificateur ou d'une constante. On trouvera la liste de ces codes en annexe 4.

Champ 2 : Pour un identificateur ou une constante, l'analyseur fournit, en plus du code de famille, le chaîne des caractères constituant l'unité lexicale. Ce champ pointe la chaîne, générée dans l'espace de travail décrit plus loin.

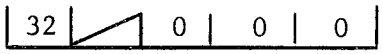
Champs 3, 4 et 5 : Ils sont en liaison avec le champ précédent. La chaîne représentant l'unité lexicale peut être monolithique ou constituée de deux ou trois sous-chaînes. Cete caractéristique est reflétée par la valeur du code lexical. La ou les longueurs, exprimées en nombre de caractères, permettent d'interpréter la chaîne considérée.

Exemples :

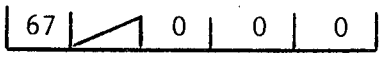
mot-clé description



symbole simple !



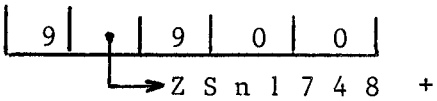
symbole double =<



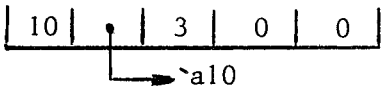
identificateur ROSALIE



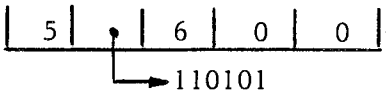
constante littérale 'ZSn1748 +'



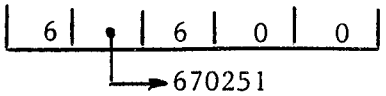
constante logique `a10



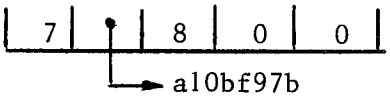
constante entière binaire ##b110101



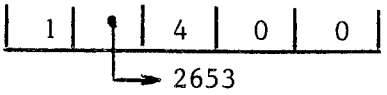
constante entière octale ##o670251



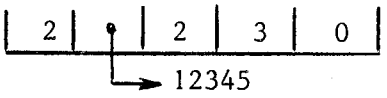
cste. entière hexadécimale ##ha10bf97b



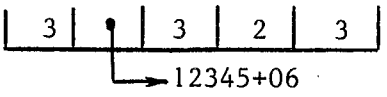
constante décimale 2653



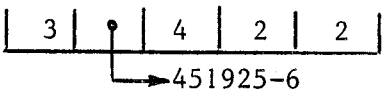
cste. réelle simple 12.345



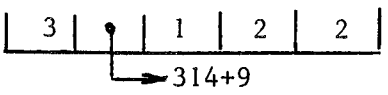
constante réelle avec exposant 123.45E+06



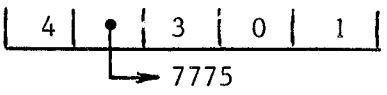
constantes réelles avec facteur d'échelle
4519.25MC (micro)



3.14G (giga)



constante 777E5



III.2.2.2. TRAITEMENT D'UNE DESCRIPTION, STRUCTURE INTERNE ET CODAGE

III.2.2.2.1. ORGANISATION GÉNÉRALE DE L'ESPACE DE TRAVAIL

Les différentes notions du langage CASCADE se traduisent structurellement en :

- un descripteur principal de la description, racine de la structure,
- un ensemble de chaînes de caractères, entassées dans l'espace de travail par la lexicographie, avec leur système d'accès rapide,
- des listes constituant le répertoire des objets et segments déclarés, avec leur structure de définition,
- des listes d'assertions,
- une chaîne codée unique, traduisant l'ensemble des instructions de la partie "relations" de la description.

La structure est construite en séquence, à partir du début de l'espace de travail (espace liste), sauf la chaîne codée des instructions qui, elle, est générée à la fin de celui-ci, en sens inverse (voir figure III.12).

Cette utilisation en double sens d'une zone répond à un souci d'optimisation de l'occupation mémoire.

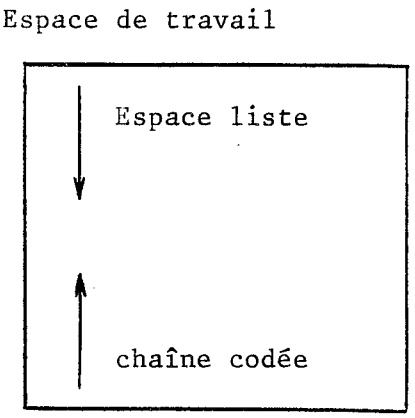


Figure III.12

Aucun mécanisme de retassement ou de gestion de liste libre n'est mis en oeuvre ici, car la quantité de "trous" présents dans la structure générée finale est négligeable.

Lorsque la vérification d'une description est terminée, on vide la partie utilisée de l'espace de travail dans le fichier de sortie. Pour cela, on recopie dans ce dernier d'abord l'espace liste, puis la chaîne codée en la retournant (voir figure III.13).

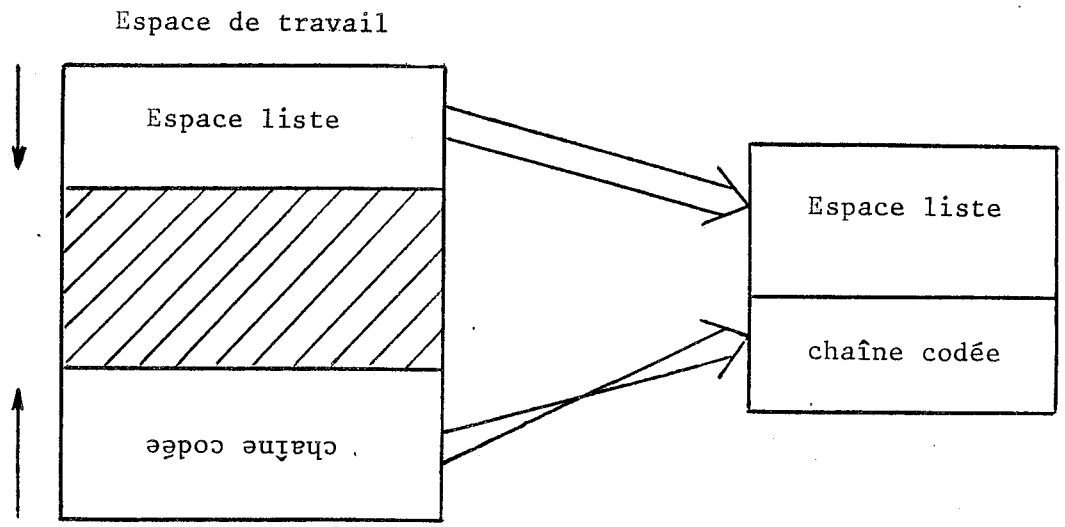


Figure III.13

Examinons maintenant le cas de descriptions, fonctions ou procédures, définies internes à une description. Leur traitement peut s'effectuer suivant le principe d'une double pile. Cette technique se prête très bien à la sémantique de l'imbrication des segments, notamment sur la question de la portée des noms des objets.

Un exemple, avec deux descriptions imbriquées, est donnée figure III.14.

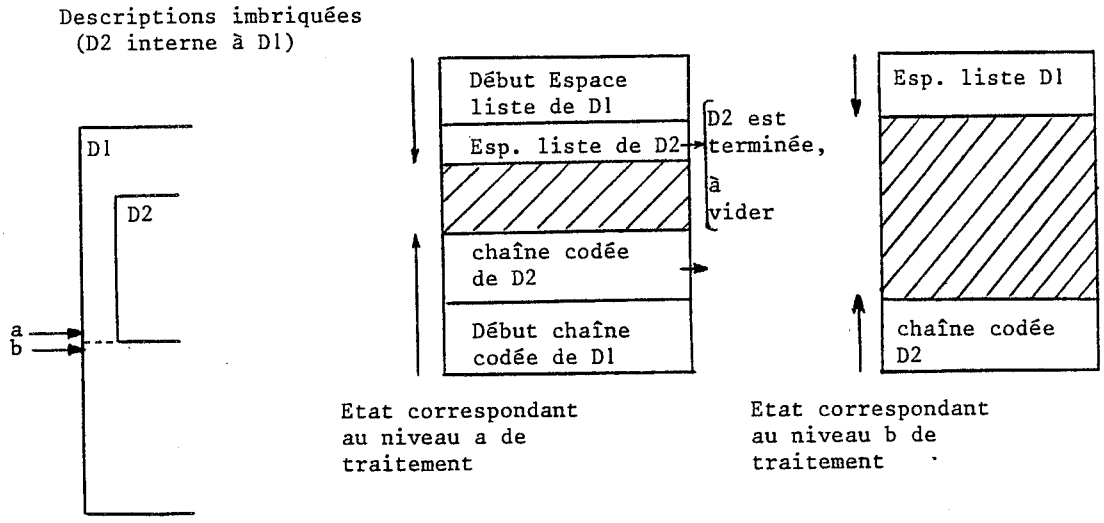


figure III.14

L'espace de travail détermine, par sa taille, la capacité du compilateur pour ce passage. Cette capacité est aussi liée à la taille des piles utilisées.

III.2.2.2.2. ESPACE LISTE

Le début de l'espace liste est réservé au descripteur principal de la description (voir figure III.15), qui contient les racines de toutes les listes. C'est par lui qu'on accède à ces dernières. On y trouve aussi un certain nombre d'informations générales relatives à la description traitée : indicateurs, compteurs, etc. et en particulier la table des clés relative à l'adressage dispersé, permettant d'accélérer l'accès aux chaînes de caractères entassées.

Un exemple d'ensemble d'éléments contenus dans le descripteur principal est donnée en annexe 5.

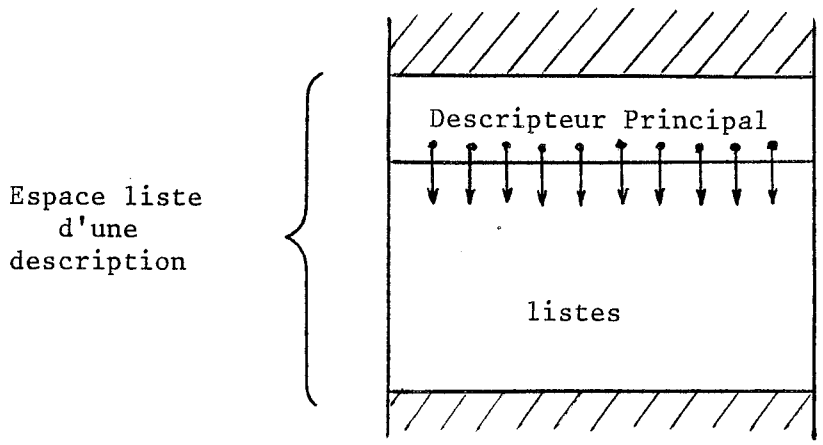


figure III.15

Afin de traite convenablement les règles de portée de noms, il faut, tout au long de la vérification, être capable pour tout point du texte, de représenter le contexte courant.

Il faut donc :

- connaître les régions accessibles depuis le point courant du texte,

- d'autre part, connaître l'ordre dans lequel il faut explorer ces régions (de la plus interne vers la plus externe en général).

Une fois ceci déterminé, pour l'imbrication des définitions de descriptions, par exemple, nous procèderons de la façon suivante :

- A la rencontre de toute nouvelle description définie interne, on initialise l'espace de travail avec une nouvelle base et un nouveau descripteur principal, dont le premier élément contient l'ancienne base. Toute description de ce type pointe donc sur celle qui l'englobe directement. Ce chaînage remontant permet tout l'acces désiré aux différentes régions (voir figure III.16).

- Lorsqu'on sort d'une description on restaure le pointeur de l'espace liste courant avec la base de la description englobante

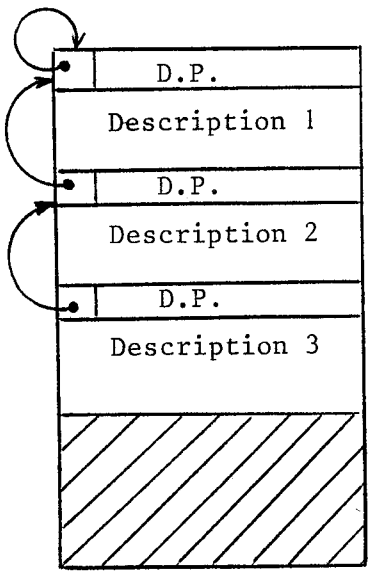


figure III.16

Les identificateurs et les constantes littérales, numériques et logiques, contrairement aux symboles et aux mots clés, ne peuvent pas être représentés uniquement par leur code de famille. Il faut donc garder dans la structure leur représentation externe sous forme de chaines de caractères. Ces dernières sont entassées dans l'espace liste, au fur et à mesure de leur arrivée. Une même chaine n'est stockée qu'une seule fois.

La figure III.17 montre l'organisation de la structure de stockage.

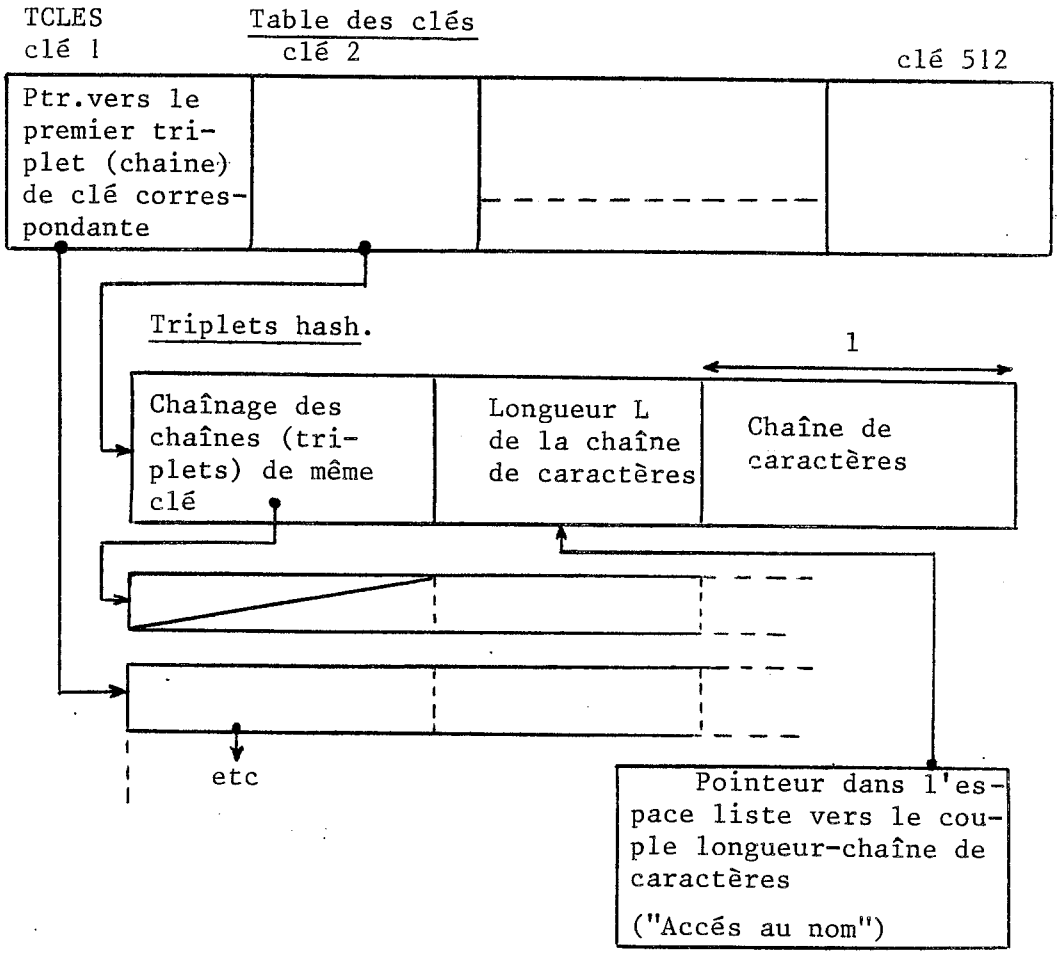


Figure III.17

L'opération de compilation nécessite de disposer, au moment du traitement, de tables dites tables de symboles, rassemblant toutes les informations sur les différents objets utilisés dans la description traitée. On collecte donc dans la structure appropriée qu'est l'espace liste, toutes les définitions des mots afin de connaître parfaitement le sens de chacun d'eux.

D'une manière générale, les données sont représentées au fur et à mesure de leur apparition dans le texte, sous une forme arborescente caractérisée par l'accès aux différentes représentations.

La figure III.18 montre la structure de stockage des "variables" du langage et de leurs types. Pour certains de ces derniers, le codage est trivial - types de valeurs prédéfinis par exemple -, pour d'autres, tels que les types composés, il est beaucoup plus complexe.

La figure III.19 traduit la référence aux descriptions, définies de manière interne ou externe par rapport à la description courante, l'ensemble des éventuels paramétrages par attributs effectifs de ces descriptions dans l'ordre use, et enfin l'ensemble des unités générées par ces dernières avec leurs éventuelles connexions permanentes.

Il est nécessaire, pour générer cette structure et pour effectuer tous les contrôles portant sur les connexions de ces modules dans la descriptions courante, d'avoir accès à l'interface, déjà compilé et stocké dans le catalogue, de chacune de ces descriptions référencées.

Pour cela, on recopie dans une zone de travail INTF, distincte de l'espace de travail général, les interfaces des descriptions spécifiées dans l'ordre use. Ils y restent tout au long du traitement de la description courante. On voit donc, plus précisément que l'espace INTF se comporte comme une pile, tout comme l'espace général, à cause de la notion de description interne.

Pour une description traitée, les interfaces appelés sont chaînés dans la sous-zone de INTF concernée.

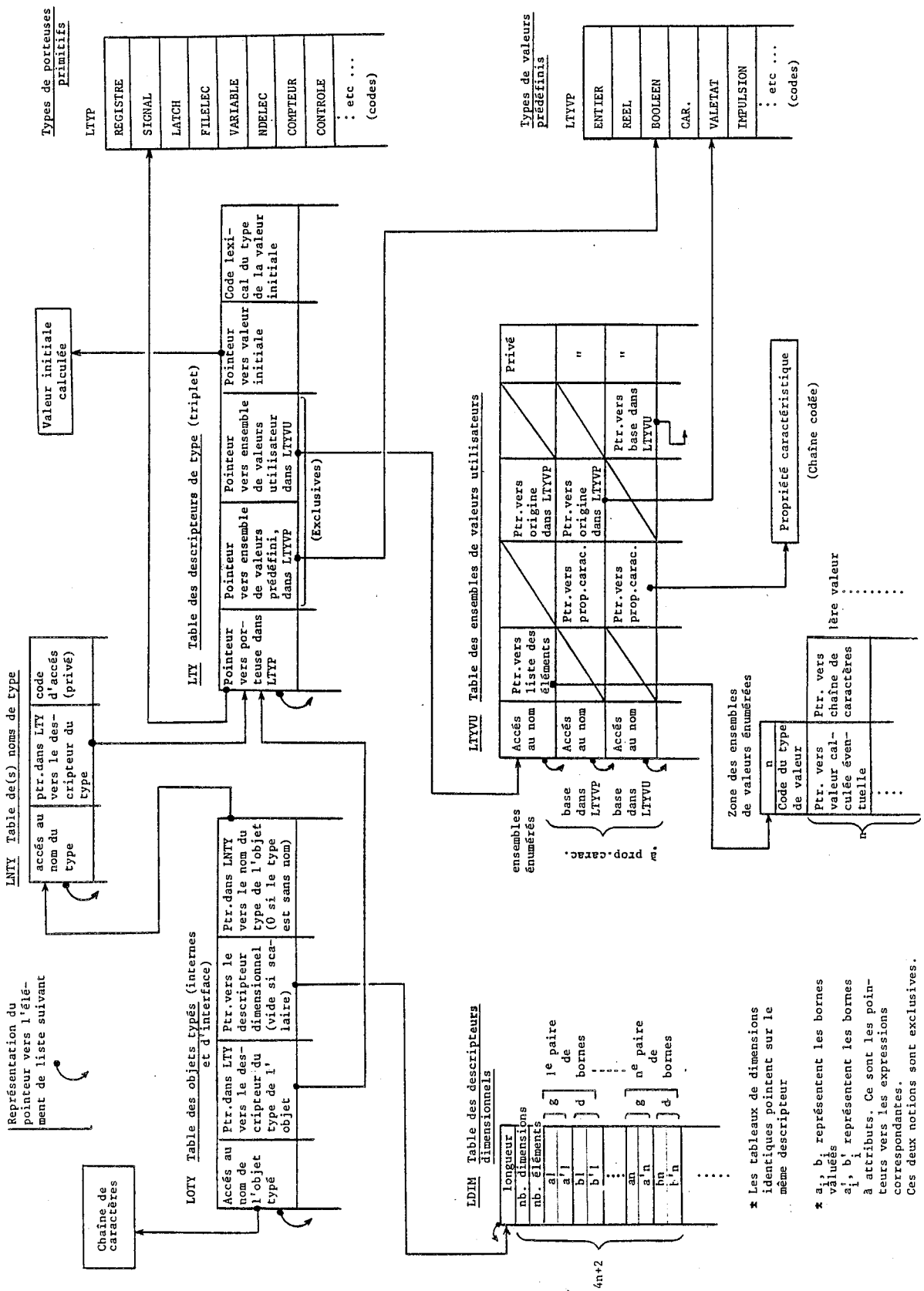


figure III.18

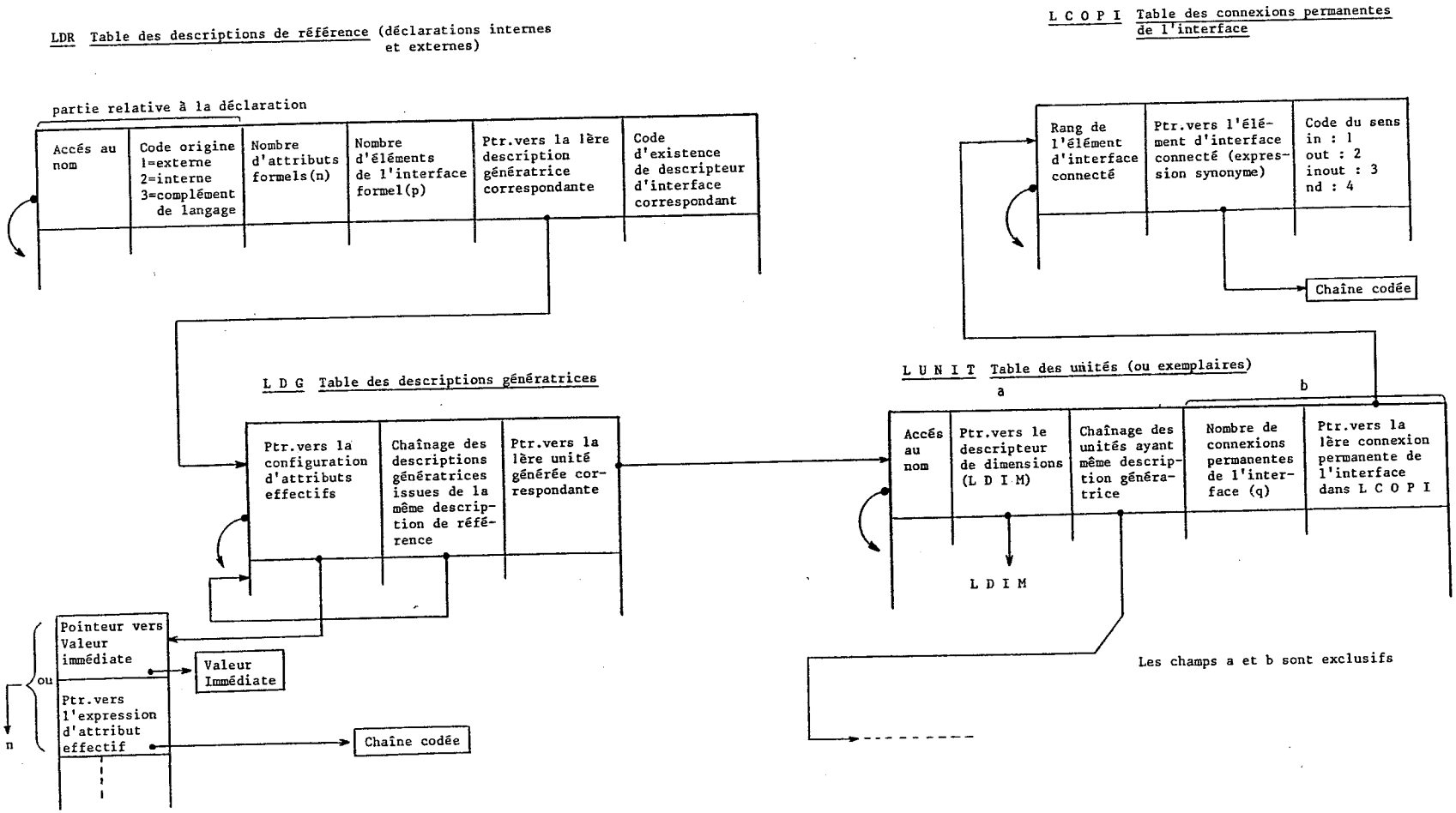


figure III.19

On trouvera, figure III.20, la structure résultant de la prise en compte des expressions de synonymie utilisées dans les déclarations internes de porteuses.

Exemple

```
Declare
  ......   A[1:8], C ...   B[0:8] = A\C,
                           D[1:4] = B[2:5],
                           E=C,    F=-3;
```

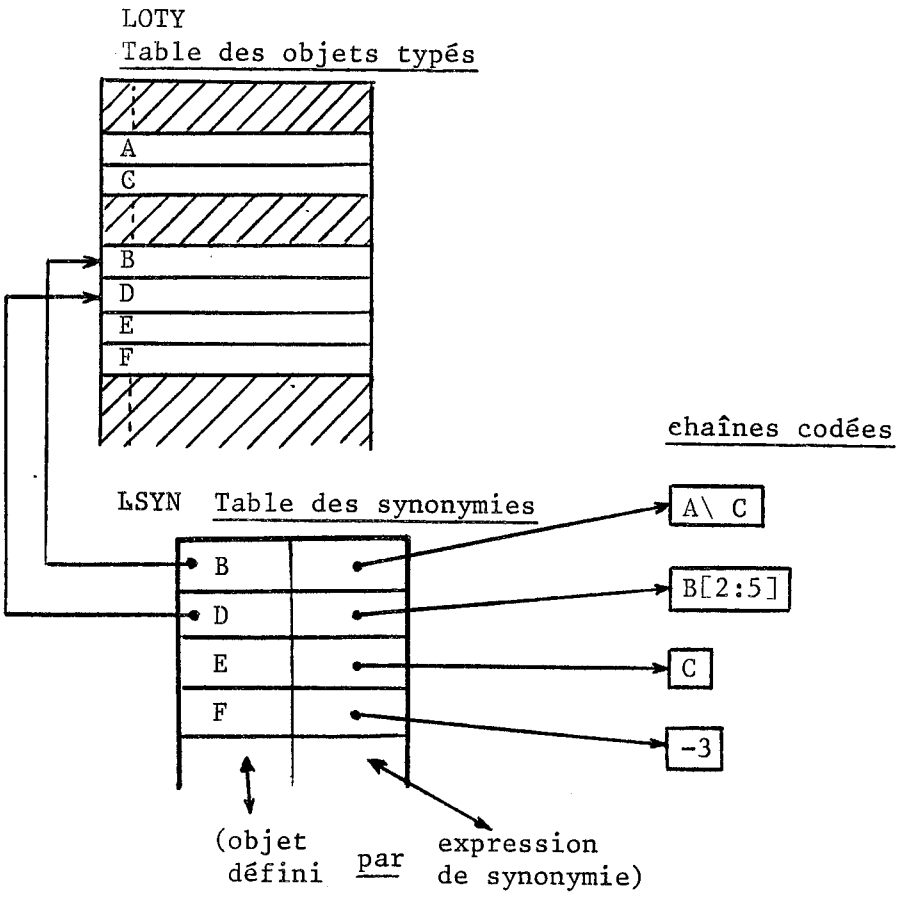


Figure III.20

La figure III.21 exprime le résultat du traitement des constantes. Toute déclaration d'objet de cette sorte donne lieu à la création d'un descripteur, élément de la liste LCTE, contenant l'accès au nom de l'objet, la définition de son type et celle de sa valeur.

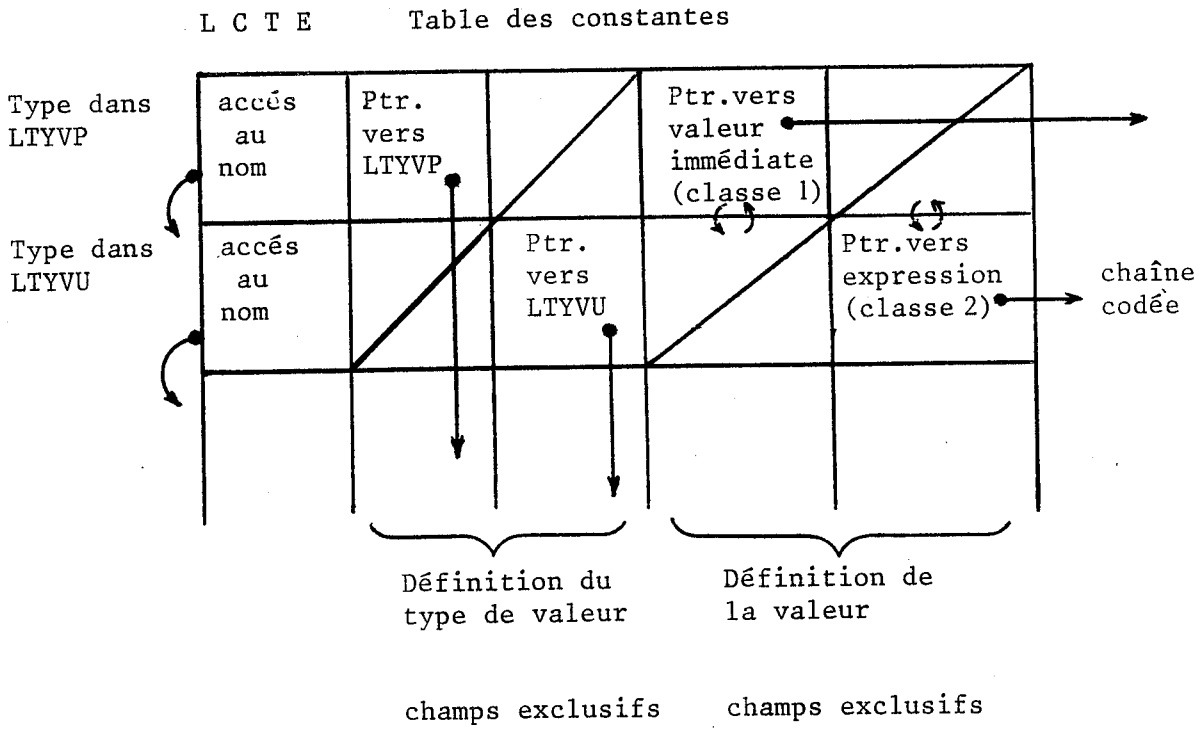


Figure III.21

Les attributs formels, c'est-à-dire les déclarations d'attributs, sont stockés dans la liste LATT (voir figure III.22). Un élément de cette liste est un descripteur d'attribut constitué d'un pointeur vers le nom et d'un pointeur vers le type de valeur déclarateur (prédéfini ou utilisateur, les champs correspondants sont exclusifs). Si l'on désire généraliser cette notion aux attributs tableaux, avec valeurs par défaut par exemple, il faudrait introduire des champs supplémentaires, pointant vers un descripteur dimensionnel et vers les valeurs par défaut.

Les éléments d'interface, tous stockés dans LOTY comme toute porteuse, sont chaînés dans une liste LINTF, dans leur ordre d'apparition dans l'interface et avec leur sens (voir figure III.23).

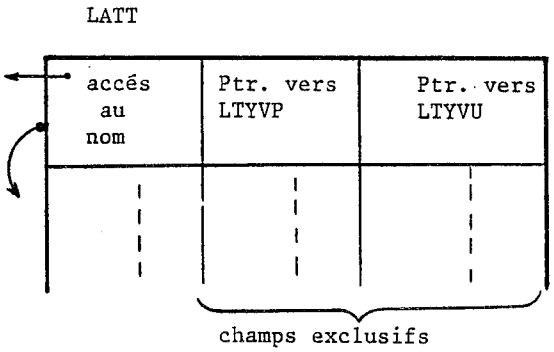


Figure III.22

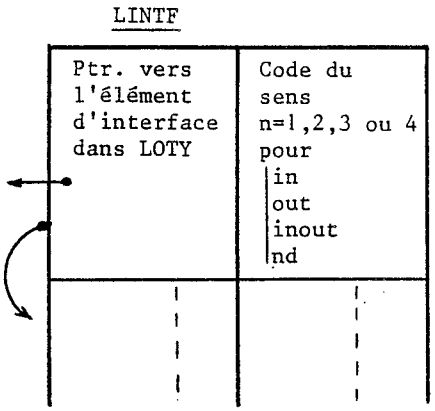


Figure III.23

Nous terminerons cet exposé des principaux éléments de l'espace liste d'une description par les listes des assertions d'interface et de corps (figure III.24).

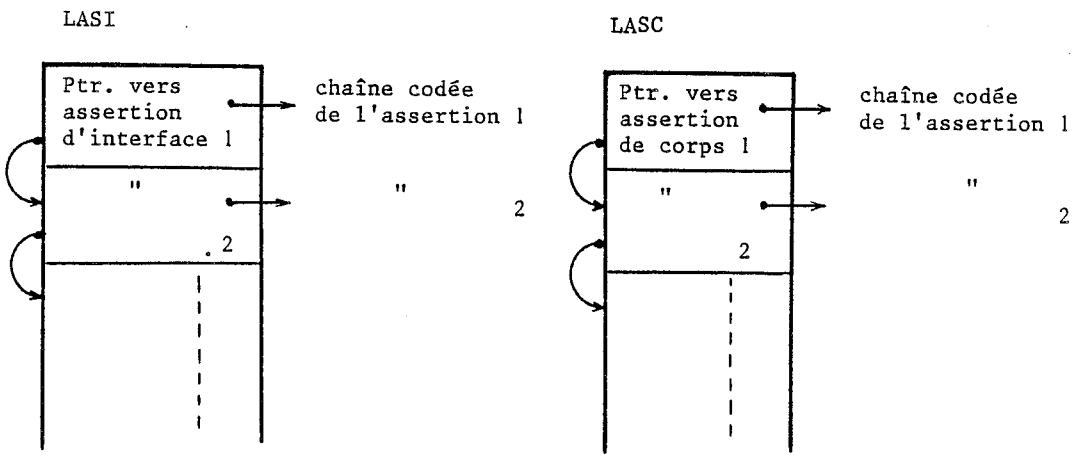


Figure III.24

III.2.2.2.3. CHAÎNE CODÉE

Les chaînes codées sont des structures internes traduisant le texte d'entrée correspondant sous forme continue condensée. Ce qui n'est pas le cas avec la structuration en descripteurs et en listes par exemple. Ces structures abstraites, ininterrompues, peuvent être aisément reprises par des passages de compilation ultérieurs, à l'aide d'un analyseur syntaxique, pourvu qu'elles présentent une structure syntaxique.

Les chaînes codées rencontrées dans la compilation d'une description CASCADE correspondent à :

- des propriétés caractéristiques (voir LTYVU),
- des expressions synonymes, dans les déclarations de constantes ou de porteuses (voir LCTE et LSYN),
- des expressions "attributs effectifs" (voir LDG),
- des expressions bornes de dimensions (voir LDIM),
- des assertions (voir LASI et LASC),
- la partie "relations" contenant les instructions de fonctionnement.

Pour cette dernière, étant donné que lors de la vérification le traitement des instructions génère parfois des éléments de structure, nous voyons qu'il est indispensable, pour éviter des ruptures, de stocker la chaîne codée des instructions dans une zone distincte de l'espace liste. D'où la solution adoptée, précédemment évoquée, de générer la chaine codée en remontant du fond de l'espace de travail associé à la description. Les autres chaînes codées, par contre, peuvent être générées sans inconvénients dans l'espace liste.

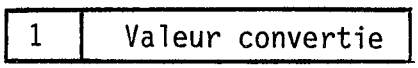
Notons que pour le type de traitement que subissent ultérieurement ces chaînes, il n'est pas intéressant de disposer de leur longueur. Leur fin est marquée par un code spécial (13) détecté en tant que tel par l'analyseur.

Le codage conserve presque entièrement la structure syntaxique du langage source.

- Les mots clés, symboles, séparateurs et opérateurs sont codés par le code que leur a affecté la lexicographie.
- Les identificateurs sont remplacés par le code de la liste qui les contient (LOTY, LATT, LCTE, LUNIT, LFI, LPI, LTYVU, LNTY, etc...) suivi de leur numéro d'ordre dans cette liste. Nous avons préféré ce mode de repérage au classique pointeur car la chaîne codée devient comme cela moins dépendante de l'espace liste.
- Les constantes sont codées par leur code lexicographique suivi :
  * de la constante elle-même si la valeur est immédiate pour la représentation choisie,
  * du pointeur vers la constante en question dans le cas contraire (pointeur vers le tas des chaînes de caractères, par exemple, si c'est une constante littérale).

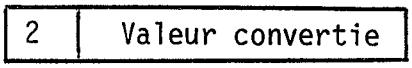
Exemple :

- Codage d'un entier :



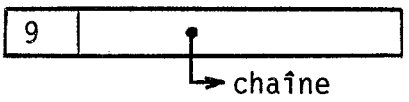
On ne tient pas compte de la forme externe de l'entier (décimale, hexadécimale, binaire ou octale).

- Codage d'un réel :



et ceci quelle que soit la forme du nombre réel.

- Codage d'un littéral :

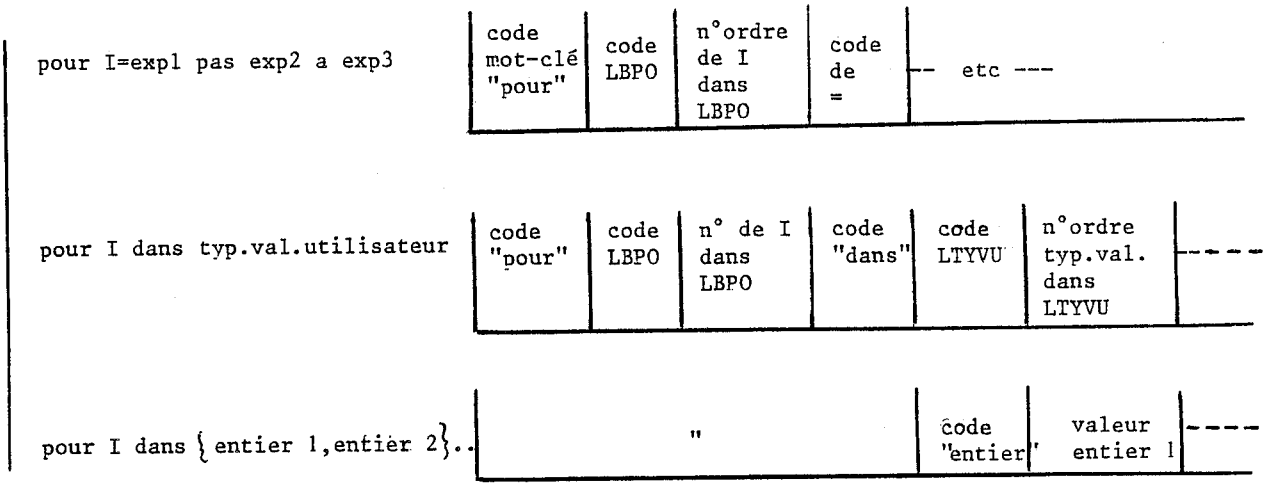


Certaines instructions de contrôle du langage donnent lieu à la création, en plus d'une portion de chaîne codée, d'un morceau de structure dans l'espace liste. Nous allons illustrer ce fait avec les boucles "pour" et les instructions conditionnelles "if", "case" et "chargement sous horloge".

Le traitement des boucles "pour" nécessite la création de deux listes supplémentaires :

- LBPO : liste des indices de boucles pour (tous les indices),
- LBPA : liste des indices actifs de boucles pour.

Voyons maintenant le codage des différentes formes de boucles pour.



Pour les instructions conditionnelles, les conditions sont répertoriées dans une liste LIFCAS, dont la structure est donnée figure III.25

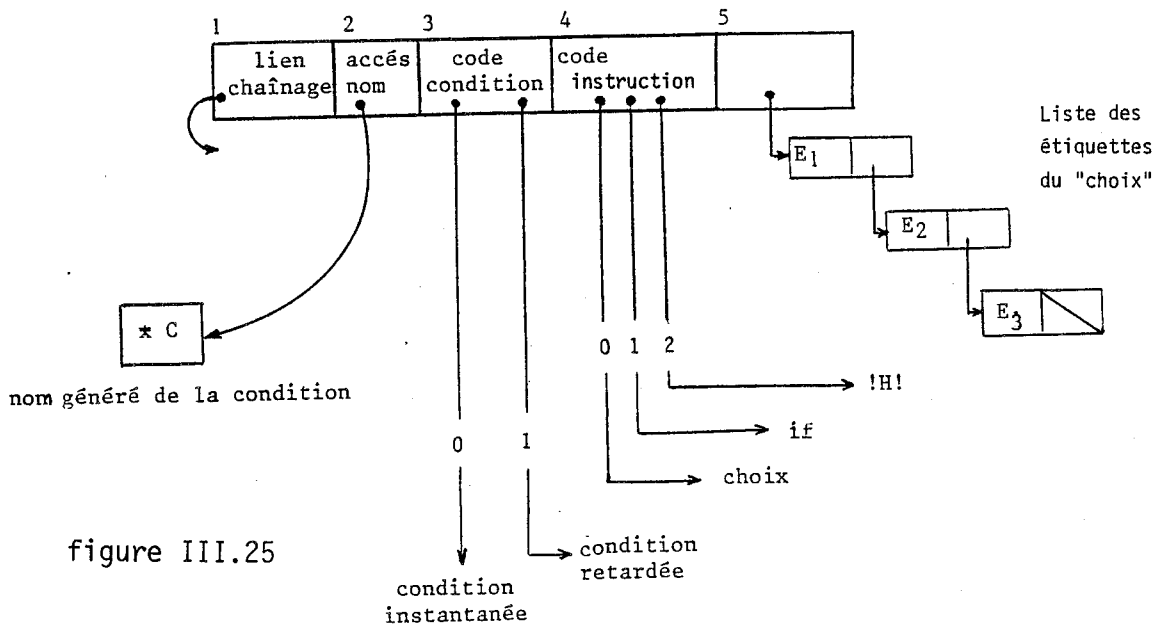
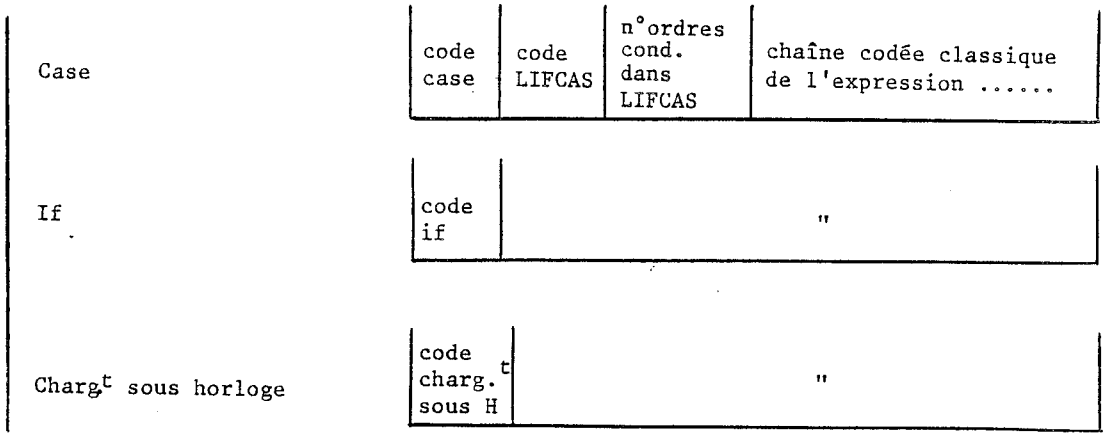


figure III.25

Le codage de ces instructions est le suivant :



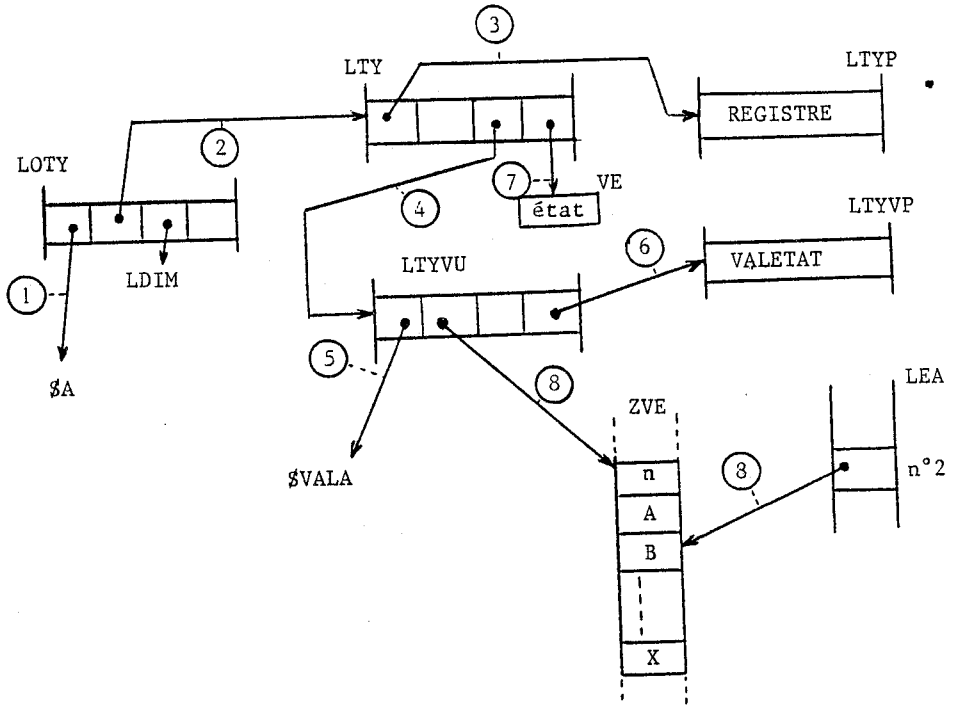
III.2.2.2.4. CAS DES SPÉCIFICITÉS DES NIVEAUX DE LANGAGE

La structure exposée dans les paragraphes précédents est générale et forme un noyau commun pour tous ou plusieurs niveaux de langage. Certaines notions, bien spécifiques à certains niveaux, donnent toutefois lieu à génération d'éléments de structure supplémentaires. Nous en donnons pour exemple, ici, les automates de contrôle des niveaux CASSANDRE et LASCAR, exposés en annexe 2.

Une liste, LEA, des états de l'automate traité, est constituée, et l'automate est codé de la manière la plus naturelle (voir figure III.26).

Les fonctions et procédures écrites en langage CASCADE, non implémentées, pourraient subir un traitement s'inspirant fortement de celui des descriptions.

Espace liste



Construction de la structure associée à un automate : ① LOTY , ② LTY , ③ LTYVU , ④ LEA

Ⓘ Création d'une entrée dans chacune des tables

Ⓘ Ⓘ Puis chaînage selon l'ordre 1, ⟶ , 8

Chaîne codée

code de l'automate 17	code de LOTY	n°d'ordre de $A dans LOTY	code de LEA	n°ordre étiquette dans LEA	chaîne codée de la liste d'instructions suivant l'étiquette	code de LEA	n°ordre étiquette dans LEA	------

Donne le pointeur sur ZVE pour la 1ère étiquette

2ème étiquette ..

Figure III.26

III.2.2.3. STRUCTURE ET FONCTIONNEMENT DES DESCRIPTEURS D'INTERFACE

Le descripteur d'interface d'une description en cours de vérification est généré dès la fin du traitement de la partie en-tête de cette description. Il consiste simplement en la recopie de tous les renseignements stockés dansla structure lors de l'analyse de l'en-tête (listes LOTY, LATT, etc...).

D'où la structure générale d'un descripteur d'interface (d.i.), donnée figure III.27. (Nous n'avons pas détaillé la définition des types, descripteurs dimensionnels...).

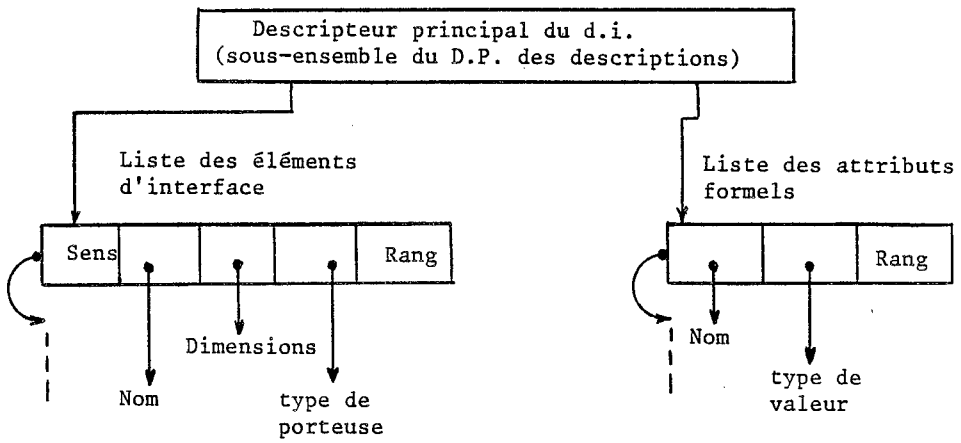


figure III.27

La rencontre de l'utilisation d'une description dans un ordre use provoque la recherche et le chargement de son d.i. associé. On peut voir figure III.28, une structure possible, permettant la gestion de l'espace d'accueil des d.i.. Ces derniers restent accessibles durant toute l'analyse des zones correspondant à la porté des ordres use en question.

Les éléments présents dans un d.i. permettent d'effectuer toutes les vérifications nécessaires – dimensionnelles, de sens et de types – des connexions des modules générés par la description.

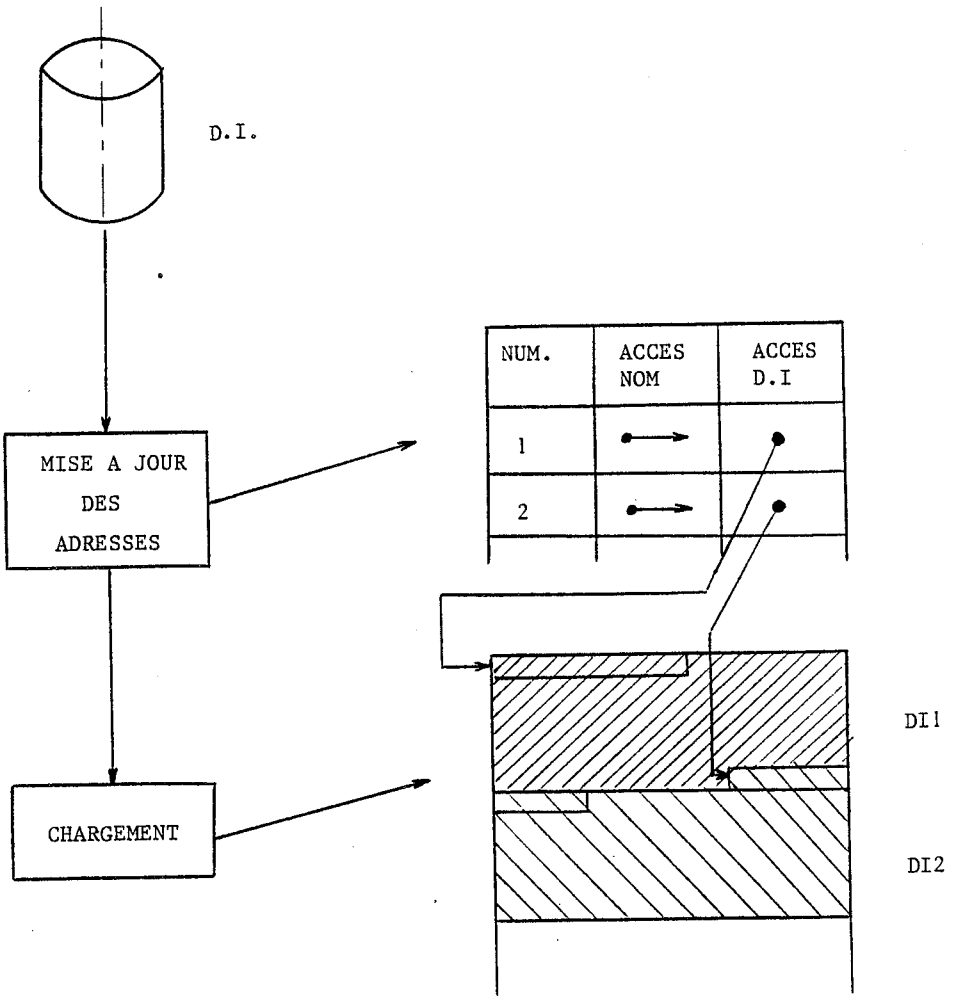


figure III.28

III.2.2.4. STRUCTURE ET FONCTIONNEMENT DES COMPLÉMENTS DE LANGAGE

Les compléments de langage (c.ℓ.) sont vérifiés dans l'espace de travail unique, tout comme les segments descriptions, fonctions et procédures déjà vus. Le mécanisme de pile de cet espace est aussi particulièrement adapté à la prise en compte des descriptions, fonctions et procédures définies dans les c.ℓ.

Compiler un c.ℓ. c'est, en plus de le vérifier, générer une struture décrivant son contenu, c'est-à-dire constituée de définitions de types, de noms de descriptions, fonctions et procédures. Ces trois derniers segments sont traités, dans les c.ℓ., par le mécanisme général compilant les segments définis internes. A la fin de l'analyse du c.ℓ., lorsqu'on a dépilé les segments internes, il reste la structure du c.ℓ., au début de l'espace de travail. Cette dernière, brièvement exposée dans ses principes figure III.29, est ensuite vidée dans le sous-catalogue des c.ℓ., pour exploitation ultérieure.

Descripteur principal d'un c.ℓ.

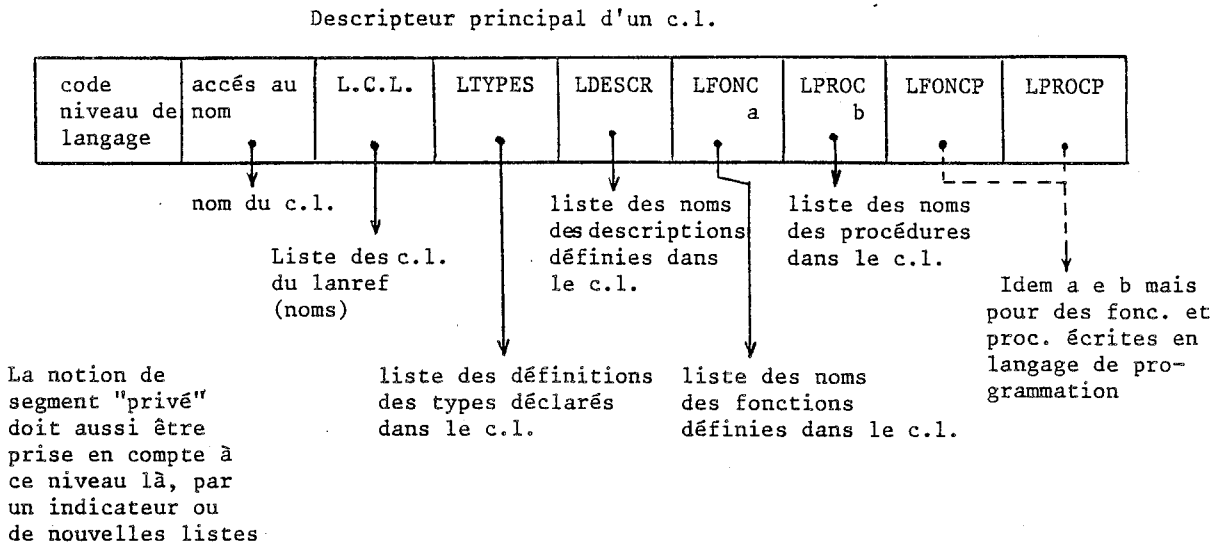


figure III.29

L'exploitation d'un c.ℓ. est déclenchée par la détection de son nom dans un ordre lanref. Le c.ℓ. est recherché et chargé dans un espace prévu à cet effet (COL), où il devient une extension de la zone des définitions de segments et types, pour tous les segments, y compris les c.ℓ., compilés sous la portée de cet ordre lanref. L'espace COL contient toujours, au moins, le c.ℓ. système du niveau de langage considéré.

L'analyse de l'ordre lanref donne lieu à la création d'une partie résidant en permanence dans l'espace d'exploitation durant la compilation de tous les segments ou c.ℓ. sous la portée de ce lanref. La figure III.30 illustre le traitement général d'un modèle avec utilisation de c.ℓ. . L'utilisation de c.ℓ. pour la compilation de c.ℓ. est tout à fait analogue.

Notons enfin que l'utilisation, à partir d'un segment, d'un type défini dans un c.ℓ., provoque la recopie pure et simple de la structure interne du type dans ce segment, comme s'il était explicitement déclaré dans celui-ci. Ceci garantit l'autonomie ultérieure des segments.

L'utilisation d'un segment description, fonction ou procédure, défini dans un c.ℓ., n'est pas particulière, et revient en fait à utiliser son descripteur d'en-tête ou d'interface correspondant, créé tout normalement lors de l'analyse de ces segments

III.2.2.5. STRUCTURE DU FICHIER DE SORTIE DU VÉRIFICATEUR

Il s'agit du fichier intermédiaire en sortie du passage 1.

Ce fichier standard, qui sert d'entrée au catalogueur, contient les segments descriptions, fonctions et procédures qui viennent d'être vérifiés. Il est constitué d'enregistrements de longueur variable, à

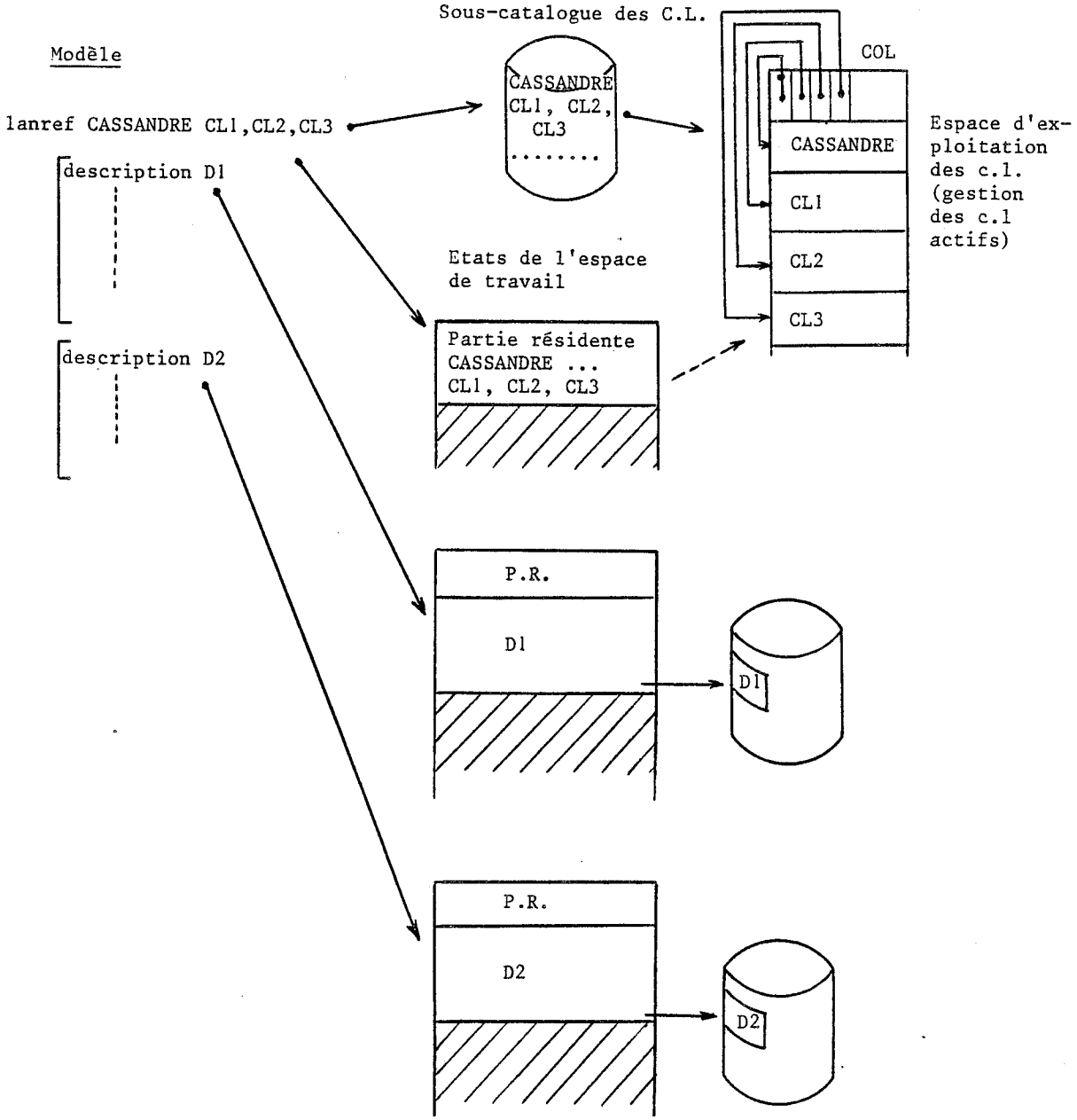


Figure III.30

raison de trois par segment. L'organisation d'un segment dans ce fichier est donnée par la figure III.31.

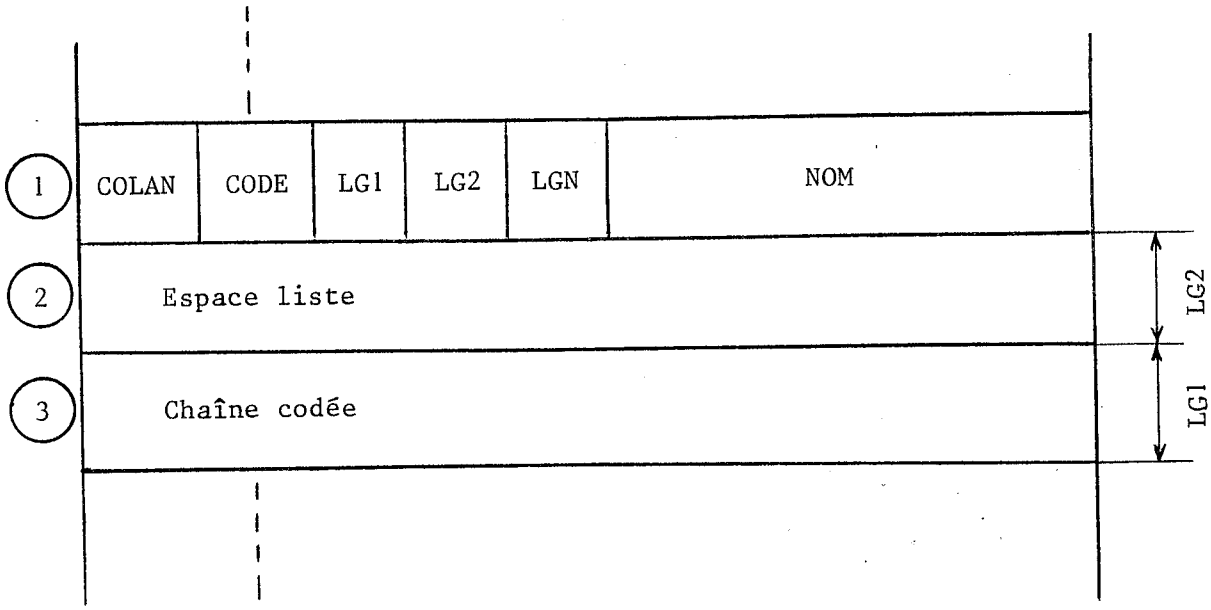


figure III.31

COLAN : Code du niveau de langage
CODE : Type du segment (D,F ou P pour description, fonction ou procédure)
LG1 : Volume de l'espace liste
LG2 : Volume de la chaîne codée
LGN : Longueur du nom
NOM : Nom complet de la description (identificateur de la description, postfixé avec les identificateurs de tous ses ascendants

- L'enregistrement (1) est un enregistrement de contrôle
- Le (2) contient l'espace liste généré à partir du segment traité
- Le (3) contient la chaîne codée correspondante.

B

SUPERVISEUR DE SIMULATION

Au sortir de la chaîne de traitement $\phi 1$ - $\phi 2$, on dispose d'un ensemble de données qui, une fois chargées en mémoire, constituent un modèle virtuel du système décrit. Pour faire fonctionner ce modèle, il faut lui adjoindreun séquenceur dont le rôle principal est de gérer le temps et de faire ainsi progresser le système.

Pour prendre en compte les actions de l'utilisateur, il faut également avoir recours à un interface entre ce dernier et le modèle. Cet interface est un superviseur interactif, en liaison avec le séquenceur pour conduire la simulation, et avec la SOS pour permettre l'accès à tout objet du modèle sous sa forme externe.

Rappelons la structure du modèle et les fonctions de ses principaux composants. A la sortie de $\phi 2$, on trouve :

- Le modèle proprement dit, sous forme d'un arbre interprétable ("interne") appelé Structure De Simulation (SDS), et de données permettant de construire la zone valeur des différents objets du modèle. Les feuilles de l'arbre interne sont les Blocs de Simulation compilés, exécutables.
- La Structure Orientée Simulation (SOS), déjà évoquée ; elle est constituée des descriptions valuées, c'est-à-dire des modules avec leur contenu, accessibles à travers l'arbre d'imbrication de ces modules (arbre "externe"). Il s'agit en fait de ce que l'on appelle en compilation la "table des symboliques" (répertoire des descripteurs, convenablement organisé, contenant notamment les noms donnés par l'utilisateur aux diférents objets). La figure IV.1 résume ces notions.

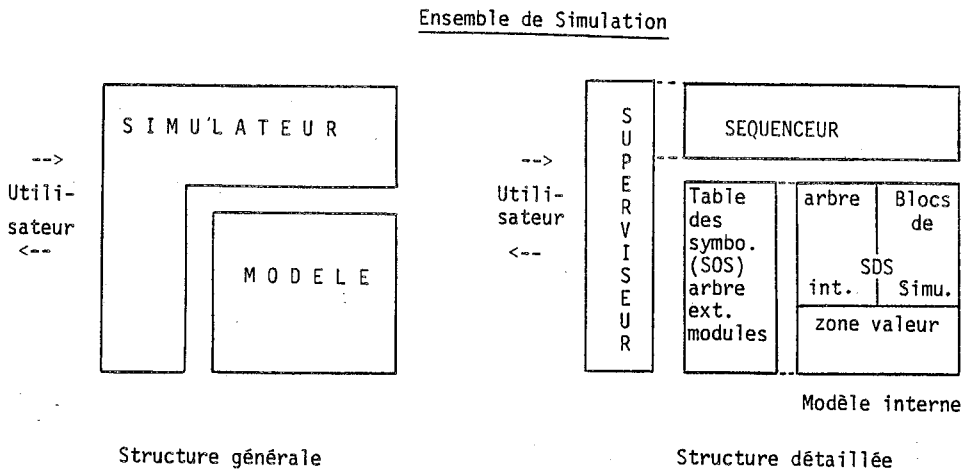


figure IV.1

Dans cet ensemble, le rôle du séquenceur (Bre.85) est d'assurer la succession des actions constituant le processus de simulation. En première approximation, ce pilotage agit dans deux directions :

- A un instant t de calcul donné, le séquenceur interprète l'arbre de la SDS (arbre interne), et lance l'exécution des blocs de simulation compilés. C'est lui qui prend en compte, entre autres, la notion d'activité/inactivité attachée à chacun des blocs.
- A un niveau supérieur, le séquenceur a pour fonction de faire progresser le temps et de lancer l'opération précédente autant de fois que nécessaire, de la date courante t_c à une date t_s fournie par l'utilisateur (exécution de cycles, etc...).

Le Superviseur, développé dans le cadre de ce mémoire (Bre.83b) s'inscrit dans le cadre de l'Environnement de Simulation CERES (BMP.84, BaP.83b) et du Langage de Commande général de Simulation du même système (BaP.83a). Il représente un noyau minimal indispensable pour faire fonctionner la simulation CASCADE, et est constitué d'un jeu de commandes élémentaires et d'un ensemble de fonctions de vérification et d'exécution de ces commandes. En attente de la réalisation définitive du langage de commande, nous avons surtout porté notre effort sur la définition et la réalisation des fonctions, de manière à ce qu'elles soient aussi indépendantes que possible de leur environnement d'utilisation. Ceci afin de pouvoir les réutiliser le moment venu avec le minimum de modifications.

Au cours de l'exposé du jeu de commandes, nous évoquerons souvent, dans ce document, des possibilités d'adjonction de nouvelles commandes ou d'augmentation de la complexité de celles existant déjà. De même, ce superviseur reste ouvert à la prise en compte de commandes "système" d'espionnage du fonctionnement du simulateur et du modèle, présentant un intérêt pour la conception et la mise au point de CASCADE.

Enfin, ce superviseur, grâce à l'utilisation d'une SOS compatible, devrait fonctionner sans problèmes particuliers avec les simulateurs concurrents développés pour les niveaux portes logiques et interrupteurs (BLR.84).

CHAPITRE IV

RÉALISATION DU SUPERVISEUR DE SIMULATION CASCADE

IV.1. DEFINITION DU JEU DE COMMANDES

La liste qui suit donne une première idée de la façon dont sont perçus les modèles CASCADE durant la phase de simulation. L'utilisateur dispose des commandes générales suivantes :

BOX Pour se positionner sur le module de son choix, dans la hiérarchie du modèle.

STO (Store) - Pour stocker des valeurs dans les porteuses.

PRI (Print) - Pour imprimer des valeurs d'objets au terminal.

BTU (Base Time Unit) - Pour préciser la correspondance entre l'échelle de temps logique et l'échelle de temps utilisée aux niveaux électriques.

RUN Pour demander l'exécution de la simulation (évolution du modèle dans le temps), jusqu'à une date donnée.

HEL (Help) - Pour imprimer la liste des commandes permises au terminal.

TRA (Trajectory) - Pour stocker dans un fichier les valeurs successives prises par des porteuses, durant une simulation.

ETR (End Trajectory) - Pour stopper le stockage défini par la commande précédente.

WAB (What box ?) - Pour imprimer le nom du module courant dans la hiérarchie du modèle.

STP Pour demander l'exécution de la simulation pendant une durée donnée.

? Pour demander l'impression du temps courant.

END Pour sortir du superviseur, une fois la simulation terminée.

Un certain nombre de commandes sont spécifiques à la simulation électrique. Il s'agit de :

DC Pour le choix de la méthode d'initialisation.

HMIN et HMAX Pour spécifier les pas d'intégration minimum et maximum.

ERMAX Pour spécifier l'erreur maximum tolérée.

INIT Pour les valeurs d'initialisation par défaut du système algébro-différentiel.

TR Pour le choix de la méthode de calcul en régime transitoire.

Enfin, un ensemble de commandes système, en principes inconnues de l'utilisateur, a été mise en oeuvre :

TIM et EOT Pour mesurer la consommation en temps machine des cycles de calcul.

DGE, EDG, DTE et ETE Pour les dumps en simulation électrique.

IV. 2. STRUCTURE ET FONCTIONNEMENT DU SUPERVISEUR

Du point de vue organisation hiérarchique, le superviseur est le programme de plus haut niveau du simulateur ; c'est la couche en contact avec le niveau appelant (VAX pour le moment, langage de commande CASCADE plus tard). Il est activé par la commande SIMU.

Dans le prototype actuel, le superviseur et la partie compilée du modèle sont chargés ensemble en mémoire, après édition de liens. On ne peut donc simuler qu'un seul modèle pour un chargement du simulateur.

Le superviseur traite une à une les commandes émises par l'utilisateur. De toutes celles-ci, seules la commande RUN, et STP sa dérivée, donnent le contrôle au séquenceur. Elles seules ont pour effet de lancer l'exécution du modèle. Ainsi il apparait bien que toutes les commandes définies ici, sont prises en compte par le superviseur entre les périodes d'évolution du modèle, en des instants stables du système, ou pauses. Ceci, bien sûr, n'est pas contradictoire avec le fait que des actions définies par commande soient exécutées en différé, durant les périodes d'évolution, comme avec la commande TRA par exemple.

La structure logique du superviseur se présente comme une boucle où l'on récupère à chaque passage la commande à traiter. Celle-ci est stockée dans des tampons où elle est décodée, vérifiée, puis exécutée. Ce traitement, d'abord déroulé de manière entièrement interprétative dans une première version, est maintenant effectué à l'aide du Moniteur de Dialogue exposé dans le chapitre I. Cette modification a été réalisée dernièrement par Mademoiselle Sylvie LAFONT et Monsieur Patrice UVIETTA. Plus précisément, le Moniteur de Dialogue prend en charge le décodage et la vérification syntaxique des commandes, les vérifications sémantiques et l'exécution continuant à relever des autres programmes du superviseur.

Le premier module activé lors de l'appel du simulateur est l'ensemble des routines d'initialisation.

- Elles effectuent les initialisations système de plus haut niveau, telles que l'initialisation du Moniteur de Dialogue et celle du Moniteur de Communication. Elles lisent aussi, sur fichier, une partie du modèle à simuler, sa structure de données (SOS, SDS, zone valeur,...).

- Elles ne sont exécutées qu'une seule fois, au début de toute session de simulation.

- Elles passent le contrôle au Moniteur de Dialogue qui se comporte schématiquement comme une boucle d'attente et de récupération des commandes. Il vérifie la validité syntaxique de ces dernières et appelle les routines de traitement appropriées.

On trouvera, figure IV.2., l'architecture simplifiée du superviseur.

Notons en outre, relativement à l'organisation des programmes, que tous les textes, messages, messages d'erreurs, avertissements, etc... ont été regroupés dans un seul module.

Le module de traitement et d'exécution des commandes est développé dans la suite de ce document. On trouvera tout d'abord l'exposé de l'ensemble des fonctions primitives d'accès à la SOS, puis le détail de la sémantique de chacune des commandes et le principe de leur traitement.

IV.3. FONCTIONS D'ACCES A LA SOS (*)

On s'est efforcé, dans la définition de ces fonctions, de rester aussi indépendant que possible de leur environnement d'utilisation. C'est à-dire qu'elles peuvent être utilisées par toute séquence ou module ayant besoin d'accéder à des éléments de la SOS. Le découpage auquel nous sommes arrivés semble optimum pour l'ensemble des besoins répertoriés actuellement. Dans le contexte du superviseur, ces fonctions sont utilisées dans les commandes BOX, STO, PRI et TRA.

IV.3.1. FONCTION DE LOCALISATION D'UN MODULE DANS LA SOS

Utilisée par les commandes BOX et TRA, cette fonction permet d'accéder à un module quelconque de la hiérarchie du modèle. Pour le moment, nous ne traitons que le cas de modules simples et de modules structurés en vecteur. Le cas de modules dimensionnés de façon générale sera pris en compte ultérieurement.

La fonctionnalité est donnée figure IV.3.

(*) *On entend ici fonction au sens général du terme, pas le sens FORTRAN.*

ARCHITECTURE SIMPLIFIEE DU SUPERVISEUR

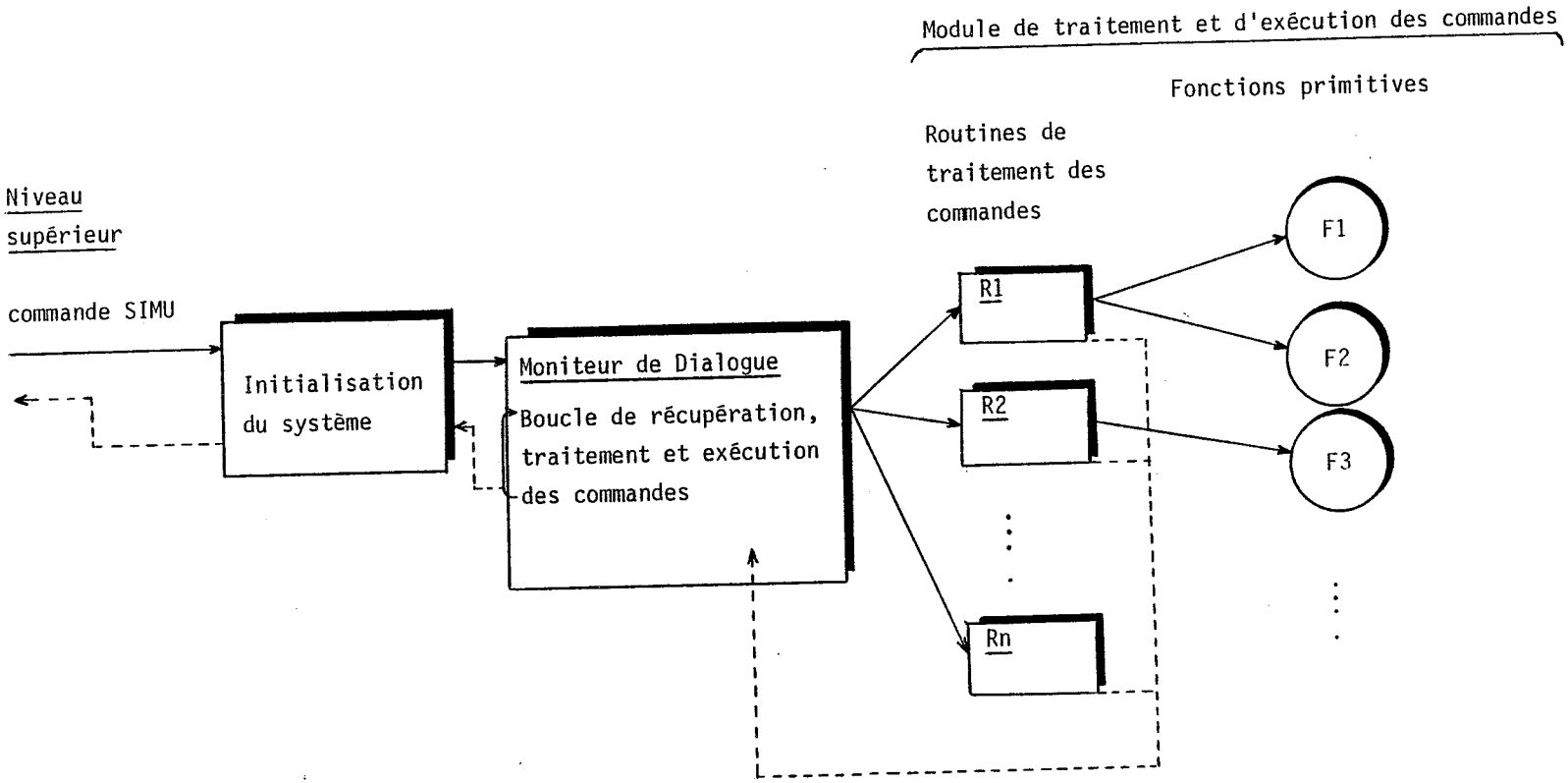
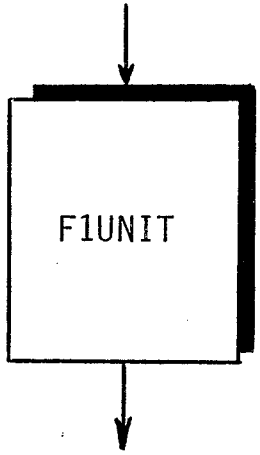


figure IV.2

Définition d'un chemin
(ex. : /A/C[2]/D)



Ensemble de paramètres localisant le module sélectionné (param-boîte-sélect) dans la SOS et la zone valeur

Exemple de chemin dans une hiérarchie (arbre)

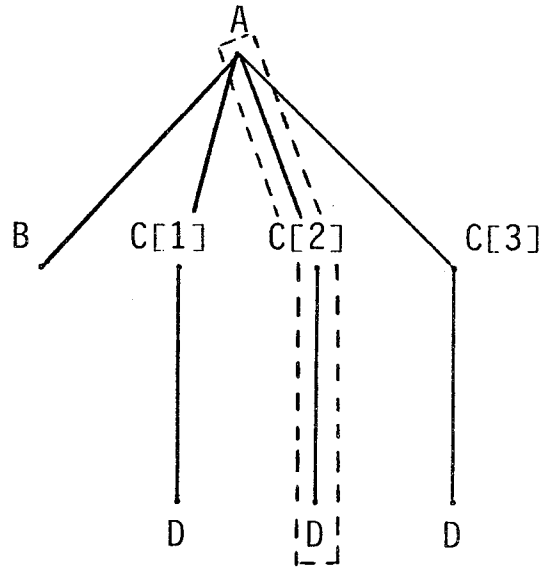


figure IV.3

IV.3.2. FONCTION DE LOCALISATION D'UN DESCRIPTEUR DE PORTEUSE DANS LA SOS

Elle est utilisée chaque fois que, pour un module donné, localisé, on veut accéder à une porteuse référencée par son nom. D'où la fonctionnalité (figure IV.4).

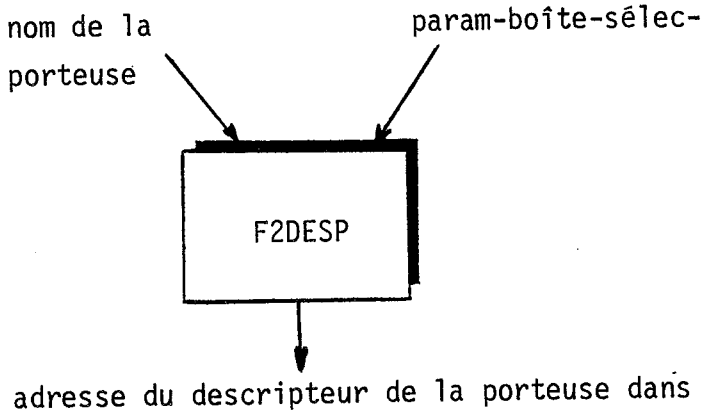


figure IV.4

L'appel de cette fonction n'a bien sûr de sens que s'il a été précédé d'un appel à la fonction de recherche de module.

IV.3.3. FONCTION DE LOCALISATION DE LA ZONE VALEUR D'UNE PORTEUSE

Elle est utilisée pour accéder à la zone valeur d'une porteuse, localisée dans un module donné. On obtient en retour l'adresse de la valeur de la porteuse. Si cette dernière est un tableau, c'est le premier élément qui est pointé, et on trouve la longueur de la zone valeur dans le descripteur. On peut ensuite lire ou écrire dans cette zone (figure IV.5).

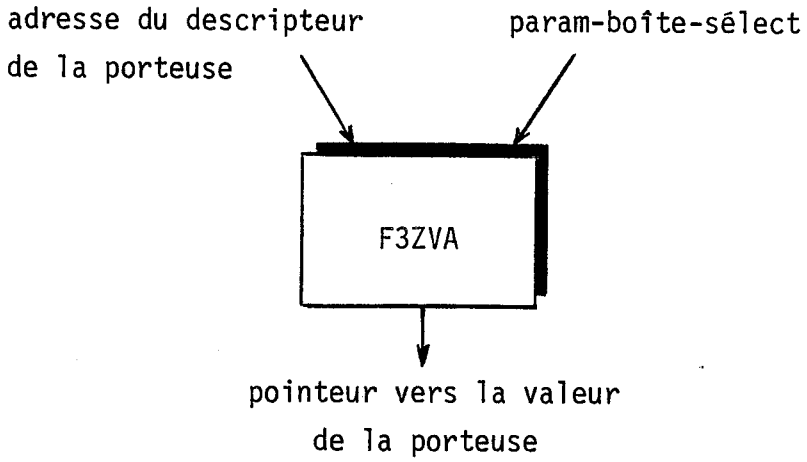


figure IV.5

Les paramètres de localisation du module sont indispensables à cette fonction à cause d'une certaine factorisation de la SOS. En effet, pour des modules identiques, une partie de la structure est unique alors que les zones valeur sont dupliquées autant de fois que nécessaire. Dans le prototype actuel, cette fonction localise la valeur de la porteuse à l'instant courant. Ultérieurement, elle pourra être généralisée afin de localiser les valeurs passées de la porteuse, dans le cas où celle-ci est retardée. Elle accèdera alors à l'historique associé à la porteuse, où sont stockées ces valeurs, et devra donc admettre un paramètre de plus en entrée : la valeur du retard ou de la date de stockage de la valeur recherchée.

IV.3.4. AUTRES FONCTIONS

Au cours du traitement des commandes, on est amené à avoir accès à certaines caractéristiques d'une porteuse, en général disponibles dans le descripteur de celle-ci. Il s'agit :

- du nom,
- du descripteur dimensionnel,
- du type de valeur.

La réalisation des trois fonctions suivantes, données figure IV.6, contribue ainsi à limiter le nombre de routines en contact "fin" avec la structure.

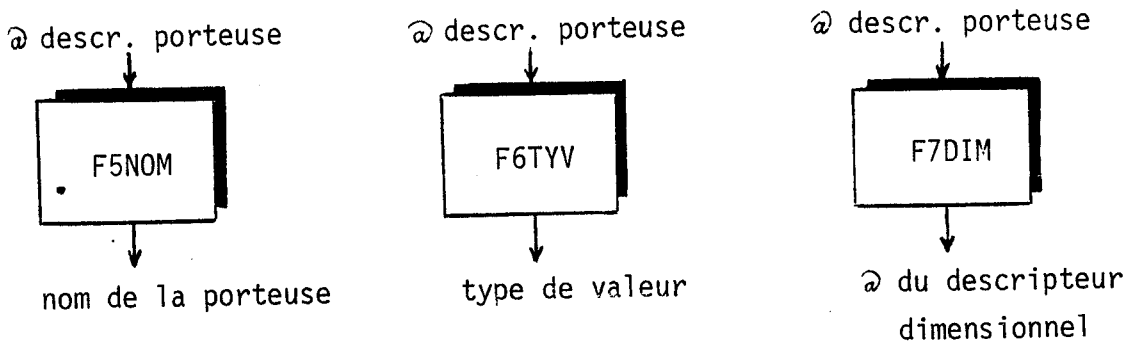


figure IV.6

IV.3.5. COMPLÉMENTS

Pour le prototype actuel, les six fonctions que nous venons d'exposer suffisent, mais il est nécessaire de les compléter par de nouvelles en vue d'obtenir une plus grande adaptabilité de ces outils à nos méthodes de compilation.

IV.3.5.1. FONCTION DE TRANSFERT DE VALEUR

Elle s'inscrit logiquement à la suite des précédentes et a pour but de manipuler les valeurs dont la zone de stockage a été localisée. Ses caractéristiques doivent être les suivantes :

- Les transferts sont bi-directionnels, c'est-à-dire que cette fonction doit servir aussi bien à lire qu'à écrire en zone valeur.
- Elle doit travailler aussi bien sur les porteuses dimensionnées que non dimensionnées.
- Pour une porteuse dimensionnée, elle doit permettre de faire référence aussi bien aux valeurs de tous les éléments du tableau qu'aux valeurs d'éléments constituant un sous-tableau. Dans le premier cas, nous dirons que nous avons affaire à un transfert total, la porteuse est considérée dans sa globalité ; dans le deuxième cas, nous parlerons de transfert partiel, le sous-tableau étant défini par une indexation de la porteuse.

- Elle utilise une zone de communication avec l'extérieur se comportant comme une pile. Il faut prévoir d'utiliser cette zone aussi bien pour transférer des mots que des bits, pour tenir compte du type de valeur de la porteuse.

Dans ses grandes lignes, la fonctionnalité est donnée figure IV.7.

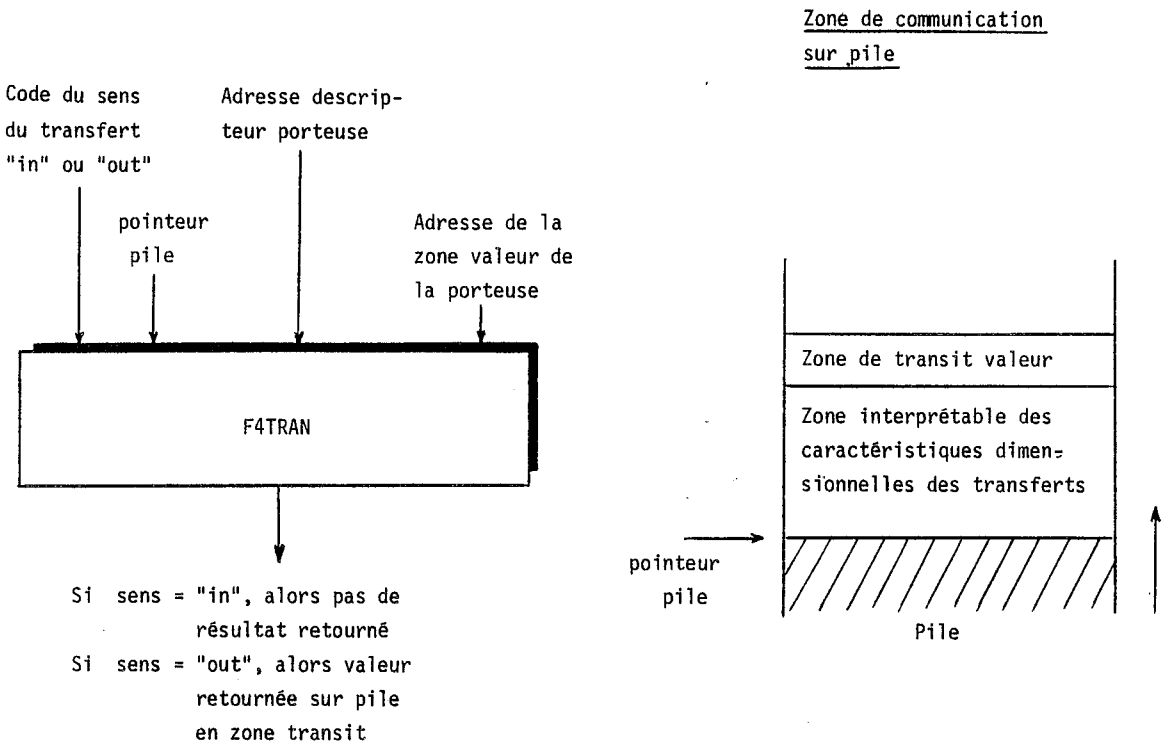


figure IV.7

REMARQUE : Nous supposons, à ce stade-là, que toutes les vérifications dimensionnelles et de compatibilité de type ont été effectuées.

- Dans le cas d'une lecture (code sens = "out"), la valeur de la porteuse, localisée par "adresse zone valeur", est empilée (zone transit).

- Dans le cas d'une écriture (code sens = "in"), la valeur préalablement empilée est stockée dans la zone valeur de la porteuse.

D'autre part, pour une porteuse dimensionnée, nous avons vu que deux situations étaient à considérer :

- Transfert Total : la zone interprétable, empilée avant l'appel à la fonction, est constituée simplement d'un code ("T").
  La fonction l'interprète en tant que tel et effectue le transfert de la valeur de tous les éléments de la porteuse, le nombre et la disposition de ces derniers lui étant donnés dans le descripteur dimensionnel de la porteuse, accessible à partir du descripteur localisé par F2DESP.
- Transfert Partiel : le code du type du transfert est à "P" et est surmonté d'un descripteur de l'indexation. Après interprétation du code, la fonction sélectionne les éléments de tableau concernés, par interprétation du descripteur d'indexation, et effectue les transferts de valeurs.

D'où la structure plus détaillée de la zone de communication, figure IV. 8.

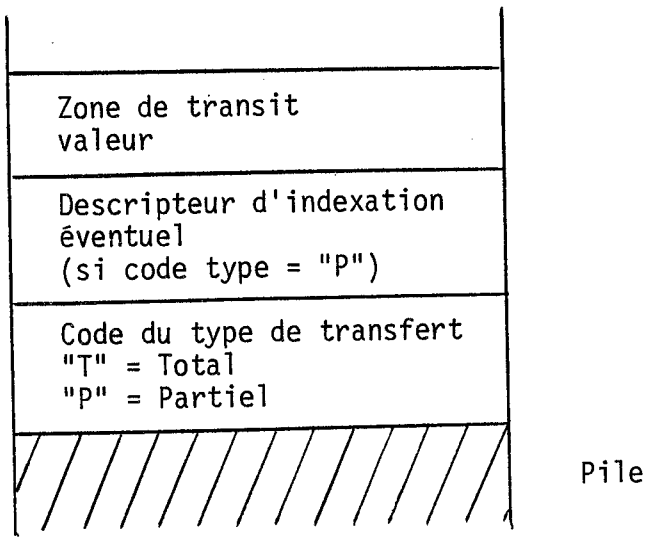
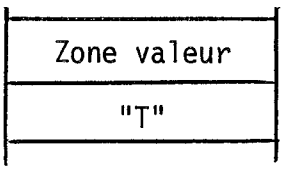


figure IV.8.

Les porteuses non dimensionnées sont un cas limite du précédent. Le transfert est obligatoirement total et le format des données sur la pile est le suivant, figure IV.9.



Le fait que l'objet soit non dimensionné est mentionné dans le descripteur de la porteuse.

figure IV.9.

IV.3.5.2. FONCTION D'IDENTIFICATION

Il est indispensable de pouvoir retrouver, à partir d'une adresse dans la zone valeur, le nom de la porteuse correspondante, ainsi que le nom du module auquel cette porteuse appartient. Ceci en passant par l'intermédiaire du descripteur de la porteuse qui est à retrouver dans sa liste (figure IV.10.).

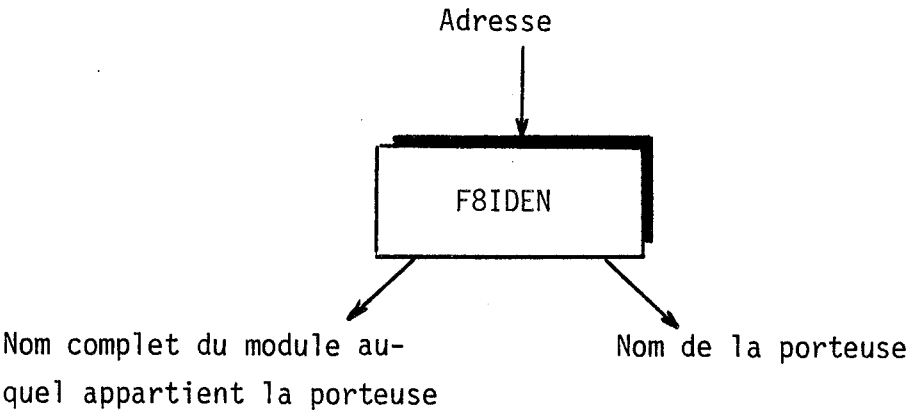


figure IV.10.

Ce travail, inverse de celui effectué par les fonctions vues précédemment, est nécessaire afin de pouvoir sortir des messages clairs et exploitables par l'utilisateur, lorsque des objets, connus alors seulement par leur adresse valeur ou celle d'un de leurs éléments dans le cas d'un tableau, sont impliqués dans des situations qui doivent être signalées à l'utilisateur, comme dans le cas d'un conflit de connexion ou de chargement, par exemple.

IV.4. FONCTIONNALITE DETAILLEE DES COMMANDES, TRAITEMENT

IV.4.1. BOX ET WAB

La nécessité d'accéder aux objets du modèle implique d'accéder d'abord au module qui les contient. Compte tenu des règles de portée d'identificateurs du langage CASCADE, il est nécessaire, pour lever toute ambiguïté de désignation, de préfixer le nom du module choisi avec ceux de tous les modules englobants, en tenant compte de ceux éventuellement dimensionnés (c'est le chemin qui va de la racine de l'arbre au module en question).

Par défaut, on considère qu'on est positionné, au départ de la simulation, sur le module le plus externe du modèle, c'est-à-dire la racine de l'arbre.

Les limitations actuelles de BOX sont celles exposées dans la définition de la fonction F1UNIT, de localisation d'un module donné dans la SOS, c'est-à-dire que l'on limite le dimensionnement des modules aux vecteurs.

La syntaxe est :

```
BOX  /racine de l'arbre/noeud suivant/.../noeud sélectionné
```

TRAITEMENT :

Après avoir récupéré la chaîne que constitue la définition du chemin, il suffit d'appeler la fonction F1UNIT et de stocker le résultat retourné dans une zone de données commune.

Le nom complet du module courant est stocké en permanence dans une zone prévue à cet effet. Si la recherche par F1UNIT est courronnée de succès, BOX stocke dans cette zone le nom du nouveau module courant, qui vient écraser l'ancien.

La commande WAB, dont la syntaxe est :

WAB a justement pour effet d'imprimer le contenu de cette zone. L'utilisateur peut ainsi connaître à tout moment où il est situé dans la hiérarchie du modèle.

Enfin, en début de simulation, le module d'initialisation réalise le traitement par défaut déjà signalé en faisant exécuter systématiquement une commande "BOX /racine de l'arbre" qu'il génère lui-même.

POSSIBILITES D'EXTENSION

En plus de la prise en compte des modules à dimensionnement général, il est possible de rendre la commande BOX plus agréable. La solution mise en oeuvre actuellement, et définie précédemment, peut s'avérer lourde à l'utilisation, dans le cas de nombreux niveaux d'imbrications, car le chemin à préciser chaque fois est alors long. Des commandes de déplacement pas à pas mémorisé, faisant mouvoir un pointeur fictif sur l'arbre, apporteraient une souplesse certaine.

On peut, par exemple, envisager :

- une commande BOU (pour BOX UP), qui permet de remonter n'importe où dans l'arbre, sur le chemin courant.

Exemple : BOU XXXX

où XXXX est le nom du composant du chemin où l'on veut aller.

Pratiquement, la routine de traitement a seulement à tronquer le chemin qui va de la racine à XXXX compris.

- Une commande BOD (pour BOX DOWN) qui permet de descendre d'un niveau dans l'arbre.
  Avec BOD XXXX, il suffit d'ajouter le module XXXX au chemin courant.

- Une commande BOS (pour BOX SIDE) qui permet un déplacement horizontal dans le niveau courant, sur des modules frères seulemnt (ayant même père). Pour BOS XXXX, la routine de traitement remplace l'extrémité du chemin courant par XXXX.

Si l'on ne veut pas multiplier le nombre de commandes, il suffit de concevoir une commande BOX avec options. De plus, ce dernier groupe de commandes peut être avantageusement remplacé par des moyens graphiques, où l'on montrerait sur l'arbre, affiché dans un coin de l'écran, le module dans lequel on désire aller.

IV.4.2. STO

Elle permet de ranger des valeurs dans les porteuses, et sa syntaxe est :

```
STO idf_porteuse = constante_à_ranger
```

Exemple :

```
STO H = `1, EM = `00001110,  A = 2
```

STO est utilisée pour modifier les valeurs contenues dans les objets spécifiés, au cours d'une pause de simulation, ou au début, pour modifier leur initialisation.

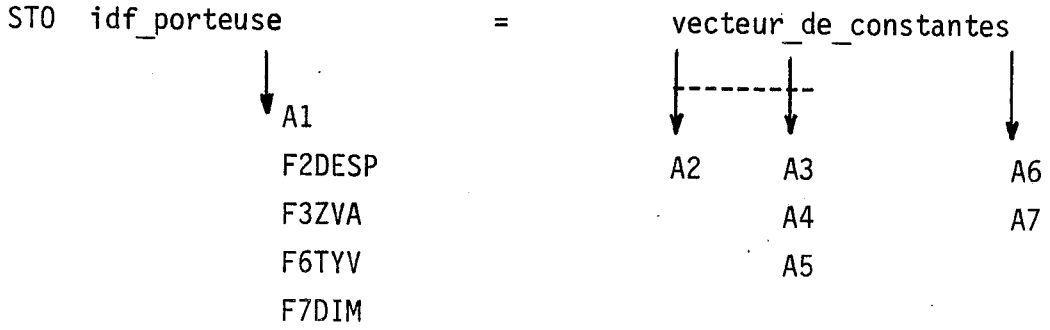
Pour le prototype actuel, les caractéristiques de cette commande sont les suivantes :

- on peut charger les porteuses non dimensionnées et celles dimensionnées, mais on est alors limité aux vecteurs.
- Dans ce dernier cas, le transfert est total, il ne peut y avoir d'indexation.
- Les types de valeurs acceptés aujourd'hui sont :

  ENTIER, BOOLEEN, IMPULSION, LOGIQUE et REEL.

TRAITEMENT

Les actions principales à dérouler au fur et à mesure de l'analyse de la commande, pour vérifier et exécuter cette dernière, sont les suivantes.



A1 : Récupère le nom de la porteuse spécifiée (fourni par le moniteur de dialogue).

F2DESP : Obtient, à partir de ce nom, l'adresse du descripteur de la porteuse.

F3ZVA : Obtient, à partir du descripteur de la porteuse, l'adresse de la zone valeur de cette dernière.

F6TYV : Récupère le type de valeur, accessible par le descripteur de la porteuse.

F7DIM : Récupère, à partir du descripteur de la porteuse, le descripteur dimensionnel (nombre d'éléments seulement, pour un vecteur).

A2 : Initialise à zéro un compteur C pour dénombrer les éléments du vecteur de constantes.

A3 : Récupère chacun des éléments de la constante et fait +1 dans C.

A4 : Vérifie que le type de chaque constante est bien celui attendu (voir F6TYV).

A5 : Comme les deux précédentes, elle concerne chaque élément du vecteur, qu'elle stocke dans une zone Z.

A6 : Vérifie, à l'aide du descripteur dimensionnel fourni par F7DIM, qu'il y a compatibilité dimensionnelle entre la réceptrice et la valeur chargée, c'est-à-dire que la porteuse est bien un vecteur et qu'elle a le même nombre d'éléments que le tableau de constantes (test de C).

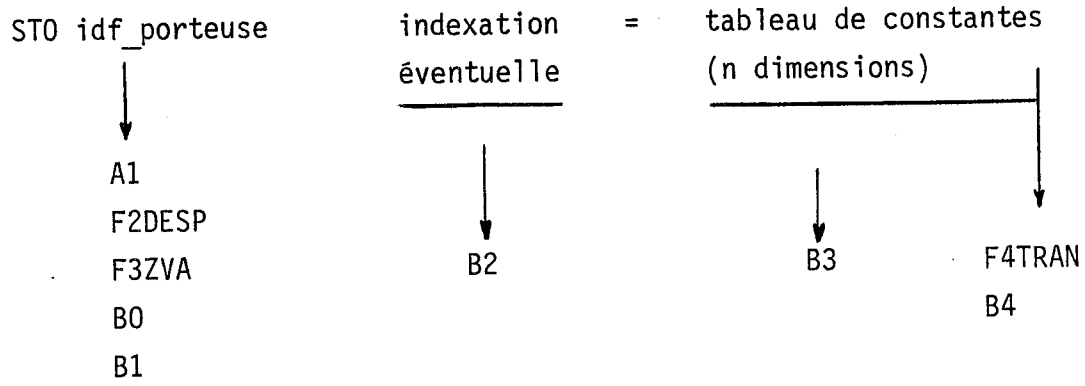
REMARQUE :

Le cas d'une porteuse non dimensionnée chargée par une constante est un cas particulier traité par une simplification de ce mécanisme.

A7 : Rangement des valeurs, préalablement stockées dans Z, dans la zone valeur de la porteuse (déjà localisée par l'appel à F3ZVA).

Notons que ces actions sont programmées directement dans la routine de traitement ou structurées en sous-programmes, comme les fonctions Fi ... par exemple.

Exemple de traitement général pour commande STO étendue, tenant compte d'un dimensionnement général et de l'indexation CASCADE. Nous nous contenterons d'indiquer seulement les grandes lignes du traitement.



A1 : Récupère le nom de la porteuse.

F2DESP : Obtient, à partir de ce nom, l'adresse du descripteur de la porteuse.

F3ZVA : Obtient, à partir de ce descripteur, l'adresse de la zone valeur.

B0 : Vérifie si la porteuse est modifiable.

B1 : Empile le type de valeur, défini dans le descripteur de la porteuse (figure IV.11).

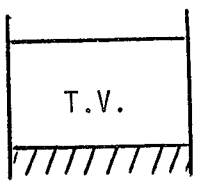


figure IV.11.

B2 : S'il n'y a pas d'indexation (transfert total), c'est une action qui se contente d'empiler le code du type de transfert, ici "T" (voir définition de la fonction F4TRAN).

Dans le cas contraire, c'est un ensemble d'actions activées au cours de l'analyse de l'indexation. La première empile le code "P" pour "transfert partiel". Ensuite, certaines effectuent les vérifications nécessaires (de compatibilité dimensionnelle surtout), les autres construisent le descripteur d'indexation sur la pile. Le contenu de cette dernière est alors le suivant (figure IV.12) :

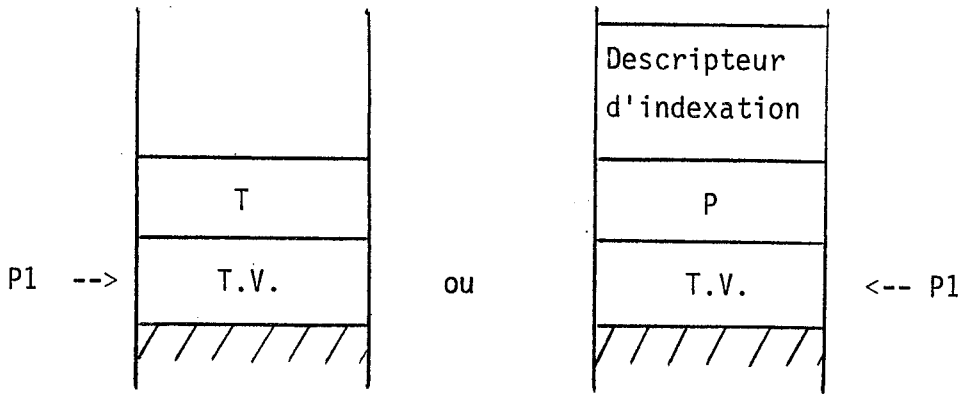


figure IV.12.

B3 : C'est aussi un ensemble d'actions activées au cours de l'analyse du tableau de constantes, effectuant les vérifications de type de valeur (avec T.V. empilé), et de compatibilité dimensionnelle. Enfin, elles empilent les constantes (valeurs à transférer).

D'où le contenu de la pile (figure IV.13.) :

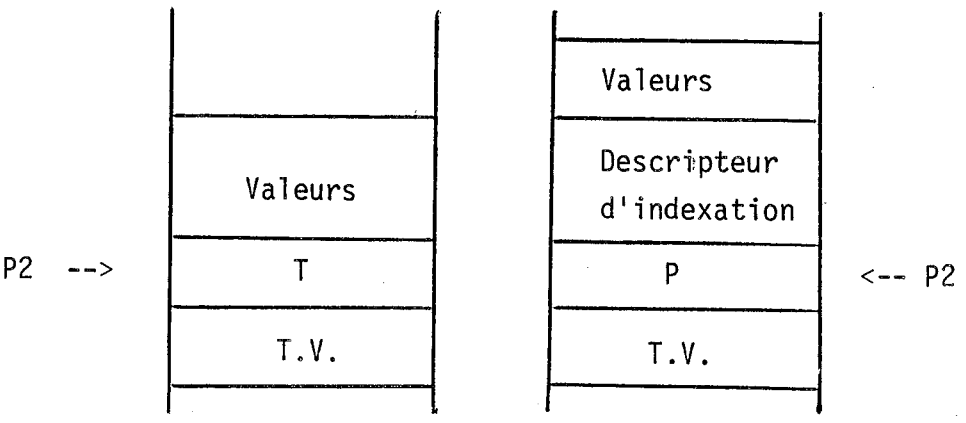


figure IV.13.

Notons que la réalisation de ces mécanismes nécessite de détailler B2 et B3 à un niveau beaucoup plus fin.

F4TRAN : Appel de la fonction de transfert de valeurs avec le code de sens à "in", le pointeur pile à la valeur P2, et les autres paramètres nécessaires. Le rangement en zone valeur de la porteuse est effectué.

B4 : Dépile tout ce qui a été empilé pour STO.

POSSIBILITÉ D'EXTENSION

Comme il vient d'être montré dans l'exemple de traitement précédent, il est souhaitable d'étendre la commande STO aux porteuses dimensionnées en général. Ce qui implique d'être capable de traiter des tableaux de constantes et la référence à des sous-tableaux (avec interdiction de répéter des éléments dans l'indexation).

Actuellement, tout objet accessible par la SOS et disposant d'une zone valeur peut être chargé. Aucune restriction n'intervient. Il est indispensable de limiter cette possibilité aux seules porteuses pour lesquelles cette opération a un sens. Les critères intervenant dans cette discrimination sont : le niveau de langage du module simulé, la nature des porteuses et leur localisation. Pour les niveaux Transfert de Registres CASSANDRE et LASCAR, par exemple, nous n'autoriserons le chargement par STO que pour certaines porteuses à mémorisation, comme les registres, et pour les entrées du modèle. Modifier les sorties du modèle ou des signaux internes n'offrirait aucun intérêt.

D'autres critères de restriction de cette notion peuvent être envisagés. On peut consacrer le STOre au chargement manuel des entrées du modèle, utiliser le FORCAGE pour les variables internes modifiables (registres, mémoires vives,...) et le FORCAGE des constantes, tout ceci étant réalisé avec STO.

Le FORCAGE des constantes pourrait être un niveau ultra-privilégié où l'on pourrait, par exemple, modifier des mémoire mortes décrites en CASCADE.

Ceci sous-entend des niveaux d'accès dans le langage de commande.

IV.4.3. PRI

Elle permet d'imprimer la valeur d'une porteuse, et sa forme est :

```
PRI idf_porteuse
```

C'est la commande duale de STO. Pour le prototype actuel, ses caractéristiques sont les suivantes :

- On ne peut imprimer que les porteuses non dimensionnées et les vecteurs.
- Dans ce dernier cas, c'est le vecteur tout entier qui est imprimé.
- Les types de valeurs acceptés aujourd'hui sont :

  ENTIER, BOOLEEN, IMPULSION, LOGIQUE et REEL.

TRAITEMENT

Les actions à dérouler au fur et à mesure de l'analyse de la commande sont les suivantes :

PRI idf_porteuse
↓
A1,F2DESP,F3ZVA,F6TYV,F7DIM,C1

A1 : Récupère le nom de la porteuse spécifiée.

F2DESP : Obtient, à partir de ce nom, le descripteur de la porteuse.

F3ZVA : Obtient, à partir du descripteur de la porteuse, l'adresse de sa zone valeur.

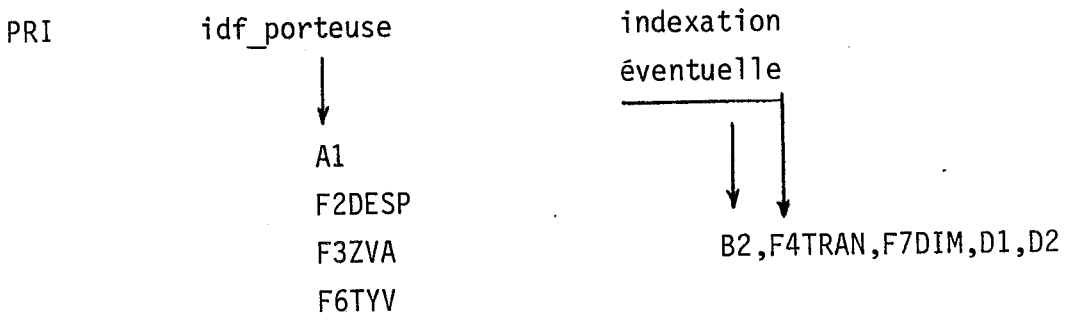
F6TYV : Récupère dans le descripteur le type de valeur.

F7DIM : Récupère dans le descripteur de la porteuse le descripteur dimensionnel. L'information à prendre ici est : si la porteuse est dimensionnée, c'est un vecteur, et on récupère le nombre d'éléments n.

C1 : Imprime la valeur de la porteuse, éventuellement sous forme d'un vecteur de n constantes. La connaissance du type de valeur est nécessaire pour effectuer d'éventuelles conversions:données internes --> forme imprimable (exemple : les états d'automate).

Exemple de traitement général pour commande PRI étendue

En première approximation, nous avons :



A1,F2DESP,F3ZVA,F6TYV : déjà vues.

B2 Déjà vue pour STO. Après analyse de l'indexation, la pile contient alors (figure IV.14) :

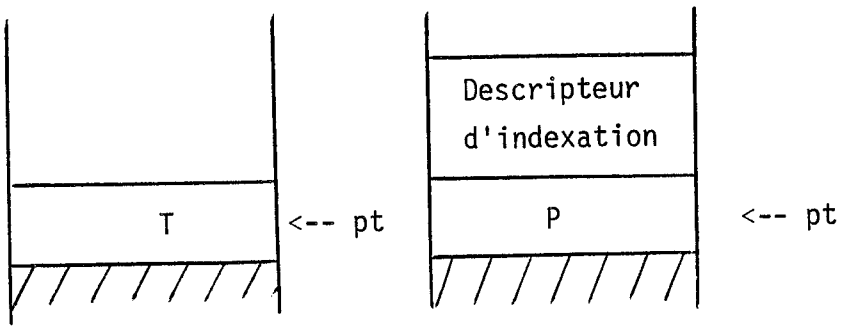


figure IV.14.

F4TRAN : Appel de la fonction de transfert de valeurs avec le code "out", le pointeur pile à la valeur pt, et les autres paramètres nécessaires. Le transfert zone valeur --> pile est alors effectué (figure IV.15)

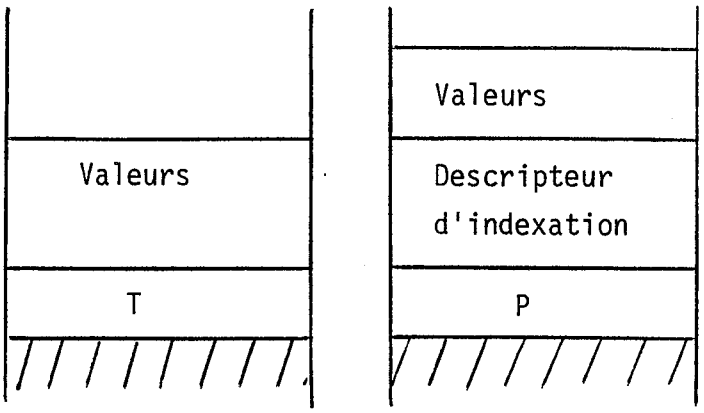


figure IV.15

F7DIM : Accède au descripteur dimensionnel de la porteuse.

D1 : Imprime la valeur empilée sous forme d'un tableau n dimensions dans le cas général, avec conversion éventuelle des données internes en données imprimables.

D2 : Dépile tout ce qui a été empilé pour PRI.

POSSIBILITÉS D'EXTENSION

Tout comme pour STO, le traitement de porteuses dimensionnées et indexées suivant les conventions CASCADE doit être pris en compte dans PRI.

Il serait aussi utile de disposer d'options permettant de choisir l'aspect externe des valeurs de certains types.

Exemple :

- Impression de réels sous forme de puissance de dix ou avec facteur d'échelle.
- Impression d'entiers en base décimale, hexadécimale, octale, et pourquoi pas binaire.
- Impression de vecteurs booléens sous forme d'entiers en base décimale, hexadécimale ou octale.

Enfin, on pourrait généraliser PRI par une option provoquant l'impression des valeurs de toutes les porteuses du module courant.

Exemple : PRI *ALL

IV.4.4. TRA ET ETR

Cette commande permet de stocker dans un fichier les valeurs successives prises par des porteuses, durant une simulation ou une partie de celle-ci. Il s'agit en fait d'un environnement dans lequel on entre par :

```
TRA entier [+T]
```

où "entier" est le numéro d'identification de la trace, compris entre 0 et 99, et écrit sur deux chiffres. L'option "+T" provoque l'impression de la trace au terminal durant la simulation, en plus du stockage dans le fichier trace.

L'utilisateur dispose ensuite de deux commandes permettant de définir la liste des objets à tracer. Une première :

```
MODULE nom_complet_de_module
```

permet de sélectionner un module de son choix dans l'arbre du modèle. La liste d'objets locaux à ce module peut alors être définie par :

```
OBJETS idf_objet_1,idf_objet_2, etc
```

On peut reboucler autant de fois qu'on le désire sur ces deux commandes. On indique que la demande de trace est terminée par FINTRA . On sort alors de l'environnement TRA.

Le nom du fichier de stockage est construit à l'aide du numéro d'identification de la trace et de la chaine de caractères constituant le nom du modèle simulé, suivant le format

nom_modèle.VXX
↳ N° de trace.

Le stockage prend effet dès l'émission de la commande et se poursuit jusqu'à ce que l'on sorte du simulateur ou que l'on émette la commande ETR prévue à cet effet.

On ne peut avoir qu'une seule trace ouverte à la fois.

Les caractéristiques de cette commande se rapprochent assez de celles de la commande PRI (ses limitations vis à vis des objets sont les mêmes). L'originalité de TRA réside dans le fait qu'elle provoque des actions en différé : le stockage des valeurs dans un fichier, durant la simulation.

TRAITEMENT

Il se présente comme suit, pour TRA :

- Vérification qu'une trace n'est pas déjà en cours, si oui, il y a émission d'un message d'erreur.
- Récupération du nom du modèle et du numéro de la trace, et construction du nom du fichier.
- Création et ouverture de ce dernier. Attention, s'il existe déjà, l'ancien est effacé.

Ensuite, avec MODULE et OBJETS :

- Décodage des identificateurs, vérification de l'existence des porteuses et des modules spécifiés, de leur éventuelle indexation, etc... comme dans BOX et PRI.
- Construction, au fure et à mesure de l'analyse de ces deux commandes, d'une structure interprétable contenant les adresses réelles de la zone valeur de chaque objet.

Le stockage des valeurs dans le fichier est effectué par une routine interprétant cette table, appelée par le séquenceur pour les parties logiques du modèle, à la fin de chaque cycle, et par les algorithmes de calcul pour les parties électriques.

La commande ETR, enfin,

- suspend l'appel à la routine d'interprétation
- ferme le fichier contenant la trace
- reste sans effet et imprime un avertissement si la trace n'est pas ouverte.

Le fichier trace ainsi obtenu peut être exploité ultérieurement, à loisirs, à l'aide de l'environnement EDIVAL, permettant de définir des listes, des chronogrammes, etc...

Les commandes qui vont suivre dans cet exposé, bien qu'indispensables, sont moins intéressantes que celles que nous venons de voir, sur le plan traitement, car elles ne font pas référence à des objets du modèle. Aussi les avons-nous regroupées dans un seul paragraphe.

IV.4.5. AUTRES COMMANDES

B T U

Cette commande est nécessaire pour le cas de modèles écrits en partie à un niveau logique et en partie à un niveau électrique. En effet, pour faire communiquer ces deux types de simulation, il faut préciser la correspondance entre l'échelle de temps logique exprimée en unités de temps de base et l'échelle de temps utilisée aux niveaux électriques, exprimée en secondes.

D'où la commande BTU , permettant de fixer, en secondes, la valeur de l'unité de temps logique.

Le format est :

```
BTU constante_réelle
```

Ce paramètre est passé au séquenceur.

R U N

La syntaxe est

```
RUN n
```

où n est la date exprimée en unité de temps logique (BTU), jusqu'à laquelle l'utilisateur désire faire fonctionner le modèle. C'est donc un entier positif, et l'on doit avoir :

n > t.courant

La routine de traitement appelle le séquenceur en lui fournissant les deux dates t.courant et n. C'est le séquenceur qui remet t.courant à jour et le retourne à la routine.

STP

La syntaxe est STP n

où n ne représente plus une date absolue mais une durée pendant laquelle on demande au modèle de fonctionner, à partir de la date courante t_c ; ce qui est équivalent à :

RUN p où $p = t_c + n$

?

Il est indispensable en cours de simulation de connaître, lorsqu'on le désire, où en est le temps. La commande "?" provoque l'impression de la valeur courante du temps t_c.

HEL

C'est la commande "help", qui a pour effet d'imprimer le menu au terminal, c'est-à-dire l'ensemble des commandes disponibles pour l'utilisateur, dans le simulateur. Un texte bref, de une à deux lignes donne la signification de chacune des commandes.

La réalisation de "help(s)" spécialisés pour chaque commande est indispensable.

Exemple : HEL TRA pour obtenir les explications minimum nécessaires sur la commande TRA et son utilisation.

Tout un jeu de commandes est spécifique à la simulation électrique. Il s'agit de : DC, HMIN et HMAX, ERMAX, INIT et TR, dont la signification a été brièvement donnée au début de ce chapitre.

Nous ne nous attarderons pas sur leur traitement qui consiste simplement soit à positionner des indicateurs, soit à récupérer des paramètres numériques pour les algorithmes de calcul électriques.

Il en est de même pour les commandes "système" TIM et EOT, DGE et EDG, DTE et ETE.

END

C'est la commande pour sortir de l'environnement de simulation. Elle positionne un indicateur pour transmettre le renseignement au Moniteur de Dialogue qui interrompra sa boucle et passera le contrôle au niveau supérieur.

La routine de traitement de cette commande a aussi pour rôle de terminer proprement la simulation en cours : fermeture de la trace en cours encore ouverte, vidage de buffers éventuels, etc...

IV.4.6. EXTENSION DU JEU DE COMMANDES

Il serait souhaitable de disposer, à l'avenir, d'un certain nombre de commandes que nous n'avons pas eu le temps de réaliser. Il s'agit de :

GRA

pour afficher la hiérarchie du modèle, c'est-à-dire de l'arbre externe des modules, défini par l'utilisateur.

WHO

pour afficher la liste des noms de tous les objets du module courant.

SAV

pour sauvegarder l'état du modèle. Cette possibilité est particulièrement intéressante pour les longues simulations où elle permet des interruptions dans le travail. De plus, si elle est fréquemment employée, elle permet des retours arrière en cas d'erreur ou de fausse manoeuvre. Enfin, si un grand nombre de simulations doit s'effectuer à partir d'une certaine configuration du modèle, cette commande permet de "photographier" le modèle convenablement initialisé une fois pour toutes, et il suffit de le restaurer dans cet état-là avant chaque simulation.

Le format de la commande est SAV n

où n, entier positif, est un numéro affecté à l'opération de sauvegarde.

La "photographie" du modèle est stockée dans un fichier dont l'identification exacte reste à définir, mais fait intervenir le nom du modèle et n.

Ce qui est à sauvegarder est évidemment tout ce qui présente un caractère dynamique, non seulement dans le modèle, mais aussi dans le simulateur, à savoir :

Modèle

- Toute la zone valeur.
- Tous les indicateurs spécifiques au modèle.

Simulateur

- Indicateurs d'état du séquenceur : valeur du temps courant, etc...

Il reste à étudier et à décider si on sauvegarde aussi le contexte du superviseur de simulation : trace ouverte , etc... ; il peut y avoir des incompatibilités.

Remarquons qu'il n'est pas nécessaire de sauvegarder les blocs de simulation, ni la SOS.

R E S

Elle a pour fonction de restaurer un modèle en cours de simulation, préalablement sauvegardé.

Son format est

RES n

S I M

C'est une commande permettant d'activer, à n'importe quel moment de la simulation, des fichiers de commandes. L'exécution de SIM CALCUL, par exemple aura pour effet de lancer l'exécution en séquence des commandes contenues dans le fichier CALCUL.SIM. Ensuite, le Moniteur de Dialogue rend la main à l'utilisateur.

Des commandes de contrôle, permettant de programmer des sauts, des boucles, etc... sont à définir afin de pouvoir profiter de toute la puissance du mode fichier que nous venons d'évoquer. Un mécanisme gérant les appels de fichiers de commandes par d'autres fichiers de commandes est aussi à prévoir.

T A C

Un fichier de commandes, utilisé comme macro-commande, peut être créé par l'éditeur de textes, mais peut aussi être généré automatiquement au cours de l'exécution d'une séquence de simulation.

TAC, de format

TAC nom

a pour effet de provoquer, à partir du moment où elle est émise, le stockage dans un fichier de toutes les commandes correctes traitées, quelle que soit leur origine, et dont l'exécution s'est déroulée normalement. Le fichier est identifié par nom.SIM.

a pour effet de provoquer, à partir du moment où elle est émise, le stockage dans un fichier de toutes les commandes correctes traitées, quelle que soit leur origine, et dont l'exécution s'est déroulée normalement. Le fichier créé, est identifié par

nom.SIM

E T C

A pour effet de fermer la trace commande ci-dessus, s'il y en a une d'ouverte. Dans le cas contraire, elle reste sans effet et imprime un avertissement.

IV.5. EXEMPLE D'UTILISATION DU SUPERVISEUR DE SIMULATION

Le superviseur de simulation est utilisé intensivement dans le cadre du développement, des essais et de la mise au point de la chaîne des programmes CASCADE. Il contribue notamment à la validation du système lors du passage des benchmarks prévus (Bre.84c, BrL.85).

Nous allons illustrer cette communication du modèle avec son environnement et l'utilisateur, objet du présent chapitre, par un exemple de session de simulation. Le modèle choisi est celui d'un comparateur séquentiel de quatre mots de quatre bits.

Ce circuit est réalisé à l'aide d'un comparateur, entièrement combinatoire, donnant le plus grand de deux mots quelconques. Le circuit global est organisé pour fonctionner séquentiellement, en comparant le dernier mot le plus grand, trouvé parmi deux, avec le suivant. Cette façon de faire se justifie par le fait que la combinatoire pour un comparateur quatre mots serait beaucoup trop importante.

On arrive ainsi à effectuer le calcul en trois phases :

① Comparaison (A, B)
↓
② Comparaison (Résultat, C)
↓
③ Comparaison (Résultat, D)
↓
Résultat final

Le modèle CIRCUIT comprend donc le comparateur deux mots COMP4, avec en plus l'automate de séquencement AUTO et son générateur d'horloge bi-phase TIME ; (Voir figure IV.16).

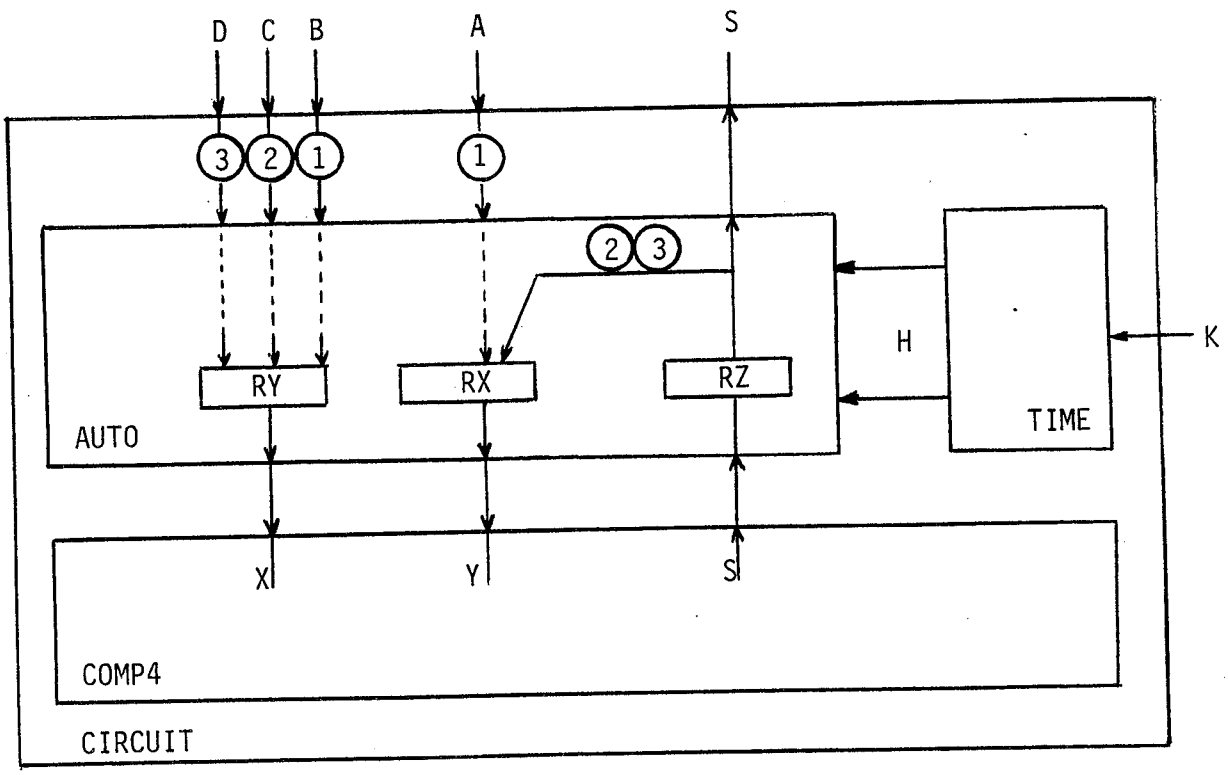


figure IV.16

Description du circuit en CASSANDRE_S

```
<< Description du circuit de comparaison de
   4 mots de 4 bits >>

lanref CASSANDRE+S

description CIRCUIT (in HORLOGE K; in SIGBØ A[3:Ø],
         B[3:Ø], C[3:Ø], D[3:Ø];
        out SIGBØ S[3:Ø] )
corps
  declare
   HORLOGE H[1:2] ;
   SIGBØ X[3:Ø],Y[3:Ø],Z[3:Ø] ;

   externe COMP, TIMER , AUTOC ;
   use COMP COMP4 (X, Y, Z);
       TIMER TIME (K, H) ;
       AUTOC AUTO (A,B,C,D,H,Z,X,Y,S);
enddescr



<<  Automate gerant la comparaison de
   4 mots de 4 bits >>

lanref CASSANDRE+S

description AUTOC( in SIGBØ A[3:Ø],
         B[3:Ø], C[3:Ø], D[3:Ø];
         in HORLOGE H[1:2]; in SIGBØ Z[3:Ø];
        out SIGBØ X[3:Ø], Y[3:Ø];
        out SIGBØ S[3:Ø] )
corps
  declare
   REGBØ RZ[3:Ø],RX[3:Ø],RY[3:Ø];

 relations
   << Connection interne >>
     X .= RX,
     Y .= RY,
     S .= RZ
   << Sequencement : AUTOMATE >>
  :E1: !H[1]! RX <= A, RY <= B;
       !H[2]! RZ <= Z, load E2 ;
  :E2: !H[1]! RY <= C, RX <= RZ;
       !H[2]! RZ <= Z, load E3 ;
  :E3: !H[1]! RY <= D, RX <= RZ;
       !H[2]! RZ <= Z, load E1;
enddescr
```

```
<< Description du comparateur de mots de 4 bits >>
lanref CASSANDRE+S
description COMP ( in SIGBØ X[3:Ø], Y[3:Ø] ;
                   out SIGBØ S[3:Ø] )

corps
  declare
    SIGBØ R ;
relations
 R .=     X[3]              & ~Y[3]
      ; ~(X[3] xor Y[3]) & ((X[2]&~Y[2])
      ; ~(X[2] xor Y[2]) & ((X[1]&~Y[1])
      ; ~(X[1] xor Y[1]) & (X[Ø]&~Y[Ø]))) ,
 pour I depuis Ø jusqua 3
   repeter
    S[I] .= R&X[I] ! ~R&Y[I]
   fpour
enddescr
```

```
<< Description du timer :
      generateur d'horloge biphase >>
lanref CASSANDRE+S
description TIMER (in HORLOGE K ; out HORLOGE H[1:2] )
  corps
    type HAUT=SIGNAL(IMPULSION,1) ;
   declare
    HAUT H1 ;
    HORLOGE HØ ;

   relations
    :E1: H[1] .= H1 , H[2] .= HØ , !K! load E2 ;
    :E2: H[1] .= HØ , H[2] .= H1 , !K! load E1 ;
 enddescription
```

Simulation du circuit

```
@NCASCADE

        + NOUVEAU CASCADE +

    Phase1 = 1, Phase2 = 2, Simu = 3, Edit = 4, End = Carriage Return : 3

**  SIMULATION  **
    Model Name ? CIRCUIT

    External Command File, Y/N ? Y
    Command File Name ? CIRCUIT
```

```
    ********  Session of CASCADE SIMULATION  ********

  Command:  << Fonctionnement en mode fichier >>

  Command:  << Activation de l'horloge de base >>

  Command:  STO K=`1

  Command:  << Chargement des quatre entrees A, B, C et D >>

  Command:  STO A=`0010 , B=`0001 , C=`1000 , D=`1001

  Command:  << Comparaison de A et B >>

  Command:  RUN 3
 Inquired Logical Date
           3

  Command:  << Examen de la sortie >>

  Command:  PRI S
  `0010

  Command:  << Comparaison du resultat avec C >>

  Command:  STP 2
 Inquired Logical Date
           5

  Command:  PRI S
  `1000

  Command:  << Comparaison du resultat avec D >>

  Command:  STP 2
 Inquired Logical Date
           7

  Command:  PRI S
  `1001

  Command:  << Modifions l'entree C >>

  Command:  STO C=`1111

  Command:  << et effectuons les 6 cycles necessaires pour la

  Command:  comparaison des 4 mots >>

  Command:  ?
Current Time =
    7
```

```
 Command:  RUN 13
Inquired Logical Date
        13

 Command:  << et examinons le resultat >>

 Command:  PRI S
`1111

 Command:  << Terminons la session de simulation >>

 Command:  END STOP

      ******  END of the Session of CASCADE Simulation******
```

Autre simulation

```
@NCASCADE

        ← NOUVEAU CASCADE ←

    Phase1 = 1, Phase2 = 2, Simu = 3, Edit = 4, End = Carriage Return : 3

**  SIMULATION  **
    Model Name ? CIRCUIT

    External Command File, Y/N ? N


        ********  Session of CASCADE SIMULATION  ********

      -> << Cette simulation de CIRCUIT est deroulee de maniere entierement
      ->    interactive.>>
      -> << Elle a pour but de montrer l'evolution de l'automate de
      ->    sequencement , ainsi que de mettre en evidence les variations de
      ->    l'horloge biphase H, tracee sous forme de chronogramme. >>
      -> << Nous en profiterons pour afficher aussi les valeurs successives
      ->    prises par la sortie S. >>
      -> <<>>
```

```
-> << Rappel des principales commandes du simulateur >>
-> HEL
   ********  List of Commands  (asked with HELP) ********
   *                                                     *
   *                 GENERAL  COMMANDS                   *
   *                                                     *
   * RUN: to drive the simulation of the model           *
   * STOre: to store values in the object (carriers)     *
   * PRInt: to print values of the objects (carriers)    *
   * BOX: to move in the tree of the model               *
   * WAB: to print the full name of the current module   *
   * TRAce: to store in a file the values taken by the   *
   *      carriers during the simulation                 *
   *      See EDIVAL environment for edition and         *
   *      exploitation of the trace                      *
   * ETR: to stop storage already defined by TRA         *
   * HELp: to print this list of commands                *
   * TIMe: to request measurement of the CPU time during*
   *       the next RUN commands                         *
   * EOT: to stop the last measurement requested by TIM  *
   * !: to announce a comment                            *
   * END: to end the current session of simulation       *
   *                                                     *
   *   SPECIFIC  COMMANDS  FOR  ELECTRICAL  LEVELS       *
   *                                                     *
   * BTU: to establish a correspondence between the      *
   *      logical time unit and the electrical time unit*
   * DGE: request for Dump Global Electric               *
   * EDG: to stop the last dump requested by DGE         *
   * DTE: request for Dump Time Electric                 *
   * ETE: to stop the last dump requested by DTE         *
   * DC: selection of method for initialisation          *
   * TR: selection of method in transient analysis       *
   * HMIN: minimum step of integration                   *
   * HMAX: maximum step of integration                   *
   * ERMAX: maximum error value conceded                 *
   * INIT: default init for the values of the algebro-   *
   *       differential system                           *
   *                                                     *
   *******************************************************

-> << Preparation de la trace de l'horloge biphase et de la sortie. >>
-> TRA Ø1
-> MODULE /CIRCUIT/TIME
-> OBJETS K,H
-> MODULE /CIRCUIT
-> OBJETS S
-> FINTRA
Opening of the trace number :
    Ø1
-> << Initialisation de la simulation. >>
-> STO K=`1,A=`ØØ1Ø,B=`ØØØ1,C=`1Ø1Ø,D=`11ØØ
-> << Positionnons-nous maintenant dans l'automate >>
-> BOX /CIRCUIT/AUTO
-> << et deroulons la simulation>>
```

```
-> << Nous verifions d'abord que l'automate est bien dans le premier
->    etat ( on est positionne dans l'etat que l'on VA executer)>>
-> PRI $
 E1
-> RUN 3
Inquired Logical Date
          3
-> << Verifions les transitions de l'automate. >>
-> PRI $
 E2
-> STP 2
Inquired Logical Date
          5
-> PRI $
 E3
-> STP 2
Inquired Logical Date
          7
-> PRI $
 E1
-> << Retournons dans le module CIRCUIT >>
-> BOX /CIRCUIT
-> << Verification >>
-> WAB
/CIRCUIT
-> << Verifions la valeur du resultat >>
-> PRI S
 `11ØØ
-> << FIN >>
-> END
The current trace is closed :
    Ø1

      ******  END of the Session of CASCADE Simulation******


  Phase1 = 1, Phase2 = 2, Simu = 3, Edit = 4, End = Carriage Return : 4


    Enter the name of the model :
  CIRCUIT
    Enter the number of the trace
  Ø1
  Do you want the trace :
    1 - at the screen only ?
    2 - in a file to be printed, only ? (name-of-model.EXX)
    3 - at the screen and in a file ?
          Answer 1,2 or 3
  3
  Of what edition have you need ? (enter its number)
    1 - General edition of all objects of the trace (all types)
    2 - Choronogram of all objects of the trace
        They must not represent more than 17 elements and theymust
        been of type of values: Boolean, Pulse, Logic (4 values
        `Ø, `1, `u, `z) or normalized Real near of interval
        [Ø.Ø - 1.Ø]                          ...mixture is possible
  2
```

TRACE NUMBER 01 OF MODEL CIRCUIT

- MODULE /CIRCUIT/TIME
  .1 K
  .2 H

- MODULE /CIRCUIT
  .3 S

```
                    1       2       2       3       3       3       3
                I
                I   K       H               S
   DATES        I
                I   Ø 1   Ø 1   Ø 1   Ø 1   Ø 1   Ø 1   Ø 1
-------------------------------------------------------------------
                I
                I
              Ø I   Ø       Ø       Ø       Ø       Ø       Ø       Ø
                I   Ø       Ø       Ø       Ø       Ø       Ø       Ø
                I   Ø       Ø       Ø       Ø       Ø       Ø       Ø
              Ø I   ---1    Ø       Ø       Ø       Ø       Ø       Ø
                I       1   Ø       Ø       Ø       Ø       Ø       Ø
                I       1   Ø       Ø       Ø       Ø       Ø       Ø
              1 I       1   ---1    Ø       Ø       Ø       Ø       Ø
                I       1       1   Ø       Ø       Ø       Ø       Ø
                I       1       1   Ø       Ø       Ø       Ø       Ø
              2 I       1   Ø---    ---1    Ø       Ø       Ø       Ø
                I       1   Ø           1   Ø       Ø       Ø       Ø
                I       1   Ø           1   Ø       Ø       Ø       Ø
              3 I       1   ---1    Ø---    Ø       Ø       ---1    Ø
                I       1       1   Ø       Ø       Ø           1   Ø
                I       1       1   Ø       Ø       Ø           1   Ø
              4 I       1   Ø---    ---1    Ø       Ø           1   Ø
                I       1   Ø           1   Ø       Ø           1   Ø
                I       1   Ø           1   Ø       Ø           1   Ø
              5 I       1   ---1    Ø---    ---1    Ø           1   Ø
                I       1       1   Ø           1   Ø           1   Ø
                I       1       1   Ø           1   Ø           1   Ø
              6 I       1   Ø---    ---1        1   Ø           1   Ø
                I       1   Ø           1       1   Ø           1   Ø
                I       1   Ø           1       1   Ø           1   Ø
              7 I       1   ---1    Ø---        1   ---1    Ø---    Ø
                I       1       1   Ø           1       1   Ø       Ø
                I       1       1   Ø           1       1   Ø       Ø
```

IV.6. PROGRAMMATION

Le superviseur de simulation CASCADE est programmé en FORTRAN 77 (VAX.83). Bien que FORTRAN accuse un certain retard par rapport à l'évolution générale des méthodes et des techniques du génie logiciel, on peut, par l'observation de règles strictes, l'utiliser pour écrire des programmes de bonne qualité, adaptables à des ordinateurs différents (Vap.82).

Pour qu'un programme soit utilisable sur plusieurs machines et par des utilisateurs différents, il ne suffit pas qu'il soit strictement portable. C'est pourquoi nous nous sommes efforcés de concevoir le programme de ce superviseur de façon à ce qu'il soit facilement compréhensible, c'est-à-dire qu'il soit clair, lisible et bien structuré.

La pratique montre, de plus, qu'il est impossible d'écrire un système tel que CASCADE, complètement portable. La norme doit, en effet, être transgressée lorsque l'accès à une primitive de la machine ou du système est souhaité ou lorsque la portabilité est acquise trop au détriment de l'efficacité. Mais alors, dans tous ces cas, cela ne concerne que des parties isolées, de petite taille, signalées dans le programme par des commentaires, et dans sa documentation.

Le respect de normes de programmation dans un langage n'est qu'un des moyens concourant à garantir la qualité du logiciel. De telles normes doivent s'insérer dans un plan de qualité. C'est pourquoi nous avons l'intention d'utiliser au plus vite l'analyseur de programme QUALIMETRE-C, que nous avons acquis tout récemment.

A partir du seul code source des programmes, ce logiciel produit automatiquement une documentation utile pour les tests et la maintenance ; de plus, il fournit l'ensemble des mesures classiques de complexité structurelle et contextuelle des programmes (Del.84).

Terminons ce paragraphe en précisant, à titre d'information, que le volume du superviseur de simulation est de 6000 lignes FORTRAN.

C O N C L U S I O N

Le système CASCADE est aujourd'hui à l'état de prototype qui fonctionne, mais il ne peut être confié à des utilisateurs inexpérimentés. Tout en poursuivant l'introduction de fonctionnalités nouvelles, une phase de consolidation doit être menée afin de le rendre plus fiable et plus agréable à utiliser.

Notons que nous avons souvent été gênés, au cours de nos réalisations, par l'inadéquation de FORTRAN à l'écriture de compilateurs. Ceci à cause de la pauvreté des notions que présente ce langage : pas de structures primitives puissantes, pas de types évolués (pointeurs, listes, piles,...). Nous avons dû, pourtant, adopter ce langage de programmation pour plusieurs raisons, dont une était, au début du projet, la cohérence avec nos partenaires.

La phase $\phi 1$ de compilation fait aujourd'hui, comme d'autres modules, l'objet de travaux d'intégration dans l'ensemble du système. Quelques difficultés ont été rencontrées dans cette opération, car lorsque l'analyse de la phase $\phi 1$ a été effectuée, celle de la gestion des données n'avait pratiquement pas commencé. Aussi certains de nos choix avaient-ils été guidés par le souci de "faire au mieux".

Enfin, d'autres travaux ont actuellement lieu afin d'intégrer le langage de commande de la simulation CASCADE au sein de l'Environnement Général de Simulation évoqué au chapitre I.

ANNEXE 1

GRAMMAIRE DE VÉRIFICATION DE LA SYNTAXE CASCADE

```
                                                    --
                                                    -- DEBUT DE LA DEFINITION DES REGLES
                                                    --

563     axiome
(1)             premier_ref_langage <er47> segment <er33> <lgede6> 'cfifi'

564     premier_ref_langage
(1)             'creflan' <er15> 'cidf' <fconiv> <er22> list_bib_opt

565     segment
(1)             def_de_langage
(2)             suite_de_descriptions <lgvcax>

566     def_de_langage
(1)             <finit> 'clanguage' <flang> <er2> 'cidf' <fnoml> <er3>
                     <lgede6> 'cbody' definitions <fdiscl> <er11>
                     end_language

567     end_language
(1)             'cendlanguage'
(2)             'cend'

568     definitions
(1)             ()
(2)             'cprive' <fprive> <er12> def_segments <er11> definitions
(3)             <fnpriv> <er12> def_segments definitions

569     def_segments
(1)             def_type
(2)             def_fonction
(3)             def_procedure
(4)             def_description_lan

570     suite_de_descriptions
(1)             def_description <er33> description_opt

571     description_opt
(1)             ()
(2)             suite_de_descriptions
(3)             ref_langage <er47> suite_de_descriptions

572     ref_langage
(1)             'creflan' <fdnila> <er15> 'cidf' <fconiv> list_bib_opt

573     list_bib_opt
(1)             <fptcr>
(2)             'cidf' <fnomcl> suite_list_bib_opt <fptcr>

574     suite_list_bib_opt
(1)             ()
(2)             'cvirg' <er22> 'cidf' <fnomcl> suite_list_bib_opt

575     def_description
(1)             <finit> <finaut> 'cdescription' <er9> 'cidf' <inicax>
                     <fnomd> <lgede6> <fglidi> en_tete <fglcdi> <faseri>
                     partie_assertions <er13> partie_corps
```

```
576     def_description_lan
(1)           <finit> 'cdescription' <er9> 'cidf' <fnomd> <lgede6> entier
                   <coprim> <fglidi> en_tete <fglcd1> <faseri>
                   partie_assertions <eri3> partie_corps

577     partie_corps
(1)           end_descr <fcoch2> <fincax> <genstx> <fcodis>
(2)           'cbody' <faserc> <er4> corps_de_description <fincax>
                   <genstx> <fcodis>

578     end_descr
(1)           'cenddescr'
(2)           'cend'

579     en_tete
(1)           ()
(2)           'cparg' <er1> suite_en_tete

580     suite_en_tete
(1)           attributs <er19> 'cpard' parametres_interface_opt
(2)           parametres_ou_interface

581     en_tete_opt
(1)           ()
(2)           'cparg' <er1> interfaces <er19> 'cpard'

582     partie_assertions
(1)           ()
(2)           def_assertion
(3)           'caccoladeg' def_assertion <er39> 'caccoladed'

583     def_assertion
(1)           'cassert' <fcoch1> <finass> inst_cond1 <fcoch5> <flaser>
                   assert_opt <eri4> 'cptvirg'

584     assert_opt
(1)           ()
(2)           'cvirg' <fcoch1> <finass> inst_cond1 <fcoch5> <flaser>
                   assert_opt

585     corps_de_description
(1)           decl_externes <er4> corps_de_description
(2)           decl_internes <er4> corps_de_description
(3)           def_type <er4> corps_de_description
(4)           def_fonction <er4> corps_de_description
(5)           def_procedure <er4> corps_de_description
(6)           def_description <er4> corps_de_description
(7)           def_global <er4> <idif> corps_de_description
(8)                                                          --IMAG_F et IMAG_D
              def_assertion <er4> corps_de_description
(9)           fin_corps_de_description <fauto4>

586     fin_corps_de_description
(1)           end_descr <fcoch2> <fincax>
(2)           partie_instructions <eri1> end_descr <fcoch2> <fincax>
```

```
587     partie_instructions
    (1)         'crelations' <fchaba> <fstrel>
                    suite_relation_ou_instructions

588     suite_relation_ou_instructions
    (1)         'cnelec' <restau> <fdxrlt> <locl> <er32> <fcoch3> relations
                                                              --LASSO,LASCAR,CASSANDRE
    (2)
                'celec' <restau> <felini> <fcoch3> <fdxrlt> <idif> <er32>
                    liste_instructions_elec
                                                              --IMAG_F ET IMAG_D

589     def_type
    (1)         'ctype' <er21> liste_idf <er17> 'cegal' <er34> moyen_def
                    <er14> 'cptvirg' def_type_opt

590     def_type_opt
    (1)         ()
    (2)         liste_idf <er17> 'cegal' <er34> moyen_def <er14> 'cptvirg'
                    def_type_opt

591     moyen_def
    (1)         'caccoladeg' <fppil> <er38> ensemble_def
    (2)         type_objet

592     ensemble_def
    (1)         constante_signee <fpve> ensemble_fini_opt <er39>
                    'caccoladed'
    (2)         'cidf' <fpveta> suite_ensemble_def

593     suite_ensemble_def
    (1)         liste_etat <cl> <er39> 'caccoladed'
    (2)         'cappartient' <er45> 'cidf' <er46> <ftpvap> 'cslash'
                    <fexpc> <er2> <fcoch3> expression <fcoch5> <fclapc>
                    <fdpil> <er39> 'caccoladed'

594     liste_etat
    (1)         <fvuen>
    (2)         'cvirg' <er2> 'cidf' <fpveta> liste_etat

595     ensemble_fini_opt
    (1)         <fvuen>
    (2)         'cvirg' <er38> constante_signee <fpve> ensemble_fini_opt

596     def_procedure
    (1)         'cprocedure' <er2> suite_procedure

597     suite_procedure
    (1)         'cidf' parametres_formels_proc partie_assertions <er3>
                    'cbody' operation_def_decl_proc operation_instructions
                    <er11> end_procedure
    (2)         nom_lang_progra <er70> 'cidf' descripteur_param_proc <er11>
                    end_procedure

598     descripteur_param_proc
    (1)         ()
```

```
      (2)           'cparg' <er71> element_descr_proc element_descr_proc_opt
                         <er19> 'cpard'

599      element_descr_proc_opt
      (1)           ()
      (2)           'cptvirg' <er71> element_descr_proc element_descr_proc_opt

600      element_descr_proc
      (1)           'cidf' liste_descr_proc

601      liste_descr_proc
      (1)           w_dim_opt suite_descr_opt

602      suite_descr_opt
      (1)           ()
      (2)           'cvirg' w_dim_opt suite_descr_opt

603      w_dim_opt
      (1)           'cw' dim_opt
      (2)           dim_opt

604      end_procedure
      (1)           'cendprocedure'
      (2)           'cend'

605      def_fonction
      (1)           'cfunction' <er2> suite_fonction

606      suite_fonction
      (1)           'cidf' dim_opt <er28> 'cidf' parametres_formels
                         partie_assertions corps_de_fonction <er72> 'creturn'
                         <er2> expression <er11> end_fonction
      (2)           nom_lang_progra <er73> 'cidf' dim_opt <er2> 'cidf'
                         descripteur_param_fonc <er11> end_fonction

607      nom_lang_progra
      (1)           'cfortran'
      (2)           'cpascal'
      (3)           'cada'

608      descripteur_param_fonc
      (1)           ()
      (2)           'cparg' <er15> element_descr_fonc element_descr_fonc_opt
                         <er19> 'cpard'

609      element_descr_fonc_opt
      (1)           ()
      (2)           'cptvirg' <er71> element_descr_fonc element_descr_fonc_opt

610      element_descr_fonc
      (1)           'cidf' liste_descr_fonc

611      liste_descr_fonc
      (1)           dim_opt suite_descr_fonc

612      suite_descr_fonc
```

```
      (1)          ()
      (2)          'cvirg' dim_opt suite_descr_fonc

613   end_fonction
      (1)          'cendfunction'
      (2)          'cend'

614   corps_de_fonction
      (1)          ()
      (2)          'cbody' operation_def_decl_fonc operation_instructions

615   parametres_formels_proc
      (1)          ()
      (2)          'cparg' <er34> declaration_objets_proc
                       parametres_formels_proc_opt <er19> 'cpard'

616   parametres_formels_proc_opt
      (1)          ()
      (2)          'cptvirg' <er34> declaration_objets_proc
                       parametres_formels_proc_opt

617   declaration_objets_proc
      (1)          type_objet <er71> liste_idf_dim_w

618   liste_idf_dim_w
      (1)          idfw dim_opt liste_idf_dim_w_opt

619   idfw
      (1)          'cidf'
      (2)          'cw' <er71> 'cidf'

620   liste_idf_dim_w_opt
      (1)          ()
      (2)          'cvirg' <er71> liste_idf_dim_w

621   parametres_formels
      (1)          ()
      (2)          'cparg' <er2> declarations_objets parametres_formels_opt
                       <er19> 'cpard'

622   parametres_formels_opt
      (1)          ()
      (2)          'cptvirg' <er2> declarations_objets parametres_formels_opt

623   operation_def_decl_proc
      (1)          operation_def_decl
      (2)          accesw

624   operation_def_decl_fonc
      (1)          operation_def_decl
      (2)          'cacces' <er15> liste_idf

625   operation_def_decl
      (1)          ()
      (2)          operation_definition operation_def_decl
      (3)          'cdeclare' <fdxdcl> <er2> liste_dec operation_def_decl
```

```
626     accesw
    (1)         'cacces' <er71> liste_idfw

627     liste_idfw
    (1)         idfw suite_liste_idfw

628     suite_liste_idfw
    (1)         ()
    (2)         'cvirg' <er71> liste_idfw

629     operation_definition
    (1)         def_type
    (2)         def_fonction
    (3)         def_procedure

630     operation_instructions
    (1)         'crelations' <fchaba> <fstrel>
                    suite_relation_ou_ope_instruction

631     suite_relation_ou_ope_instruction
    (1)         ()
    (2)         'cnelec' <restau> <fdxrlt> <er32> liste_instructions
                    <fdxrlt>
    (3)         'celec' <restau> <fdxrlt> <er32> liste_instructions_elec
                    <restau> <fdxrlt>

                                                    --  ====================  ATTRIBUTS FORMELS
632     attributs
    (1)         declarations_valeurs attributs_opt

633     attributs_opt
    (1)         ()
    (2)         'cptvirg' <er28> attributs

634     declarations_valeurs
    (1)         'cidf' <fdpil> <fvpatt> <fppil> <er15> liste_idfs <flatt>

635     liste_idfs
    (1)         'cidf' <fattri> dec_val liste_idfs_opt

636     dec_val
    (1)         <valatd>
    (2)         'cegal' constante_signee <valatr>

637     liste_idfs_opt
    (1)         ()
    (2)         'cvirg' <er15> liste_idfs

                                                    --  ====================
638     parametres_interface_opt
    (1)         ()
    (2)         'cparg' <er1> parametres_ou_interface

639     parametres_ou_interface
    (1)         'cparam' param <er19> 'cpard' en_tete_opt
```

```
      (2)          interfaces <er19> 'cpard'

640     param
      (1)          declaration_val_param param_opt

641     param_opt
      (1)          ()
      (2)          'cptvirg' <er28> param

642     declaration_val_param
      (1)          'cidf' <fdpil> <tvpara> <fppil> <er15> liste_param <flatt>

643     liste_param
      (1)          'cidf' <parame> decl_val liste_param_opt

644     liste_param_opt
      (1)          ()
      (2)          'cvirg' <er15> liste_param

645     decl_val
      (1)          <paramd>
      (2)          'cegal' constante_signee <valpar>

646     liste_idf
      (1)          'cidf' <ftypi> liste_idf_opt

647     liste_idf_opt
      (1)          ()
      (2)          'cvirg' <er15> liste_idf
                                                    --  ====================  ELEMENTS FORMELS

648     interfaces
      (1)          <fexint> <flistxi> interface <fdpil> fin_interfaces
                        <flistxi>

649     fin_interfaces
      (1)          ()
      (2)          'cptvirg' <er1> interface <fdpil> fin_interfaces

650     interface
      (1)          <fglc08> sens <er2> declarations_objets

651     sens
      (1)          'cin' <fsens>
      (2)          'cout' <fsens>
      (3)          'cinout' <fsens>
      (4)          'cnd' <fsens>

652     declarations_objets
      (1)          'cidf' <fsulpr> <er7> fin_type_porteuse

653     fin_type_porteuse
      (1)          <fextyp> <er10> liste_idf_dim
      (2)          fin_type_objet_prim <flty> <er10> liste_idf_dim
      (3)          <fextyp> 'clist' <flistx> <flintf2>
      (4)          <ftypii> <flintf> dim_opt liste_idf_dim_opt
```

```
654     liste_idf_dim
(1)           'cidf' <flintf> dim_opt liste_idf_dim_opt

655     dim_opt
(1)           ()
(2)           dimensions <fdimlo>

656     liste_idf_dim_opt
(1)           ()
(2)           'cvirg' <er27> liste_idf_dim

657     dimensions
(1)           'ccrochetgauche' <er23> <findim> bornes <er24>
                   'ccrochetdroit' <fexdim>

658     bornes
(1)           <fexpdm> <fcoch3> expression <fcoch5> <fbdm> <er26>
                   'cdeuxpoints' <er25> <fexpdm> <fcoch3> expression
                   <fcoch5> <fbdm> <felem> bornes_opt

659     bornes_opt
(1)           ()
(2)           'cptvirg' <er23> bornes

660     type_objet
(1)           'cidf' <fsulpr> fin_type_objet

661     fin_type_objet
(1)           <ftyp2>
(2)           fin_type_objet_prim <flty> <flnty>

662     fin_type_objet_prim
(1)           <fvertp> 'cparg' <er35> 'cidf' <ftpvap> <fverva> <er18>
                   'cvirg' constante <fvpvin> <er19> 'cpard'

663     constante
(1)           <locl>
(2)           constante_signee
(3)           'cidf' <cl>

                                                        --LASCAR CASSANDRE
                                                        --  ================== DECLARATIONS EXTERNES
664     decl_externes
(1)           'cexternal' <er2> liste_idf_ext <er14> 'cptvirg'

665     liste_idf_ext
(1)           'cidf' <fgmlex> <fglc00> <fgnext> <fglidi> en_tete <fglcdi>
                   liste_idf_ext_opt

666     liste_idf_ext_opt
(1)           ()
(2)           'cvirg' <er2> liste_idf_ext

                                                        --  ================== DECLARATIONS INTERNES
667     decl_internes
```

```
      (1)          'cuse' <fdxuse> <er9> liste_sous_unites
      (2)          'cdeclare' <fdxdcl> <er2> liste_dec
                                                          --  ==================  GRAPHE LOCAL

668     liste_sous_unites
      (1)          sous_unites <er14> 'cptvirg' sous_unites_opt

669     sous_unites
      (1)          'cidf' <fstart1> unites

670     unites
      (1)          'cdesstand' <restau> <fgmldr> sous_unites_param <fgmldg>
                       <er35> exemplaire <fgverx> exemplaire_opt
      (2)          'cdesnode' <restau> <fgmldr> sous_unites_param <fgmldg>
                       <er35> exemplaire_node <fgverx> exemplaire_node_opt
      (3)          'cdesnand' <restau>

671     sous_unites_opt
      (1)          ()
      (2)          liste_sous_unites

672     exemplaire_opt
      (1)          ()
      (2)          'cvirg' <er35> exemplaire <fgverx> exemplaire_opt

673     sous_unites_param
      (1)          ()
      (2)          'cparg' <er35> suite_sous_unites_param

674     suite_sous_unites_param
      (1)          fin_sous_unites param_opt1
      (2)          fin_param

675     fin_sous_unites
      (1)          <fgpatf> <fexc12> attributs_effectifs <fgvaef> <er19>
                       'cpard'

676     exemplaire
      (1)          'cidf' <fgmlun> suite_exemplaire

677     suite_exemplaire
      (1)          ()
      (2)          <fdec18> dimensions <fgmlu1>
      (3)          'cparg' <fgpelf> <fcoch3> elements_interface <fgveef>
                       <fgmlu2> <er19> 'cpard'

678     param_opt1
      (1)          ()
      (2)          fin_param

679     fin_param
      (1)          <fgparf> 'cparam' <fexc12> param_effectifs <fgvpaf> <er19>
                       'cpard'

680     param_effectifs
      (1)          <fglc11> <fgpvid> param_effectifs_opt
```

```
      (2)           <fcoch3> <er35> expression <fcoch5> <fglc11> <fgptat>
                         param_effectifs_opt

681      param_effectifs_opt
      (1)           ()
      (2)           'cvirg' <fglc12> param_effectifs

682      elements_interface
      (1)           'cidf' <fstart> suite_ident1
      (2)           constante_generale <fglc04> <fgvte3> <fgmlco> <fcoch5>
                         elements_interface_opt
      (3)           elements_interface_opt

683      suite_ident1
      (1)           'cboite' <restau> <boite> suite_ident11
      (2)           'cobjet' <restau> suite_ident12

684      suite_ident12
      (1)           <fglc04> <fgvte1> <fgmlco> indexation_elements_interface
                         <fgvdin> <fcoch5> elements_interface_opt

685      suite_ident11
      (1)           <fcoch1> <fglc10> <fgreid> indexation <er80> 'cpoint'
                         <fcoch1> <er35> 'cidf' <fglc04> <fgvte2>
                         indexation_elements_interface <fcoch5>
                         elements_interface_opt

686      elements_interface_opt
      (1)           ()
      (2)           'cvirg' <fcoch3> <fglc07> elements_interface

687      exemplaire_node_opt
      (1)           ()
      (2)           'cvirg' <er35> exemplaire_node <fgverx> exemplaire_node_opt

688      exemplaire_node
      (1)           'cidf' <fgmlun> suite_exemplaire_node

689      suite_exemplaire_node
      (1)           ()
      (2)           <fdecl8> dimensions <fgmlu1>
      (3)           'cparg' <fpelf1> <fcoch3> elements_interface_node <fgmlu2>
                         <er19> 'cpard'

690      elements_interface_node
      (1)           'cidf' <fstart> suite_ident1_node
      (2)           constante_generale <fglc04> <fvte3> <fgmlco> <fcoch5>
                         elements_interface_node_opt
      (3)           elements_interface_node_opt

691      suite_ident1_node
      (1)           'cboite' <restau> <boite> suite_ident11_node
      (2)           'cobjet' <restau> suite_ident12_node

692      suite_ident12_node
      (1)           <fglc04> <fvte1> <fgmlco> indexation_elements_interface
```

```
                    <fgvdin> <fcoch5> elements_interface_node_opt

693     suite_identi1_node
(1)             <fcoch1> <fglc10> <fgreid> indexation <er80> 'cpoint'
                    <fcoch1> <er35> 'cidf' <fglc04> <fvte2>
                    indexation_elements_interface <fcoch5>
                    elements_interface_node_opt

694     elements_interface_node_opt
(1)             ()
(2)             'cvirg' <fcoch3> <fglc07> elements_interface_node

695     indexation_elements_interface
(1)             <findv1>
(2)             <fieicg> 'ccrochetgauche' <fcoch1> index_elem_interface
                    <er24> 'ccrochetdroit' <fcoch1> <fieicd>

696     index_elem_interface
(1)             dim_selectionnee index_elem_interface_opt

697     index_elem_interface_opt
(1)             ()
(2)             'cptvirg' <fcoch1> <fieiptv> index_elem_interface

698     dim_selectionnee
(1)             <fieivid>
(2)             <finexp> expression <fieib1> dim_selectionnee_opt

699     dim_selectionnee_opt
(1)             ()
(2)             'cdeuxpoints' <fcoch1> <fiei2p> <finexp> <er35> expression
                    <fieib2>

700     index
(1)             <fchgdx> 'ccrochetgauche' <fcoch1> indices <er24>
                    'ccrochetdroit' <fcoch1> <fchddx>

701     indices
(1)             select_dim indices_opt

702     indices_opt
(1)             ()
(2)             'cptvirg' <fptvdx> <fcoch1> indices

703     select_dim
(1)             <fviddx>
(2)             <finexp> expression select_dim_opt fin_select_dim

704     select_dim_opt
(1)             ()
(2)             'cdeuxpoints' <fcoch1> <f2pdx> <finexp> <er35> expression

705     fin_select_dim
(1)             ()
(2)             'cvirg' <fvirdx> <fcoch1> select_dim
```

```
706     attributs_effectifs
(1)             <fglc09> <fgavid> attributs_eff_opt
(2)             <fcoch3> expression <fcoch5> <fglc09> <fgvtat>
                    attributs_eff_opt

707     attributs_eff_opt
(1)             ()
(2)             'cvirg' <fglc08> <er35> attributs_effectifs

708     liste_dec
(1)             'cidf' <er7> <fsulpr> fin_list_dec

709     fin_list_dec
(1)             fin_type_objet_prim <flty> <fdecl2> <er10>
                    elements_declares <fdpil> <er14> 'cptvirg'
                    liste_dec_opt
(2)             <fdecli> elements_declares <fdpil> <er14> 'cptvirg'
                    liste_dec_opt
(3)             <ftypi2> <fdecl8> dim_opt2 <fdecl5> <er14>
                    suite_elements_declares <fdpil> <er14> 'cptvirg'
                    liste_dec_opt

710     elements_declares
(1)             'cidf' <fdecl8> dim_opt2 <fdecl5> <er14>
                    suite_elements_declares

711     dim_opt2
(1)             <fdecl3>
(2)             dimension2 <fdecl4>

712     dimension2
(1)             'ccrochetgauche' <er23> <findim> bornes <er24>
                    'ccrochetdroit'

713     liste_dec_opt
(1)             ()
(2)             liste_dec

714     suite_elements_declares
(1)             'cegal' <er35> <fcoch3> expression <fdecl6> <fasdcl>
                    <fcoch5> <fdecl9> elements_declares_opt
(2)             <fdecl7> elements_declares_opt

715     elements_declares_opt
(1)             ()
(2)             'cvirg' <er10> elements_declares

716     inst_cond1
(1)             'cffix' <fcoch1> <er76> cond_niveau1 <er74> 'cthen'
                    <fcoch1> <er2> liste_inst_cond_niveau2
                    sinon_niveau1_opt <er75> endif <fcoch1>
(2)             inst_cond2

717     sinon_niveau1_opt
(1)             ()
(2)             'celse' <fcoch1> <er2> liste_inst_cond_niveau2
```

```
718     liste_inst_cond_niveau2
    (1)         inst_cond2 suite_cond_niveau2_opt

719     suite_cond_niveau2_opt
    (1)         ()
    (2)         'cvirg' <fcoch1> <er2> liste_inst_cond_niveau2

720     cond_niveau1
    (1)         segment1 fin_segment1_opt

721     fin_segment1_opt
    (1)         ()
    (2)         'cunion' <fcoch1> <er76> segment1 fin_segment1_opt

722     segment1
    (1)         segment2 fin_segment2_opt

723     fin_segment2_opt
    (1)         ()
    (2)         'cinter' <fcoch1> <er76> segment2 fin_segment2_opt

724     segment2
    (1)         fenetre_fixe
    (2)         'csauf' <fcoch1> <er2> fenetre_fixe

725     fenetre_fixe
    (1)         prept <fcoch1> <er2> exp_date
    (2)         'cparg' <fcoch1> <er76> cond_niveau1 <er19> 'cpard'
                    <fcoch1>

726     prept
    (1)         'cavant'
    (2)         'cenetavant'
    (3)         'capres'
    (4)         'cenetapres'

727     exp_date
    (1)         <fcoda1> expression <fclaas>
    (2)         'clafois' <fcoch1> <fcoda2> <er2> expression <fclaas>
                    def_orig_opt <er77> 'cque' <fcoch1> <er78> exp_event

728     def_orig_opt
    (1)         ()
    (2)         'capartir' <fcoch1> <er2> exp_date

729     inst_cond2
    (1)         'cfrepet' <fcoch1> <er79> cond_niveau2 <er74> 'cthen'
                    <fcoch1> <er2> liste_relations sinon_niveau2_opt
                    <er75> endif <fcoch1> <er2>
    (2)         <fcorel> expression <fclaas>

730     sinon_niveau2_opt
    (1)         ()
    (2)         'celse' <fcoch1> <er2> liste_relations
```

```
731     cond_niveau2
    (1)         segment3 fin_segment3_opt

732     fin_segment3_opt
    (1)         ()
    (2)         'cunion' <fcoch1> <er79> segment3 fin_segment3_opt

733     segment3
    (1)         segment4 fin_segment4_opt

734     fin_segment4_opt
    (1)         ()
    (2)         'cinter' <fcoch1> <er79> segment4 fin_segment4_opt

735     segment4
    (1)         fenetre_repetable
    (2)         'csauf' <fcoch1> <er79> fenetre_repetable

736     fenetre_repetable
    (1)         'cparg' <fcoch1> <er79> cond_niveau2 <er19> 'cpard'
                    <fcoch1>
    (2)         'cchaque' <fcoch1> <er78> exp_event def_orig_opt duree_opt
    (3)         'ctousles' <fcoch1> <fcodur> <er2> expression <fclaas>
                    def_orig_opt duree_opt
    (4)         'cen' <fcoch1> <er2> exp_date duree_opt

737     duree_opt
    (1)         ()
    (2)         'cpendant' <fcoch1> <fcodur> <er2> expression <fclaas>
    (3)         'cto' <fcoch1> <er78> exp_event

738     exp_event
    (1)         terme_event fin_terme_event_opt

739     fin_terme_event_opt
    (1)         ()
    (2)         'cpuis' <fcoch1> <er78> terme_event fin_terme_event_opt
    (3)         'ccommesi' <fcoch1> <er78> terme_event fin_terme_event_opt
    (4)         'cetdeplus' <fcoch1> <er78> terme_event fin_terme_event_opt

740     terme_event
    (1)         'cparg' <fcoch1> <er78> exp_event <er19> 'cpard' <fcoch1>
    (2)         'cmonte' <fcoch1> <fcoeve> <er2> expression <fclaas>
    (3)         'cdescend' <fcoch1> <fcoeve> <er2> expression <fclaas>

741     liste_relations
    (1)         <fcorel> expression <fclaas> liste_relations_opt

742     liste_relations_opt
    (1)         ()
    (2)         'cvirg' <fcoch1> <er2> liste_relations

743     relations
    (1)         automate
    (2)         graphe
    (3)         <razif> liste_instructions <er58> suite_relations
```

```
744     suite_relations
(1)             ()
(2)             automate
(3)             graphe

745     liste_instructions
(1)             instruction <fauto6> <inconi> liste_instructions_opt
(2)             bloc_horloge <fauto6> <inconi> suite_lst_inst_opt

746     bloc_horloge
(1)             'cptexclam' <dchor> <fcoch1> <fcotxh> <fdexco> <er2>
                    expression <fexhor> <er16> 'cptexclam' <fcoch1> <er2>
                    liste_chargements <er68> 'cptvirg' <fchor> <fcoch1>

747     liste_chargements
(1)             chargement chargement_opt

748     chargement_opt
(1)             ()
(2)             'cvirg' <fcoch1> <er2> liste_chargements

749     chargement
(1)             'cif' <cl> <fcoch1> <er2> expression <er50> 'cthen'
                    <fcoch1> <er2> liste_chargements else_charge_opt
                    <er75> endif <fcoch1>
(2)             'ccase' <cl> <fcoch1> <er2> expression <er37> 'cis'
                    <fcoch1> <er32> chargements_choisis else_charge_opt
                    <er52> endcase <fcoch1>
(3)             'cover' <cl> <fcoch1> <fbpoex> <er32> 'cidf' <fbpoin>
                    <fcoch1> <er56> ensemble_parcouru <er40> 'crepeat'
                    <fcoch1> <er2> liste_chargements <er5> endover <fbpaf>
                    <fcoch1>
(4)             chargement_simple

750     else_charge_opt
(1)             ()
(2)             'celse' <fcoch1> <er2> liste_chargements

751     chargements_choisis
(1)             etiquettes <er26> 'cdeuxpoints' <fcoch1> <er2>
                    liste_chargements chargements_choisis_opt

752     chargements_choisis_opt
(1)             ()
(2)             'cptvirg' <fcoch1> <er32> chargements_choisis

753     chargement_simple
(1)             'cidf' <fcoch1> <cl> <fchrg1> <er87> suite_chargement
(2)             'cpoint' <fcoch1> <cl> <er32> 'cidf' <fcoch1> indexation
                    <er80> 'cpoint' <fcoch1> <er32> 'cidf' <fcoch1>
                    indexation_patte <er87> 'cchargement' <fcoch1>
                    <fexchg> <er2> expression <fchrg2>
(3)             'cp1' <l> <fcoch1> <er53> 'cparg' <fcoch1> <er35> 'cidf'
                    <fcoch1> <fchrg3> indexation <er19> 'cpard' <fcoch1>
(4)                                                                    --LASCAR
```

```
                'cmi' <l> <fcoch1> <er53> 'cparg' <fcoch1> <er35> 'cidf'
                    <fcoch1> <fchrg3> indexation <er19> 'cpard' <fcoch1>
(5)                                                                  --LASCAR
                'cinit' <fcoch1> <l> <er53> 'cparg' <fcoch1> <er35> 'cidf'
                    <fcoch1> <fchrg3> indexation <er19> 'cpard' <fcoch1>
(6)             'ccharger' <cl> <fcharg> <er2> 'cidf' <fauto7> <fchrg5>

754     suite_chargement
(1)             <findv1> 'cchargement' <fcoch1> <er2> <fexchg> expression
                    <fchrg2>
(2)             'cparg' <fcoch1> <er2> parametres_effectifs <er19> 'cpard'
                    <fcoch1>
(3)             index 'cchargement' <fcoch1> <er2> <fexchg> expression
                    <fchrg2>

755     suite_lst_inst_opt
(1)             <empni> <razni>
(2)             liste_instructions

756     liste_instructions_opt
(1)             <empni> <razni>
(2)             'cvirg' <fcoch1> <er32> liste_instructions

757     instruction
(1)             'cif' <dcif> <cl> <fcoch1> <emifca> <fcotxf> <fcobci>
                    <fdexco> <er2> expression <fexif> <fvecl3> <er50>
                    'cthen' <incnal> <empn> <ffexco> <fcoch1> <er32>
                    liste_instructions else_opt <er75> endif <fcif>
                    <credal> <fifca> <fcoch1>
(2)             'ccase' <dccase> <cl> <fcoch1> <emifca> <fcotxc> <fcobci>
                    <fdexco> <er2> expression <fexcas> <fvecl3> <er37>
                    'cis' <ffexco> <fcoch1> <er32> choix choix_opt <er52>
                    endcase <fccase> <fcoch1> <credal> <fifca>
(3)             'cover' <dcbpo> <cl> <fcoch1> <fbpoex> <er32> 'cidf'
                    <fbpoin> <fcoch1> <er56> ensemble_parcouru <er40>
                    'crepeat' <er2> liste_instructions <er5> endover
                    <fcbpo> <fbpaf> <fcoch1>
(4)             'cidf' <fcoch1> <fstart> <cl> suite_ident

758     suite_ident
(1)             'cboite' <restau> <boite> indexation <vdintr> <er80>
                    'cpoint' <fcoch1> <er35> 'cidf' <fconx1> <fcoch1>
                    indexation_patte <er31> 'cconnexion' <fcoch1> <fexc16>
                    <er32> expression <fconx3>
(2)             'cobjet' <restau> <fconx1> indexation <er31> connexion

759     indexation
(1)             <findv1> <freti5>
(2)             index <freti5>

760     indexation_patte
(1)             <findv2>
(2)             index

761     else_opt
(1)             ()
```

```
      (2)          'celse' <fcoch1> <incnal> <empn0> <er32> liste_instructions

762     endif
      (1)          'cendif'
      (2)          'cend'

763     endcase
      (1)          'cendcase'
      (2)          'cend'

764     endover
      (1)          'cendover'
      (2)          'cend'

765     choix
      (1)          etiquettes <er26> 'cdeuxpoints' <fcoch1> <incnal> <er32>
                       liste_instructions

766     choix_opt
      (1)          else_opt
      (2)          'cptvirg' <fcoch1> <er32> choix <er52> choix_opt

767     etiquettes
      (1)          constante_enumeree <fcoch1> <fvetiq> <incnet>
                       suite_etiquettes

768     suite_etiquettes
      (1)          <empn> <raznet>
      (2)          'cvirg' <fcoch1> <er32> etiquettes

769     connexion
      (1)          'cparg' <fcoch1> <er2> <fexc17> connexions_interface <er19>
                       'cpard' <fcoch1>
      (2)          'cconnexion' <c1> <fcoch1> <fexc16> <er32> expression
                       <fconx3>
                                                                  -- LASCAR CASSANDRE

770     connexions_interface
      (1)          expression connexions_interface_opt
      (2)          connexions_interface_opt

771     connexions_interface_opt
      (1)          ()
      (2)          'cvirg' <fcoch1> <er2> connexions_interface

772     automate
      (1)          'cdeuxpoints' <fcoch1> <c1> <er57> <fauto1> 'cidf' <fauto2>
                       <er58> 'cdeuxpoints' <fcoch1> <er32> corps_automate
                       automate_opt

773     automate_opt
      (1)          <fauto3>
      (2)          'cdeuxpoints' <fcoch1> <er57> 'cidf' <fauto2> <er58>
                       'cdeuxpoints' <fcoch1> <er32> corps_automate
                       automate_opt
```

```
774     corps_automate
(1)             liste_instructions
(2)             'cbegin' <ci> <er30> decl_internes <er81>
                     partie_instructions <er11> 'cend'

                                                        --  ====================  EXPRESSION
775     expression
(1)             <fxinit> exp2 fin_expression

776     fin_expression
(1)             op1 <fcoch1> <fxemp1> exp2 <fevddx> fin_expression
(2)             ()

777     exp2
(1)             exp3 fin_exp2

778     fin_exp2
(1)             ()
(2)             op2 <fcoch1> <fxemp1> exp3 <fevddx> fin_exp2

779     exp3
(1)             exp4 fin_exp3

780     fin_exp3
(1)             ()
(2)             op3 <fcoch1> <fxemp1> exp4 <fevddx> fin_exp3

781     exp4
(1)             exp5 fin_exp4

782     fin_exp4
(1)             ()
(2)             op4 <fcoch1> <fxemp1> exp5 <fevddx> fin_exp4

783     exp5
(1)             exp6 fin_exp5

784     fin_exp5
(1)             ()
(2)             op5 <fcoch1> <fxemp1> exp6 <fevddx> fin_exp5

785     exp6
(1)             exp7 fin_exp6

786     fin_exp6
(1)             ()
(2)             op6 <fcoch1> <fxemp1> exp7 <fevddx> fin_exp6

787     exp7
(1)             exp8 fin_exp7

788     fin_exp7
(1)             op7 <fcoch1> <fxemp1> exp8 <fevddx> fin_exp7
(2)             ()

789     exp8
```

Grammaire du Langage gramgene

```
      (1)          exp9 fin_exp8

790     fin_exp8
      (1)          op8 <fcoch1> <fxemp1> exp9 <fevddx2> fin_exp8
      (2)          ()

791     exp9
      (1)          op9 <er2> exp9 <fxpreu>
      (2)          atome
                                                    --  ====================  OPERATEURS BINAIRES

792     op1
      (1)          'cou'
      (2)          'cnou'

793     op2
      (1)          'cxor'
      (2)          'ceqv'

794     op3
      (1)          'cet'
      (2)          'cnet'

795     op4
      (1)          'cinf'
      (2)          'csup'
      (3)          'cegalinf'
      (4)          'csupegal'
      (5)          'cegal'
      (6)          'cnonegal'

796     op5
      (1)          'cplus'
      (2)          'cmoins'

797     op6
      (1)          'cmult'
      (2)          'cslash'
      (3)          'cmod'

798     op7
      (1)          'cpuissance'
      (2)          'cpuisselec'

799     op8
      (1)          'cconcat'
                                                    --  ====================  OPERATEURS UNAIRES

800     op9
      (1)          'cpuissance' <funair> <fcodun> <clas> <fcoch1> <fxemp1>
      (2)                                           --CASSANDRE_AS et LASCAR_AS
                   'cdescend' <fcodun> <clas> <fcoch1> <fxemp1>
      (3)                                           --CASSANDRE_AS et LASCAR_AS
                   'cplus' <fcodun> <fcoch1> <fxemp1>
      (4)          'cmoins' <funair> <fcodun> <fcoch1> <fxemp1>
      (5)          'cnon' <fcodun> <fcoch1> <fxemp1>
```

```
      (6)          'cdollar' <fcodun> <lol> <fcoch1> <fxemp1>
      (7)                                                          --LASSO et LASCAR
                   'crang' <fcodun> <fcoch1> <fxemp1>
      (8)          'crangx' <fcodun> <fcoch1> <fxemp1>
      (9)          'cintval' <fcodun> <lol> <fcoch1> <fxemp1>
      (10)                                                         --LASSO et LASCAR
                   'cabs' <fcodun> <fcoch1> <fxemp1>
      (11)         'cpourcent' <fcodun> <clas> <fret1> <fopret> <fcoch1>
                        <fxemp1> <er32> <fprop1> expression <fprop2> <er16>
                        'cptexclam' <fcoch1> <fret5>
      (12)                                                         --CASS_AS et LASCAR_AS
                   'cfan' <fcodun> <fcoch1> <fxemp1> <er32> <fprop1>
                        expression <fprop2> <er16> 'cptexclam' <fcoch1>
      (13)         'cconvert' <fcodun> <lol> <fcoch1> <fxemp1> <er32> <fprop1>
                        expression <fprop2> <er16> 'cptexclam' <fcoch1>
      (14)                                                         -- LASSO et LASCAR
                   'cbinval' <fcodun> <lol> <fcoch1> <fxemp1> <er32> <fprop1>
                        expression <fprop2> <er16> 'cptexclam' <fcoch1>
      (15)                                                         -- LASSO et LASCAR
                   'creduc' <fcodun> <fcoch1> <fxemp1> <er20> op_bin_reduc
                        <fcodun> <fxemp1>
      (16)         'ctransp' <fcodun> <fcoch1> <fxemp1> <er16> trans_opt

801     op_bin_reduc
      (1)          op1 <fcoch1>
      (2)          op2 <fcoch1>
      (3)          op3 <fcoch1>
      (4)          op5 <fcoch1>
      (5)          op6 <fcoch1>

802     trans_opt
      (1)          ()
      (2)          'cptexclam' <fcoch1> <er32> expression <er44> 'cvirg'
                        <fcoch1> <er32> expression <er16> 'cptexclam' <fcoch1>

803     atome
      (1)          'cparg' <fcoch1> <er32> expression <er19> 'cpard' <fcoch1>
                        indexation
      (2)          'cif' <fcoch1> <er32> expression <er50> 'cthen' <fcoch1>
                        <er32> expression <er6> 'celse' <fcoch1> <er32>
                        expression <er75> endif <fcoch1>
      (3)          'ccase' <fcoch1> <er32> expression <er37> 'cis' <fcoch1>
                        <er32> exp_choisies <er6> 'celse' <fcoch1> <er32>
                        expression <er52> endcase <fcoch1>
      (4)          constante_pos <fret6> <fcstim> <fclass> <fret7> <fcoch1>
                        <fel21>
      (5)          'cderiv' <fel8> <if> <er2> idf_elec <fel5>
      (6)                                                          -- IMAG_F
                   'cv' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1> <er32>
                        idf_elec <er18> <fel10> 'cvirg' <fel12> <fcoch1>
                        <er32> idf_elec <fel10> <er19> 'cpard' <fcoch1>
                        <fel14>
      (7)          'cmodulo' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1>
                        <er2> expression <er18> 'cvirg' <fel12> <fcoch1> <er2>
                        expression <er19> 'cpard' <fcoch1> <fel9>
      (8)          'cmax' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1> <er2>
```

```
                    expression <er18> 'cvirg' <fel12> <fcoch1> <er2>
                    expression <er19> 'cpard' <fcoch1> <fel9>
(9)             'cmin' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1> <er2>
                    expression <er18> 'cvirg' <fel12> <fcoch1> <er2>
                    expression <fel10> <er19> 'cpard' <fcoch1> <fel9>
(10)            'ci' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1> <er32>
                    idf_elec <fel11> <er39> 'cpard' <fcoch1> <fel14>
(11)            'cexp' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1> <er2>
                    expression <er19> 'cpard' <fcoch1> <fel13>
(12)            'clog' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1> <er2>
                    expression <er19> 'cpard' <fcoch1> <fel13>
(13)            'clog10' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1>
                    <er2> expression <er19> 'cpard' <fcoch1> <fel13>
(14)            'csin' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1> <er2>
                    expression <er19> 'cpard' <fcoch1> <fel13>
(15)            'ccos' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1> <er2>
                    expression <er19> 'cpard' <fcoch1> <fel13>
(16)            'ctan' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1> <er2>
                    expression <er19> 'cpard' <fcoch1> <fel13>
(17)            'csqrt' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1>
                    <er2> expression <er19> 'cpard' <fcoch1> <fel13>
(18)            'carcsin' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1>
                    <er2> expression <er19> 'cpard' <fcoch1> <fel13>
(19)            'carctg' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1>
                    <er2> expression <er19> 'cpard' <fcoch1> <fel13>
(20)            'cvabs' <fel8> <fcoch1> <idif> <er53> 'cparg' <fcoch1>
                    <er2> expression <er19> 'cpard' <fcoch1> <fel13>
(21)            'cidf' <fcoch1> <fstart> suite_idfexp

804   suite_idfexp
(1)             'cboite' <restau> indexation <er80> 'cpoint' <fcoch1> <er2>
                    'cidf' <fcoch1> indexation_patte
(2)             'cobjet' <restau> <fret8> <fopexp> <fclass> <fret9>
                    <fretar> <fel10> <fel4> suite_atome

805   suite_atome
(1)             'cparg' <fcoch1> <er2> parametres_effectifs <er19> 'cpard'
                    <fcoch1>
(2)             indexation

806   idf_elec
(1)             reel <fcoch1> <fel3>
(2)             'cidf' <fcoch1> <fstart> suite_idf_elec

807   suite_idf_elec
(1)             'cboite' <restau> <fel17> <er80> 'cpoint' <fcoch1> <er2>
                    'cidf' <fcoch1> <fel18>
(2)             'cobjet' <restau>

808   exp_choisies
(1)             etiquettes <er26> 'cdeuxpoints' <fcoch1> <er32> expression
                    <er6> exp_choisies_opt

809   exp_choisies_opt
(1)             ()
(2)             'cptvirg' <fcoch1> <er32> exp_choisies
```

```
810     constante_generale
(1)             constante_signee
(2)             tableau

811     tableau
(1)             <fconst> 'ccrochetgauche' <er23> <fcpdi1> constante_signee
                     <flgdi1> tableau_opt <er24> 'ccrochetdroit'

812     tableau_opt
(1)             ()
(2)             'cdeuxpoints' <fcpdi2> <flgdi2> deuxpts_opt <er38>
                     constante_signee tableau_opt

813     deuxpts_opt
(1)             ()
(2)             'cdeuxpoints' <fcpdi3> deuxpts_opt <fcp2pt> <flgdi3>

814     constante_signee
(1)             signe <er38> reelouentier
(2)             'clogique' <clloid>
(3)                                                     -- LASCAR CASSANDRE LASSO IMAG_D
                'clitteral' <lol>

                                                        -- LASSO et LASCAR
815     reelouentier
(1)             reel
(2)             entier_pos

816     constante_pos
(1)             entier_pos
(2)             reel
(3)             'clogique' <clid>
(4)             'clitteral' <lol>
(5)             tableau

817     entier
(1)             signe <er38> entier_pos

818     reel
(1)             'creel' <idif>
(2)             'creeldecexp' <idif>
(3)             'creelexp' <idif>

819     entier_pos
(1)             'centier'
(2)             'coctal'
(3)             'chexa'
(4)             'cbinaire'

820     signe
(1)             ()
(2)             'cplus'
(3)             'cmoins'

821     constante_enumeree
```

```
       (1)          entier <valent>
       (2)          'clitteral' <lol> <valcar>
       (3)          'clogique' <clid> <vallo>
       (4)          'cidf' <valcar>

822      ensemble_parcouru
       (1)          'cfrom' <fbpc4> <fbpep1> <er32> expression <fbpof1>
                        step_opt <er54> 'cto' <er32> expression <fbpof2>
                        <fbpopb> <fbpep0>
       (2)          'cin' <fbpopd> <er2> 'cidf' <fbpc5> <fbpoen> <fbpopd>
       (3)          'cin' <er55> ensemble_fini <fbpopd>

823      ensemble_fini
       (1)          'caccoladeg' <er38> <fbpc6> liste_entier <er39>
                        'caccoladed'

824      liste_entier
       (1)          entier <fbpob1> liste_entier_opt

825      liste_entier_opt
       (1)          ()
       (2)          'cvirg' <er38> entier <fbpoef> liste_entier_opt

826      step_opt
       (1)          <fbpopd>
       (2)          'cstep' <er32> expression <fbpopa>

827      parametres_effectifs
       (1)          <fcoch3> expression <fcoch5> parametres_effectifs_opt

828      parametres_effectifs_opt
       (1)          ()
       (2)          'cvirg' <er2> parametres_effectifs
                                                                    --  LASSO

829      graphe
       (1)          'cpint' <lo> <er32> <signae> exp_de_controle <er83> 'cpint'
                        <lo> <er32> transition graphe_opt

830      exp_de_controle
       (1)          expression_de_controle
       (2)          'clogique' <fbool1>

831      graphe_opt
       (1)          ()
       (2)          graphe

832      expression_de_controle
       (1)          signal_de_controle fin_expression_de_controle

833      fin_expression_de_controle
       (1)          ()
       (2)          'cet' <er32> expression_de_controle

834      signal_de_controle
       (1)          'cidf' <fcoch1> <fstart> suite_idf_signal
```

```
835     suite_idf_signal
(1)             'cobjet' <restau> <fopexp> <fclass> <tyctrl> <vsigna>
                    <cpent1>
(2)             'cboite' <restau> indexation 'cpoint' <fcoch1> 'cidf'
                    <tyctrl> <vsigna> <fcoch1> <er80> 'cpoint'
                    indexation_patte

836     transition
(1)             bloc_opt delai_opt <primi0> primitive_opt <cpent0> <er84>
                    <cpsor0> <signas> 'cvalider' <er53> sortie_transition
                    <versor> 'cptvirg'

837     bloc_opt
(1)             ()
(2)             <er2> liste_affectations <er11> 'cend'

838     liste_affectations
(1)             affectation affectation_opt

839     affectation_opt
(1)             ()
(2)             'cvirg' <er2> liste_affectations

840     affectation
(1)             'cif' <er2> expression <er74> 'cthen' <er2>
                    liste_affectations si_affect_opt <er75> endif
(2)             'ccase' <er2> expression <er37> 'cis' <er32> affect_choisis
                    si_affect_opt <er52> endcase
(3)             'cover' <er32> 'cidf' <er56> ensemble_parcouru <er40>
                    'crepeat' liste_affectations <er5> endover
(4)             affectation_simple

841     si_affect_opt
(1)             ()
(2)             'celse' <er2> liste_affectations

842     affect_choisis
(1)             etiquettes <er26> 'cdeuxpoints' bloc_opt affect_choisis_opt

843     affect_choisis_opt
(1)             ()
(2)             'cptvirg' <er32> affect_choisis

844     affectation_simple
(1)             'cidf' <fcoch1> <fstart> suite_aff_simple

845     suite_aff_simple
(1)             'cobjet' <restau> suite_affectation
(2)             'cboite' <restau> indexation <er80> 'cpoint' <fcoch1>
                    <er32> 'cidf' indexation_patte <er86> 'caffect' <er2>
                    expression

846     suite_affectation
(1)             ()
(2)             'cparg' <er2> parametres_effectifs <er19> 'cpard'
```

```
      (3)          index <er86> 'caffect' <er2> expression
      (4)          'caffect' <er2> expression

847     delai_opt
      (1)          ()
      (2)          'cdelay' <er2> expression <typent>

848     primitive_opt
      (1)          ()
      (2)          'cselon' <primi1> <er2> expression <tyscab>
      (3)          choisir <primi2> <er32> liste_sig_ctl
      (4)          'cindex' <primi3> expression <tyscae> suite_index_opt

849     choisir
      (1)          'cchoix'
      (2)          'cchoixp'
      (3)          'cchoixr'

850     suite_index_opt
      (1)          ()
      (2)          'cfrom' <fcoch1> <er32> expression <tyscae> <er82> 'cto'
                        <fcoch1> <er32> expression <tyscae>

851     liste_sig_ctl
      (1)          signal_de_controle <cpent1> liste_sig_ctl_opt

852     liste_sig_ctl_opt
      (1)          ()
      (2)          'cvirg' <er32> liste_sig_ctl

853     sortie_transition
      (1)          position_sortie sortie_transition_opt

854     position_sortie
      (1)          'cnone' <cpsor1>
      (2)          signal_de_controle <cpsor1>
      (3)          'cparg' <er32> liste_sig_ctl <er19> 'cpard' <cpsor1>

855     sortie_transition_opt
      (1)          ()
      (2)          'cvirg' sortie_transition

--          ensemble des regles des langages IMAG_D et IM

AG_F
856     def_global
      (1)          'cglobal' <idif> <er49> liste_globale <er14> 'cptvirg'

857     liste_globale
      (1)          'cidf' liste_globale_opt

858     liste_globale_opt
      (1)          ()
      (2)          'cvirg' <er2> liste_globale

859     liste_instructions_elec
      (1)          <felini> instruction_elec <inconi>
```

```
                    liste_instructions_elec_opt
(2)           bloc_equations liste_inst_elec_opt

860     liste_inst_elec_opt
(1)           <empni> <razni>
(2)           liste_instructions_elec

861     liste_instructions_elec_opt
(1)           <empni> <razni>
(2)           'cvirg' <fcoch1> <er32> liste_instructions_elec

862     instruction_elec
(1)           'cif' <fcoch1> <emifca> <fcotxf> <fcobci> <fdexco> <er2>
                    expression <fexif> <fvecl3> <er74> 'cthen' <incnal>
                    <empn> <ffexco> <er32> <fcoch1>
                    liste_instructions_elec else_elec_opt <er75> endif
                    <credal> <fifca> <fcoch1>
(2)           'ccase' <fcoch1> <emifca> <fcotxf> <fcobci> <fdexco> <er2>
                    expression <fexcas> <fvecl3> <er37> 'cis' <ffexco>
                    <er32> <fcoch1> choix_elec choix_elec_opt <er52>
                    endcase <credal> <fifca> <fcoch1>
(3)           'cover' <fcoch1> <fbpoex> <er32> 'cidf' <fbpoin> <fcoch1>
                    <er56> ensemble_parcouru <er40> 'crepeat' <er2>
                    liste_instructions_elec <er5> endover <fbpaf> <fcoch1>
(4)           'cidf' <fcoch1> <fstart> suite_idfgau

863     suite_idfgau
(1)           'cobjet' <restau> <fconx1> <fel1> indexation <er31>
                    suite_instruction_elec
(2)           'cboite' <restau> <fel17> indexation <er80> 'cpoint'
                    <fcoch1> <er2> 'cidf' <fcoch1> <fel18>
                    indexation_patte suite_transfert

864     else_elec_opt
(1)           ()
(2)           'celse' <fcoch1> <incnal> <empn0> <er32>
                    liste_instructions_elec

865     choix_elec
(1)           etiquettes <er26> 'cdeuxpoints' <fcoch1> <incnal> <er32>
                    liste_instructions_elec

866     choix_elec_opt
(1)           else_elec_opt
(2)           'cptvirg' <fcoch1> <er32> choix_elec <er52> choix_elec_opt

867     suite_instruction_elec
(1)           'cparg' <fcoch1> <er2> connexions_interface <er19> 'cpard'
                    <fcoch1>
(2)           suite_transfert

868     suite_transfert
(1)           'cconnexion' <fel7> <fcoch1> <er32> 'cidf' <fcoch1> <fel11>
                    <fel2> indexation
(2)           'caffect' <fel7> <fcoch1> <fexc16> <er2> expression <fel20>
```

Transformateur de Grammaire Ll(1) - Universite de Grenoble - Niveau 1:6 Le 30 Octobre 1983
Grammaire du Langage gramgene

```
                                                                    --      regles de IMAG_F

869     bloc_equations
    (1).            'cequation' <fcoch1> <if> <er2> liste_equations <er14>
                        'cptvirg' <fcoch1>

870     liste_equations
    (1)             equation equation_opt

871     equation_opt
    (1)             ()
    (2)             'cvirg' <fcoch1> <er2> liste_equations

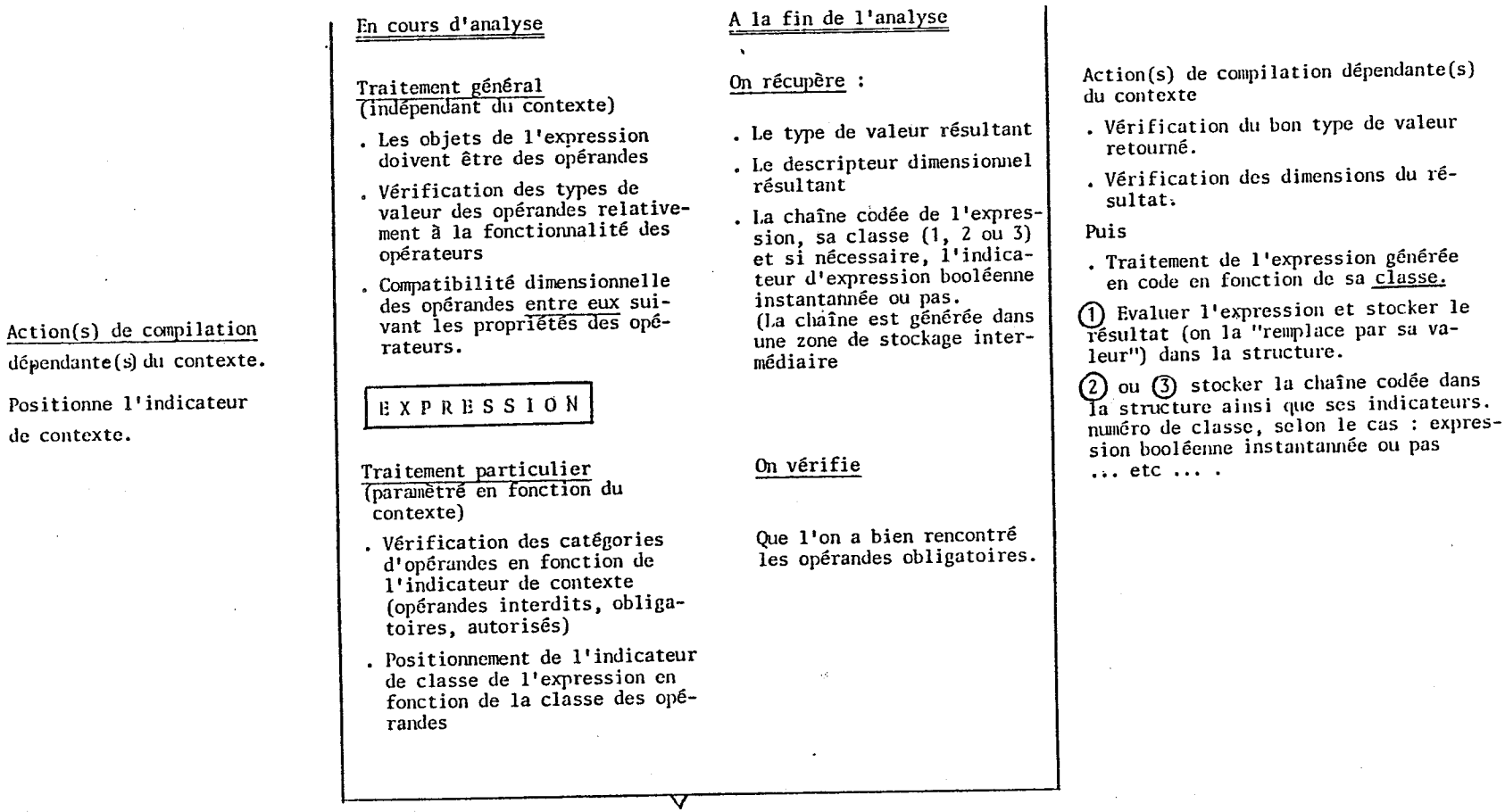
872     equation
    (1)             exp5 <er17> 'cegal' <fcoch1> <er2> exp5
                                                                    --
                                                                    -- FIN DE LA DEFINITION DES REGLES
                                                                    --
```

ANNEXE 2

LES EXPRESSIONS

Occurrence d'un expression dans une règle de grammaire du langage CASCADE

Ce schéma général est simplifié, la double récursivité complexifie notablement le traitement.



Traitement général des expressions

(règle "expression" unique, avec actions de compilation uniques)

Figure A2.1

UTILISATION DES EXPRESSIONS DANS CASCADE (pour les niveaux CASSANDRE, LASCAR, IMAG et LASSO)

Dans partie "déclarations"

Contexte d'utilisation des expressions	Type de valeur du résultat	Dimension du résultat	Utilisation des catégories d'opérandes			
			Obligatoire	Permise (en plus)	Interdite	Classes possibles
Relation utilisée dans les ensembles de valeurs définis par propriété caractéristique	Booléen	Scalaire	Variable muette associée	Constantes explicites Constantes déclarées Attributs	Porteuses Indices de b pour Fonctions	3
Borne dans une déclaration de dimension dans un interface	Entier positif (0 permis)	Scalaire	Rien	Constantes explicites Attributs	Porteuses Fonctions Indices de b pour constantes déclarées Variables muettes	1 2
Borne dans une déclaration de dimension hors interface	Entier positif (0 permis)	Scalaire	Rien	Constantes explicites Constantes déclarées Attributs	Porteuses Fonctions Indices de b pour Variables muettes	1 2
Elément d'une dimension dans une indexation de tableau (dans "partie déclarations")	Entier positif (0 permis)	Scalaire	Rien	Constantes explicites Constantes déclarées Attributs Porteuses Fonctions	Indices de b pour Variables muettes	1 2 3
Expression synonyme dans une déclaration de porteuse	Celui de la porteuse déclarée en partie gauche du signe =	Compatible avec la porteuse définie en partie gauche	Porteuses déclarées sous le même type (triplet complet) que l'objet défini en partie gauche	Rien	Indices de b pour variables muettes Constantes explicites Constantes déclarées Attributs Fonctions	3
Expression synonyme dans une déclaration de constante	Celui de la constante déclarée en partie gauche du signe =	Compatible avec la constante définie en partie gauche	Rien	Constantes explicites Constante déclarées Attributs	Indices de b pour Variables muettes Porteuses Fonctions	1 2
Relations à vérifier dans les assertions	Booléen	Scalaire	Rien	Constantes explicites Constantes déclarées Attributs Porteuses Fonctions	Indices de b pour Variables muettes	2 3
Expression "attributs-effectifs" d'une description dans un use	Celui de l'attribut correspondant dans l'interface de la description utilisée	Scalaire	Rien	Constantes explicites Constantes déclarées Attributs	Indices de b pour Variables muettes Porteuses Fonctions	1 2

Dans "partie relations"

Condition d'une instruction Si ou expression Si	Booléen	Scalaire	Rien	Constantes explicites Constantes déclarées Attributs Porteuses Fonctions Indices de b pour	Variables muettes	1 2 3
Expression d'une instruction Cas ou expression Cas	Entier ou Logique ou Chaîne de caractères	Scalaire	Rien	Constantes explicites Constantes déclarées Attributs Porteuses Fonctions Indices de b pour	Variables muettes	1 2 3
Bornes d'intervalle parcouru dans une boucle "pour" et pas associé	Entier	Scalaire	Rien	Constantes explicites Constantes déclarées Attributs Indices de b pour	Variables muettes Porteuses Fonctions	1 2

Expression en partie droite d'une connexion ou d'un chargement	Compatible avec celui de la porteuse en partie gauche des signes .= ou ⇐	Compatible avec celui de la porteuse en partie gauche des signes .= ou ⇐	Rien	Constantes explicites Constantes déclarées Attributs Indices de b pour Porteuses Fonctions	Variables muettes	1 2 3
Expression dans connexion d'interface, correspondant à une patte "in" seulement	Compatible avec celui de la patte formelle "in" correspondante	Compatible avec la patte formelle "in" correspondante	Rien	Constantes explicites Constantes déclarées Attributs Indices de b pour Porteuses Fonctions	Variables muettes	1 2 3
Expression d'horloge	Impulsion	Scalaire	Rien	Constantes explicites Constantes déclarées Attributs Indices de b pour Porteuses Fonctions	Variables muettes	1 2 3
Expression d'état	Valetat	Scalaire	Rien	Constantes explicites Constantes déclarées Attributs Indices de b pour Porteuses Fonctions	Variables muettes	1 2 3
Elément d'une dimension dans une indexation de tableau	Entier positif (0 interdit)	Scalaire	Rien	Constante explicites Constantes déclarées Attributs Indices de b pour Porteuses Fonctions	Variables muettes	1 2 3
Expression en paramètre d'opérateurs - de conversion (nombre de bits) - de transposition (n° de dimensions) - de retard - de fan	Entier positif (0 interdit)	Scalaire	Rien	Constante explicites Constantes déclarées Attributs Indices de b pour	Variables muettes Porteuses Fonctions	1 2
Rang d'élément dans un selon (LASSO)	Entier positif (0 interdit)	Scalaire	Rien	Constantes explicites Constantes déclarées Attributs Indices de b pour Porteuses Fonctions	Variables muettes	1 2 3
Bornes d'intervalle dans un selon (LASSO)	Entier	Scalaire	Rien	Constantes explicites Constantes déclarées attributs indices de b pour porteuses fonctions	Variables muettes	1 2 3
Membres d'une équation (IMAG)	Réel	Scalaire	Rien	Constantes explicites Constantes déclarées attributs indices de b pour porteuses fonctions	Variables muettes	1 2 3
Expression en partie droite d'une affectation (IMAG)	Réel	Scalaire	Rien	Constantes explicites Constantes déclarées attributs indices de b pour porteuses fonctions	Variables muettes	1 2 3

figure A2.2

ANNEXE 3

RÉALISATION DES NIVEAUX CASSANDRE ET LASCAR DU LANGAGE CASCADE

1. LA MODELISATION CASSANDRE - LASCAR (Mer.73, Bor.76)

1.1. CASSANDRE : Elle se situe au niveau transfert de registres et permet de vérifier les systèmes décrits en terme de fonctions booléennes et de séquencement d'actions élémentaires (par la notion d'automate).

1.2. LASCAR : est un langage général de description et de simulation des calculateurs au niveau architecture et spécification des composants.
C'est une extension de CASSANDRE avec lequel il est entièrement compatible.

1.3. Les versions Synchrones et Asynchrones

- Les premières correspondent à des modèles fonctionnant suivant la notion de cycle de calcul, au sens d'itérations.

- Les secondes sont basées sur des notions temporelles ("retards", opérateurs générateurs d'impulsions) permettant de vérifier que le schéma technologique réalise le schéma logique.

2. LES TYPES DE PORTEUSES ET DE VALEURS

Les types de porteuses primitifs de CASSANDRE sont :

REGISTRE
SIGNAL
LATCH

Pour LASCAR il faut ajouter à cette liste :

COMPTEUR

Les types de valeurs prédéfinis de CASSANDRE sont :

LOGIQUE
BOOL (booléen)
IMPULSION
ETAT

Pour LASCAR, il faut ajouter à cette liste :

ENT (entier)
CAR (caractères)

Détaillons maintenant en particulier les quatre types de valeurs prédéfinis de CASSANDRE.

2.1. IMPULSIONS : Ce type permet de modéliser les tops d'horloges.
On notera 0 et 1 les valeurs "bas" et "haut" (la première correspondant à l'absence d'impulsion, la deuxième à la présence).

Exemple : Déclaration du type HORLOGE et d'une horloge H

```
type HORLOGE = SIGNAL(IMPULSION,0);
                                ou 1
HORLOGE H;
Utilisation d'une horloge
!H! chargements ...
```

2.2. LOGIQUE et BOOLEEN : On a introduit dans CASCADE le type de valeurs LOGIQUE en tant que logique théorique à 36 valeurs. Ce type est prédéfini dans le langage et correspond à l'ensemble des valeurs logiques définies dans la lexicographie par les constantes

`0, `1, ... `9, `A, `B ..., `Z

Les opérations pouvant être effectuées sur ces valeurs sont les opérations logiques et de comparaison.
Elles sont paramétrables en ce sens que c'est l'utilisateur lui-même qui définit les tables de vérités (fonction calcul) nécessaires à l'exécution du calcul à la simulation (un cas par défaut est prévu). Les logiques utilisables par le concepteur peuvent être définies par restriction du type de valeurs LOGIQUE au moyen de propriétés caractéristiques ou par énumération, exactement comme pour les autres types de valeurs. Ainsi, n'importe quel sous-ensemble des 36 valeurs possibles peut être adopté mais il est évident qu'il est préférable pour des raisons de lisibilité de choisir une typographie déjà consacrée dans ce domaine.

Exemple d'une logique à 4 valeurs :

type LOGIC4 = {'0, '1, 'u, 'Z}

où les opérations permises sont celles héritées du type LOGIQUE.

Autres exemples

- type LOGIC4 = {'A, 'B, 'C, 'Z}
- une logique à 3 valeurs : '0, '1, 'Z peut être définie directement par type LOGIC3 = {'0, '1, 'Z} ou par propriété caractéristique, à partir d'une restriction déjà définie :

type LOGIC3 = {X . < LOGIC4/X = '0 | X = '1 | X = 'Z}

ou encore :

type LOGIC3 = {X . <LOGIC4/X ¬= 'u}

etc ... .

Remarquons que ce mécanisme implique, pour une simulation donnée l'unicité de la table de vérité relative à chaque opérateur. En effet, avant tout calcul l'utilisateur devra spécifier pour chaque opérateur la table de vérité correspondante. Il n'est pas obligé de rentrer les 36 × 36 cas possibles, mais seulement ceux qui correspondent à l'union des éléments des logiques qu'il s'est définies.
Les fonctions de calcul associées aux opérateurs portant sur une logique particulière sont obtenues par restriction.
On peut ainsi faire cohabiter dans un même modèle ou description plusieurs logiques disjointes ou non, dérivant l'une de l'autre ou non, avec même nombre de valeurs ou non ... etc ... .
Enfin, du point de vue pratique, il est recommandé de définir les types logiques dans des compléments de langage.

Cas de la logique booléenne (à 2 valeurs)

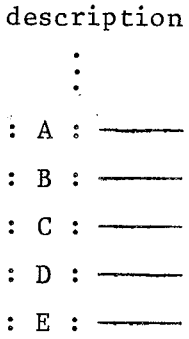
Elle est définie par exemple par : type LOGIC2 = {'0, '1} et ses tables de vérité associées. Pourtant, compte tenu du fait que cette logique est de loin

la plus utilisée, elle apparaît directement dans le système comme type de valeur prédéfini BOOLEEN et coïncide parfaitement avec la logique restreinte évoquée. De plus, ce type de valeurs est celui des conditions, des relations à vérifier etc ... c'est-à-dire des expressions booléennes (à résultat "Vrai" (1) ou "Faux" (0)).

2.3. ETAT : C'est l'ensemble des étiquettes d'états possibles dans les automates CASSANDRE et LASCAR. C'est donc l'ensemble des identificateurs. Voyons maintenant comment est définie cette notion d'automate dans CASCADE.

- Les noms d'états d'un automate constituent une liste d'étiquettes qui correspond à un type énuméré défini implicitement par l'utilisateur à partir de ETAT comme type de base. De manière interne on crée ce type sous le nom $VALnom-de-descr ($VALA dans la réalisation).

Exemple : à



correspond la définition : type $VALMULT = A,B,C,D,E .

Toujours par le même moyen, l'utilisateur peut, en restreignant ce type explicitement, définir des parties quelconques d'un automate.

Exemple : type VPART = {B,E,D}

- Le registre d'état de la description contenant l'automate est aussi défini de manière implicite, de la façon suivante (sous le nom interne $nom-de-desc; c'est en fait $A qui a été utilisé dans

type $REGETIMPL = REGISTRE($VALnom-de-descr, 1er état)
$REGETIMPL $nom-de-descr.

Soit, avec l'exemple précédent :

```
type $REGETIMPL = REGISTRE( $VALMULT, A)
$REGETIMPL $MULT
```

- Le type ETAT permettant de déclarer dans CASSANDRE et LASCAR des registres d'état auxiliaires est défini dans les compléments de langage systèmes correspondants par

```
type REGETAT = REGISTRE( ETAT, ϕ)
```

(avec ϕ = étiquette vide) et est à la disposition des utilisateurs, tout comme REGBϕ, SIGBϕ etc ... .

La philosophie du langage permet d'étendre les possibilités précédemment offertes par CASSANDRE et LASCAR, car l'utilisateur peut se définir des registres d'état auxiliaires où l'ensemble des valeurs n'est plus ETAT mais un sous-ensemble (type énuméré défini par l'utilisateur). Les registres déclarés sous de tels types ne peuvent plus contenir n'importe quelle étiquette, et le contrôle effectué automatiquement tout au long de la simulation, à chaque chargement, peut être d'un réel intérêt pour la modélisation.

3. LES COMPLEMENTS DE LANGAGE SYSTEME

Ils contiennent les types considérés comme primitifs pour ces niveaux; c'est-à-dire pouvant être utilisés dans des descriptions sans avoir à être déclarés.
Ils font partie de la définition de ces niveaux.
On trouve pour :

CASSANDRE Synchrone :

```
langage CASSANDRE_S
corps
type TERN     = {`Ø,`1,`u}   ;
     QUART    = {`Ø,`1,`u,`z};
     REGBØ    = REGISTRE ( BOOL      , `Ø ) ;
     REGTU    = REGISTRE ( TERN      , `u ) ;
     REGETAT  = REGISTRE ( ETAT      ,    ) ;
     SIGBØ    = SIGNAL   ( BOOL      , `Ø ) ;
     SIGTU    = SIGNAL   ( TERN      , `u ) ;
     HORLOGE  = SIGNAL   ( IMPULSION , Ø  ) ;
     BUS      = SIGNAL   ( QUART     , `u ) ;
     LATBØ    = LATCH    ( BOOL      , `Ø ) ;
     LATTU    = LATCH    ( TERN      , `u ) ;
     BREGØ    = REGISTER ( BOOL      , `Ø ) ;
     TREGU    = REGISTER ( TERN      , `u ) ;
     STATEREG = REGISTER ( STATE     ,    ) ;
     BTMØ     = TERMINAL ( BOOL      , `Ø ) ;
     TTMU     = TERMINAL ( TERN      , `u ) ;
     CLOCK    = TERMINAL ( PULSE     , Ø  ) ;
     BLATØ    = LATCH    ( BOOL      , `Ø ) ;
     TLATU    = LATCH    ( TERN      , `u ) ;
finlangage
```

et idem pour CASSANDRE Asynchrone

Ces compléments de langage amènent les remarques suivantes :

TERN est un type de valeurs logiques à trois éléments :

`0, `1 et `u pour "indéterminé"

QUART est un type de valeurs logiques à quatre éléments, les trois précédents plus la valeur `z ou "haute impédance".

LASCAR Synchrone :

```
langage LASCAR_S
corps
type TRI      = {`Ø,`1,`z};
     COMPTØ   = COMPTEUR ( ENT       , Ø  ) ;
     REGBØ    = REGISTRE ( BOOL      , `Ø ) ;
     REGENTØ  = REGISTRE ( ENT       , Ø  ) ;
     REGCAR   = REGISTRE ( CAR       , '' ) ;
     REGETAT  = REGISTRE ( ETAT      ,    ) ;
     SIGBØ    = SIGNAL   ( BOOL      , `Ø ) ;
     SIGENTØ  = SIGNAL   ( ENT       , Ø  ) ;
     SIGCAR   = SIGNAL   ( CAR       , '' ) ;
     HORLOGE  = SIGNAL   ( IMPULSION , Ø  ) ;
     BUS      = SIGNAL   ( TRI       , `z ) ;
     LATBØ    = LATCH    ( BOOL      , `Ø ) ;
     LATENTØ  = LATCH    ( ENT       , Ø  ) ;
     LATCAR   = LATCH    ( CAR       , '' ) ;
```

```
COUNTØ    = COUNTER  ( INT      , Ø   ) ;
BREGØ     = REGISTER ( BOOL     , `Ø  ) ;
IREGØ     = REGISTER ( INT      , Ø   ) ;
SREG      = REGISTER ( STRING   , ''  ) ;
STATEREG  = REGISTER ( STATE    ,     ) ;
BTMØ      = TERMINAL ( BOOL     , `Ø  ) ;
ITMØ      = TERMINAL ( INT      , Ø   ) ;
STM       = TERMINAL ( STRING   , ''  ) ;
CLOCK     = TERMINAL ( PULSE    , Ø   ) ;
BLATØ     = LATCH    ( BOOL     , `Ø  ) ;
ILATØ     = LATCH    ( INT      , Ø   ) ;
SLAT      = LATCH    ( STRING   , ''  ) ;
finlangage
```

et idem pour LASCAR Asynchrone

TRI est un type de valeurs logiques à trois éléments :

`0, `1 et `z

Parmi les types de porteuses, remarquons les types REGCAR et SIGCAR à valeurs "chaîne de caractères" CAR, dont la valeur par défaut est la chaîne vide.

Rappelons enfin que l'utilisateur peut déclarer explicitement dans son modèle tout type de valeur et de porteuse construit à partir des types de porteuses primitifs et des types de valeurs connus dans CASSANDRE et LASCAR, pourvu que les combinaisons imaginées soient légitimes.

4. LES NOTIONS SPECIFIQUES

Elles viennent s'ajouter aux concepts communs aux différents niveaux de langage, pour former les niveaux considérés ici.

Rappelons que la syntaxe commune porte pratiquement en totalité sur tout ce qui est :

- déclarations
- définitions
- assertions
- compléments de langage
- expressions et notion d'indexation

ainsi que certaines formes générales, que l'on trouve dans la partie relations et dont on ne reparlera pas ici, à savoir :

. les formes conditionnelles "Si" et "Cas"
. l'instruction répétitive "pour"
. les connexions d'unités.

4.1. Notions communes à CASSANDRE et LASCAR

4.1.1. Les interfaces : La seule particularité à signaler est que les sens autorisés pour les éléments d'interface sont :

in : pour les entrées
out : pour les sorties
inout : pour les éléments dont la direction peut varier selon les instants, en fonction de certaines conditions.

Exemple : description MICRO (in SIBO CONTROLE [1:8];
out SIBO CRENDUS [1:8];
inout BUS EXT [1:32])

4.1.2. Les connexions de signaux : Ce sont les connexions de signaux classiques de CASSANDRE et LASCAR dont la forme est :

objet .= expression

Exemple : A [0:3] .= B[9:13] & C[1:4];

Le récepteur objet est une porteuse dimensionnée ou non, de type SIGNAL ou LATCH (signal particulier qui à la simulation, s'il n'est pas calculé, conserve sa valeur au lieu de retomber à sa valeur par défaut).

La cohérence des dimensions et des types dans l'expression et le transfert doit bien sûr être respectée.

4.1.3. Les affectations de registres : Les chargements d'éléments de mémorisation peuvent être regroupés en plusieurs blocs, chacun sous la portée

d'une horloge. Un bloc-horloge a la syntaxe suivante :

!expression d'horloges! liste-chargements;

un chargement étant :

objet ⇐ expression

Exemple : !H! B[0;15] <= EM[0:16], C ⇐ `0000;

Le récepteur objet est une porteuse dimensionnée ou non de type REGISTRE. Là aussi la cohérence des dimensions et des types dans l'expression et le transfert doit être respectée.

4.1.4. Autres instructions sous portée d'horloge : Il s'agit des appels de procédures avec paramètres effectifs registres modifiables
!exp.h! nom-proc (liste des paramètres effectifs)
et des ordres de transition d'état associés à un paramètre
!exp.h.! transitera expression-d'état.

Exemples : !H! HISTO (INST,STA[1:12]);
!H! transitera CALCUL;

4.1.5. Les automates : Ils permettent de modéliser commodément la partie contrôle d'un circuit sans préjuger de la façon dont elle est réalisée. Ils se présentent comme une succession d'états sous forme de blocs étiquetés.
Cette notion a déjà été abordée dans cette annexe.

4.1.6. Les opérateurs : Les opérateurs spécifiques à CASSANDRE et LASCAR sont :

- ↑ détection front montant de signal ou latch (dérivation logique), pour la version asynchrone seulement de CASSANDRE et LASCAR

- ↓¬ détection front descendant (inverse du précédent)

- ⊘ conversion binaire → entier positif

- % n! opérateur retard pour CASSANDRE et LASCAR asynchrones seulement.

(ne sont pas cités ici les opérateurs généraux des expressions : opérateurs logiques, arithmétiques, de comparaison, de structuration de tableaux, etc ...).

4.2. Notions spécifiques à LASCAR :

4.2.1. Incrémentation et décrémentation de compteur : Il s'agit des opérateurs "plus 1" (pl) et "moins 1" (ml) dont la fonction est évidente. Ils doivent être utilisés sous portée d'horloge.

Exemple : !H! pl(CCK) , ml(P3);

Nous comptons rajouter un opérateur d'initialisation (init) non implanté actuellement, qui permettra de réinitialiser un compteur à sa valeur par défaut.

4.2.2. Opérateurs de conversion : Il s'agit de :

- intval : pour la conversion binaire → entier relatif en considérant le codage en complément à deux
- convert : pour la conversion entier positif → binaire sur n bits
- binval : pour la conversion entier relatif → binaire sur n bits, avec codage en complément à deux.

ANNEXE 4

CODES INTERNES DES UNITÉS LEXICALES

CODE	SYMBOLISME	TERMINAL
0	Commentaire (inconnu pour grammaire)	cv0
1	Entier sans base explicite	centier
2	Réel simple	creel
3	Réel avec exposant ou f.e.	creeldecexp
4	Réel avec expos. ou f.e.,ss part.déc.	creelexp
5	Entier binaire	cbinaire
6	Entier octal	coctal
7	Entier hexadécimal	chexa
8	Identificateur	cidf
9	Constante littérale	clitteral
10	Constante logique	clogique

SYMBOLES SIMPLES

20	;	cptvirg
21	+	cplus
22	-	cmoins
23	*	cmult
24	/	cslash
25	^	cpuissance
26	|	cou
27	&	cet
28	~	cnon
29	=	cegal
30	<	cinf
31	>	csup
32	!	cptexclam
33	{	caccolades
34	}	caccoladed
35	,	cvirg
36	.	cpoint
37	:	cdeuxpoints
38	%	cpourcent
39	\	cconcat
40	(	cparg
41	)	cpard
42	[	ccrochetgauche
43	]	ccrochetdroit
44	$	cdollar
45	?	cpint
46	_	cmoinsunr
47	^	cmonte

SYMBOLES DOUBLES

60	**	cpuisselec
61	^~	cdescend
62	:=	caffect
63	=>	cchargeretat
64	.=	cconnexion
65	.<	cappartient
66	<=	cchargement
67	=<	cegalinf
68	>=	csupegal
69	~=	cnonegal

104	a	cto
150	ada	cada
138	a_partir_de	capartir
168	arctg	carctg
167	arcsin	carcsin
78	assertion	cassert
143	and_also	cetdeplus
108	after	capres
142	as	ccommesi
119	abs	cabs
96	alors	cthen
108	apres	capres
146	acces	cacces
146	access	cacces
78	assertions	cassert
137	at	cen
88	avant	cavant
78	assert	cassert
88	before	cavant
120	binval	cbinval
125	begin	cbegin
99	case	ccase
135	chaque	cchaque
72	body	cbody
142	comme_si	ccommesi
131	choixp	cchoixp
175	choixr	cchoixr
111	charger	ccharger
99	cas	ccase
164	cos	ccos
72	corps	cbody
154	d	cderiv
180	bidon	cbidon
178	desstand	cdesstand
112	convert	cconvert
103	de	cfrom
87	declare	cdeclare
179	polcas	cpolcas
112	cv	cconvert
139	during	cpendant
130	choix	cchoix
127	delai	cdelay
87	decl	cdeclare
87	declaration	cdeclare
71	description	cdescription
71	descr	cdescription
103	depuis	cfrom
89	endproc	cendprocedure
90	endfunc	cendfunction
87	declarations	cdeclare
73	end	cend
125	debut	cbegin
157	defaut	cdefault
157	default	cdefault
74	e	cin
97	else	celse
74	entree	cin
80	externe	cexternal
101	endcase	cendcase
92	endlanguage	cendlanguage
89	endprocedure	cendprocedure
98	endif	cendif

154	deriv	cderiv
92	endlang	cendlanguage
135	each	cchaque
127	delay	cdelay
80	external	cexternal
137	en	cen
155	equation	cequat
90	endfunction	cendfunction
126	elec_relation	crelations
124	enddescription	cendescr
160	exp	cexp
107	endover	cendover
124	enddescr	cenddescr
76	es	cinout
81	fonc	cfunction
74	entrees	cin
89	finproc	cendprocedure
90	finfonc	cendfunction
80	externes	cexternal
143	et_de_plus	cetdeplus
132	en_et_apres	cenetapres
126	elec_relations	crelations
100	est	cis
155	equat	cequation
86	except	csauf
94	en_et_avant	cenetavant
155	equ	cequat
144	ffixe	cffix
145	frepete	cfrepet
92	finlangage	cendlanguage
89	finprocedure	cendprocedure
115	eqv	ceqv
92	finlang	cendlanguage
98	fsi	cendif
98	finsi	cendif
136	every	ctousles
103	from	cfrom
73	fin	cend
117	fan	cfan
148	fortran	cfortran
81	fonction	cfunction
81	function	cfunction
90	finfonction	cendfunction
124	findescription	cenddescr
107	fpour	cendover
107	finpour	cendover
124	findescr	cenddescr
101	fcas	cendcase
101	fincas	cendcase
144	fix_window	cffix
156	global	cglobal
162	log10	clog10
94	in_and_before	cenetavant
95	if	cif
159	i	ci
118	intval	cintval
74	in	cin
110	mi	cmi
104	jusqua	cto

83	inter	cinter
132	in_and_after	cenetapres
100	is	cis
140	init	cinit
76	inout	cinout
129	index	cindex
111	load	cchargeг
91	langage	clangage
91	language	clangage
70	lanref	creflan
161	log	clog
91	lang	clangage
109	pl	cpl
133	la_fois	clafois
116	mod	cmod
177	list	clist
147	modif	cw
170	min	cmin
171	modulo	cmodulo
77	nd	cnd
153	nand	cnet
176	none	cnone
169	max	cmax
152	nor	cnou
153	net	cnet
141	next	cpuis
152	nou	cnou
102	over	cover
82	proc	cprocedure
75	out	cout
151	prive	cprive
151	private	cprive
82	procedure	cprocedure
174	parametre	cparam
114	oux	cxor
149	pascal	cpascal
174	param	cparam
174	par	cparam
102	pour	cover
174	parameter	cparam
105	pas	cstep
141	puis	cpuis
139	pendant	cpendant
134	que	cque
121	reduc	creduc
126	relation_elec	crelations
126	relations_elec	crelations
123	rang	crang
173	rangx	crangx
70	reflan	creflan
85	return	creturn
126	relation	crelations
85	retour	creturn
106	repeter	crepeat
126	relations	crelations
106	repeat	crepeat
75	sortie	cout
86	sauf	csauf
145	rep_window	cfrepet

95	si	cif
138	starting_from	capartir
163	sin	csin
97	sinon	celse
128	selon	cselon
105	step	cstep
131	selectp	cchoixp
111	transitera	ccharger
75	s	cout
75	sorties	cout
166	sqrt	csqrt
130	select	cchoix
84	type	ctype
165	tan	ctan
96	then	cthen
104	to	cto
127	transp	ctransp
133	the_number	clafois
84	types	ctype
136	tous_les	ctousles
128	test	cselon
134	time_that	cque
93	use	cuse
93	unite	cuse
104	until	cto
79	union	cunion
93	units	cuse
93	unites	cuse
93	unit	cuse
113	validate	cvalider
120	valbin	cbinval
113	valider	cvalider
172	vabs	cvabs
118	valent	cintval
158	v	cv
147	w	cw
114	xor	cxor

ANNEXE 5

EXEMPLE D'ÉLÉMENTS CONTENUS DANS LE DESCRIPTEUR PRINCIPAL D'UNE DESCRIPTION

Mnémonique	Signification
BASE	Début de l'espace de travail de la description englobante par rapport au début de l'espace de travail général (cas où la description courante est définie interne).
PTC	Pointeur de remplissage de l'espace de travail (calculé par rapport à la base de la structure courante).
COLAN	Code niveau de langage.
NOMD	Nom de la description (pointeur sur le nom).
LATT	Liste des attributs formels.
PORD	Accès au nom de la description englobante (cas où la description courante est définie interne).
BASI	Début de la chaîne codée des instructions (fin du tableau espace de travail.
LINTF	Liste des éléments formels d'interface.
LASI	Liste des assertions d'interface (dynamiques).
LOTY	Liste des objets typés (internes et d'interface).
LDIM	Liste des descripteurs dimensionnels.
LTY	Liste des descripteurs de type.
LNTY	Liste des noms de type.
LTYP	Pointeur vers la table des types de porteuse (prédéfinis).
LTYVP	Pointeur vers la table des types de valeurs prédéfinis.

LTYVU Liste des types de valeurs définis par l'utilisateur.

LDR Liste des descriptions référencées.

LDG Liste des descriptions génératrices.

LUNIT Liste des unités (exemplaires de descriptions générés)

LCOPI Liste des connexions permanentes de l'interface.

LASC Liste des assertions de corps (dynamiques).

LIFCAS Liste des relations conditionnelles (chargements sous horloge, instructions "if" et "case").

LCTE Liste des constantes.

LDE Liste des descriptions déclarées externes.

LDI Liste des descriptions définies internes.

LFI Liste des fonctions définies internes.

LPI Liste des procédures définies internes.

CODES Code segment : description, fonction ou procédure.

LBPO Liste des indices de boucles POUR.

LRETA Liste des retards.

LASIA Liste des assertions d'interface, d'attributs (statiques).

PTI Pointeur de remplissage de la chaîne codée.

TCLE Table des clés de l'adressage dispersé des chaînes de caractères.

TYP — Table des types de porteuses (prédéfinis).

TYVP — Table des types de valeurs prédéfinis.

LASCA — Liste des assertions de corps, d'attributs (statiques).

LSYN — Liste des synonymes de porteuses.

NCL — Nombre de compléments de langage CL_i référencés, dans l'ordre reflan du segment.

B I B L I O G R A P H I E

Bac.79 Backhouse R.C.
"Syntax of Programming Languages,
Theory and Practice"
Prentice Hall International, Series in Computer Science
C.A.R. Hoare, Series Editor, London 1979

BaP.83a Battistoni G., Prinetto P.
"Simulation command language"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
SICO-POLI-SS-004, Politecnico di Torino 22 Sep. 1983

BaP.83b Battistoni G., Prinetto P.
"A proposal for the Simulation Supervisor implementation"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
nSYC-POLI-SS-05, Politecnico di Torino 12 Dec. 1983

BDG.84 Borrione D., Decouchant V., Genevey J.
"CASCADE : Génération de code pour les blocs de simulation
d'une description CASCADE"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
dMMS-IMAG-SD-001, Lab. ARTEMIS, Grenoble, 23 Aug. 1984.

BHL.83 Borrione D., Humbert M., Le Faou C.
"Hierarchical mixed-mode simulation in the CASCADE Project"
Proc. of the VLSI Conference, Trondheim - Aug. 1983

BLR.84 Bourdon M., Le Faou C., Routin J.
"Integration des simulateurs logiques concurrents dans
l'environnement CASCADE"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
gFGS-IMAG-SS-001, Lab. ARTEMIS, Grenoble, 27 Aug. 1984

BMB.83 Borrione D., Mermet J., Battistoni G., Prinetto P.
A Time Profile Description Language for system specification
and simulation"
Rapport de Recherche IMAG n°406, Grenoble, Nov. 1983

BMP.84 Battistoni G., Mermet J., Prinetto P.
"The Simulation Environment"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
nSYC-POLI-SS-001, Politecnico di Torino, 01 Jun. 1984

BoG.79 Borrione D., Grabowiecky J.F.
"Informal introduction to LASSO : a Language for Asynchronous Systems Specification and Simulation"
Proc. EURO IFIP 79, London, Septembre 1979

BoG.80 Borrione D., Grabowiecky J.F.
"Réalisation d'un prototype du compilateur et du simulateur du langage LASSO :
Tome 1 : Le Compilateur Prototype
Tome 2 : Le Simulateur Prototype
Tome 3 : Manuel d'Utilisation
Rapport final du contrat SESORI n°78067, Août 1980

BoM.84 Borrione D., Mermet J.
"Gate level language in CASCADE : POLO
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
bHDL-IMAG-PR-004, Lab. ARTEMIS, Grenoble, 30 Oct. 1984

Bor.76 Borrione D.
"LASCAR : un langage pour la simulation et l'évaluation des architectures d'ordinateurs"
Thèse de Troisième Cycle, I.N.P.G., Grenoble, Avril 1976

Bor.81 Borrione D.
"Langages de Description de Systèmes Logiques : Proposition pour une méthode formelle de définition"
Thèse d'Etat, I.N.P.G., Grenoble, Juillet 1981.

Bor.84 Borrione D.
"CASCADE : Manuel de Reference"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
Lab. ARTEMIS, Grenoble, Aug. 1984

Bor.85a Borrione D.
"Definition of system level language in CASCADE : LASSO"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
bHDL-IMAG-PR-006, Lab. ARTEMIS, Grenoble, Jan. 1985

Bor.85b Borrione D.
"The CASCADE multi-level Hardware Description Language"
Soumis à VLSI'85 et CHDL'85, Grenoble, Jan. 1985

Bou.84 Bourcier E.
"Conception et réalisation du Simulateur du langage de description de circuits intégrés IRENE-C"
Mémoire CNAM, CUEFA, Grenoble, Octobre 1984

Bre.75a Bressy Y.
"Version Asynchrone du Système CASSANDRE"
Séminaire de Programmation, IMAG, Novembre 1975

Bre.75b Bressy Y.
"Manuel d'utilisation CASSANDRE :
Tome 1 : Guide sémantique d'utilisation du langage CASSANDRE
Tome 2 : Le langage CASSANDRE : description et traitement
Tome 3 : Simulation CASSANDRE : Algorithmes et commandes"
IMAG, Grenoble 1975

Bre.83a Bressy Y.
"Principes de traitement du langage CASCADE dans la première phase de compilation"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
bCOT-IMAG-PR-001, Lab. ARTEMIS, Grenoble, Jul. 1983

Bre.83b Bressy Y.
"Superviseur Elementaire de Simulation : Rapport de spécification"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
dMMS-IMAG-SS-001, Lab. ARTEMIS, Grenoble, Sep. 1983

Bre.83c Bressy Y.
"Définition et implémentation des différentes notions et des mécanismes du langage CASCADE"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
bHDL-IMAG-SR-001, Lab. ARTEMIS, Grenoble, Nov. 1983

Bre.84a Bressy Y.
"Réalisation des niveaux CASSANDRE et LASCAR du langage CASCADE"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
bHDL-IMAG-SR-003, Lab. ARTEMIS, Grenoble, Jan. 1984

Bre.84b Bressy Y.
"Réalisation du niveau LASSO du langage CASCADE"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
bHDL-IMAG-SR-002, Lab. ARTEMIS, Grenoble, Jan. 1984

Bre.84c Bressy Y.
"Validation of the CASCADE prototype by the Benchmarks"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
Lab. ARTEMIS, Grenoble, Aug. 1984

Bre.85 Bressy Y.
"Sequenceur de Simulation CASCADE"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
cRSI-IMAG-SD-001, Lab. ARTEMIS, Grenoble, Jan. 1985

BrL.85 Bressy Y., Le Faou C.
"Validation of the CASCADE prototype by the Benchmarks"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
pSYI-IMAG-ST-003, Lab. ARTEMIS, Grenoble, Jan. 1985

Cau.84 Caubet B.
"Le Catalogueur CASCADE : mise en oeuvre"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
bCOT-IMAG-SD-001, Lab. ARTEMIS, Grenoble, Jan. 1984

CGV.80 Cunin P.Y., Griffiths M., Voiron J.
"Comprendre la Compilation"
Springer-Verlag, Berlin, Heidelberg, New-York, 1980

Dec.84 Decouchant V.
"Mécanismes d'ordonnancement dans CASCADE"
Rapport de DEA Informatique, I.N.P.G., Sept. 1984

DeG.80 Delaunay M., Grabowiecky J.F.
"Note technique d'utilisation du Transformateur de Grammaire Lℓ(1)"
Rapport de Recherche IMAG n°234, Grenoble, Sept. 1980

Del.84 Delacroix M.
"QUALIMETRE-C, analyseur de qualité des programmes : manuel d'utilisation"
Document I.G.L., Décembre 1984

DLU.84 David B., Lafont S., Uvietta P.
"CASCADE User Interface Managment System"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG nSYC-IMAG-PR-003, Lab. ARTEMIS, Grenoble, Jul. 1984

Dur.84 Durant Y.
"Le Compilateur Logique du Système CASCADE : Etude et Spécifications"
Rapport de DEA Informatique, I.N.P.G., Grenoble, Sept. 1984

Gen.83 Genevey J.
"Construction de la SDP"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG SIMU-IMAG-SD-003, Lab. ARTEMIS, Grenoble, Jul. 1983

Gen.84 Genevey J.
"Restructuring tools"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG dMMS-IMAG-SS-001, Lab. ARTEMIS, Grenoble, Jan. 1984

Gra.81 Grabowiecky J.F.
"LASSO : un langage pour la description et la simulation de Systèmes Informatiques - Mise en oeuvre -"
Mémoire CNAM, CUEFA, Grenoble, Mars 1981

Hum.84 Humbert M.
"Projet CASCADE : une approche de la simulation hiérarchisée multi-modes"
Thèse de Docteur-Ingénieur en Informatique, I.N.P.G., Grenoble, Octobre 1984

Kre.84 Kreling B.
"Static Semantics of CONLAN : Rules and Algorithms"
Institut fuer Datentechnik, Darmstadt, March 1984

Lef.74 Le Faou C.
"Un Programme Général de Simulation de Circuits Electroniques : IMAG 3"
Electronique et Micro-Electronique Industrielle, pp 39-43, Octobre 1974

Lef.82 Le Faou C.
"CASCADE : Système de CAO Intégré pour Ensembles Logiques et Electroniques, introduction du niveau Electrique"
Rapport de Recherche IMAG n°308, Grenoble, Juillet 1982

Lef.84a Le Faou C.
"Simulation Data Structure : SDS"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG dMMS-IMAG-SS-003, Lab. ARTEMIS, Grenoble, Jan. 1984

Lef.84b Le Faou C.
"Specification of the SDP (Data Structure of the Project)"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG dMMS-IMAG-SS-004, Lab. ARTEMIS, Grenoble, Jan. 1984

Lef.84c Le Faou C.
"Catalogue Data Structure"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
bCOT-IMAG-SS-001, Lab. ARTEMIS, Grenoble, Jan. 1984

Lef.85 Le Faou C.
"Hierarchical Multilevel Mixed-Mode Simulation in CASCADE"
Soumis à VLSI'85 et CHDL'85, Grenoble, Janvier 1985

Mar.84 Marty J.C.
"Un Editeur Graphique pour le Système CASCADE : EDICAS"
Thèse de 3ème Cycle, I.N.P.G., Grenoble, Octobre 1984

Mer.73 Mermet J.
"Etude Méthodologique de la Conception Assistée par Ordinateur des systèmes logiques : CASSANDRE"
Thèse d'Etat USMG, Grenoble, Avril 1973

Mer.83 Mermet J.
"Circuits and Systems Computers Aided Design & Engineering : CASCADE
CAPE Conference, Avril 1983

Mer.84 Mermet J.
"Transistor level language in CASCADE : CASTOR"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
bHDL-IMAG-PR-005, Lab. ARTEMIS, Grenoble, Oct. 1984

Mer.85 Mermet J.
"Several steps towards a Circuits Integrated CAD System : CASCADE
Soumis à VLSI'85 et CHDL'85, Grenoble, Jan. 1985

Rou.84a Routin J.
"SOS, Structure Orientée Simulation : structure d'accés et d'initialisation de la zone valeur pour Simulateur Logique"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
Lab. ARTEMIS, Grenoble, May 1984

Rou84b Routin J.
"Construction de la SOS"
Rapport CASCADE-CERES-Project, Contrat CEE MR-01-IMG
dMMS-IMAG-SD-003, Lab. ARTEMIS, Grenoble, May 1984

PBB.83 Piloty R., Barbacci M., Borrione D., Dietmeyer D., Hill F. and Skelly P.
"CONLAN Report"
Lecture notes in Computer Science 151,
Springer-Verlag, Berlin, 1983

Vap.82 Vapone F.
"Vers une norme, 101 conseils pour la programmation en FORTRAN"
EDF-Atelier Logiciel n°35, Clamart, Mars 1982

Vax.83 VAX-11 FORTRAN
"Language Reference Manual"
Order N° AA-D034C-TE, 1983.