Spécification et simulation fonctionnelles de circuits complexes : le système CADOC

Michel Crastes de Paulet

Laboratoire Circuits et Systèmes
Institut National Polytechnique de Grenoble

Michel CRASTES DE PAULET

SPECIFICATION ET SIMULATION FONCTIONNELLES DE CIRCUITS COMPLEXES : LE SYSTEME CADOC

Thèse de Docteur Ingénieur en Microélectronique

Date de Soutenance : le 28 Novembre 1985

M. CHEIN : Président
Y. FRANCILLON
J. Cl. LAGARDE
G. MICHEL : Examinateurs
G. SAUCIER

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

ANCEAU François	E.N.S.I.M.A.G.
BARRAUD Alain	E.N.S.I.E.G.
BAUDELET Bernard	E.N.S.I.E.G.
BESSON Jean	E.N.S.E.E.G.
BLIMAN Samuel	E.N.S.E.R.G.
BLOCH Daniel	E.N.S.I.E.G.
BOIS Philippe	E.N.S.H.G.
BONNETAIN Lucien	E.N.S.E.E.G.
BONNIER Etienne	E.N.S.E.E.G.
BOUVARD Maurice	E.N.S.H.G.
BRISSONNEAU Pierre	E.N.S.I.E.G.
BUYLE BODIN Maurice	E.N.S.E.R.G.
CAVAIGNAC Jean-François	E.N.S.I.E.G.
CHARTIER Germain	E.N.S.I.E.G.
CHENEVIER Pierre	E.N.S.E.R.G.
CHERADAME Hervé	U.E.R.M.C.P.P.
CHERUY Arlette	E.N.S.I.E.G.
CHIAVERINA Jean	U.E.R.M.C.P.P.
COHEN Joseph	E.N.S.E.R.G.
COUMES André	E.N.S.E.R.G.
DURAND Francis	E.N.S.E.E.G.
DURAND Jean-Louis	E.N.S.I.E.G.
FELICI Noël	E.N.S.I.E.G.
FOULARD Claude	E.N.S.I.E.G.
GENTIL Pierre	E.N.S.E.R.G.
GUERIN Bernard	E.N.S.E.R.G.
GUYOT Pierre	E.N.S.E.E.G.
IVANES Marcel	E.N.S.I.E.G.
JAUSSAUD Pierre	E.N.S.I.E.G.
JOUBERT Jean-Claude	E.N.S.I.E.G.
JOURDAIN Geneviève	E.N.S.I.E.G.
LACOUME Jean-Louis	E.N.S.I.E.G.
LATOMBE Jean-Claude	E.N.S.I.M.A.G.

.../...

LESSIEUR Marcel	E.N.S.H.G.
LESPINARD Georges	E.N.S.H.G.
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G.
MAZARE Guy	E.N.S.I.M.A.G.
MOREAU René	E.N.S.H.G.
MORET Roger	E.N.S.I.E.G.
MOSSIERE Jacques	E.N.S.I.M.A.G.
PARIAUD Jean-Charles	E.N.S.E.E.G.
PAUTHENET René	E.N.S.I.E.G.
PERRET René	E.N.S.I.E.G.
PERRET Robert	E.N.S.I.E.G.
PIAU Jean-Michel	E.N.S.H.G.
POLOUJADOFF Michel	E.N.S.I.E.G.
POUPOT Christian	E.N.S.E.R.G.
RAMEAU Jean-Jacques	E.N.S.E.E.G.
RENAUD Maurice	U.E.R.M.C.P.P.
ROBERT André	U.E.R.M.C.P.P.
ROBERT François	E.N.S.I.M.A.G.
SABONNADIERE Jean-Claude	E.N.S.I.E.G.
SAUCIER Gabrielle	E.N.S.I.M.A.G.
SCHLENKER Claire	E.N.S.I.E.G.
SCHLENKER Michel	E.N.S.I.E.G.
SERMET Pierre	E.N.S.E.R.G.
SILVY Jacques	U.E.R.M.C.P.P.
SOHM Jean-Claude	E.N.S.E.E.G.
SOUQUET Jean-Louis	E.N.S.E.E.G.
VEILLON Gérard	E.N.S.I.M.A.G.
ZADWORNY François	E.N.S.E.R.G.

PROFESSEURS ASSOCIES

BASTIN Georges	E.N.S.H.G.
BERRIL John	E.N.S.H.G.
CARREAU Pierre	E.N.S.H.G.
GANDINI Alessandro	U.E.R.M.C.P.P.
HAYASHI Hirashi	E.N.S.I.E.G.

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

PROFESSEURS E.N.S. Mines de Saint-Etienne

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

ALLIBERT Michel	Maître de Recherche
ANSARA Ibrahim	Maître de Recherche
ARMAND Michel	Maître de Recherche
BINDER Gilbert	
CARRE René	Maître de Recherche
DAVID René	Maître de Recherche
DEPORTES Jacques	
DRIOLE Jean	Maître de Recherche
GIGNOUX Damien	
GIVORD Dominique	
GUELIN Pierre	
HOPFINGER Emil	Maître de Recherche
JOUD Jean-Charles	Maître de Recherche
KAMARINOS Georges	Maître de Recherche
KLEITZ Michel	Maître de Recherche
LANDAU Ioan-Dore	Maître de Recherche
LASJAUNIAS J.C.	
MERMET Jean	Maître de Recherche
MUNIER Jacques	Maître de Recherche
PIAU Monique	
PORTESEIL Jean-Louis	
THOLENCE Jean-Louis	
VERDILLON André	

CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)

LESBATS Pierre	Directeur de Recherche
BISCONDI Michel	Maître de Recherche
KOBYLANSKI André	Maître de Recherche
LE COZE Jean	Maître de Recherche
LALAUZE René	Maître de Recherche
LANCELOT Francis	Maître de Recherche
THEVENOT François	Maître de Recherche
TRAN MINH Canh	Maître de Recherche

PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)

ALLIBERT Colette	E.N.S.E.E.G.
BERNARD Claude	E.N.S.E.E.G.
BONNET Rolland	E.N.S.E.E.G.
CAILLET Marcel	E.N.S.E.E.G.
CHATILLON Catherine	E.N.S.E.E.G.
CHATILLON Christian	E.N.S.E.E.G.
COULON Michel	E.N.S.E.E.G.
DIARD Jean-Paul	E.N.S.E.E.G.
EUSTAPOPOULOS Nicolas	E.N.S.E.E.G.
FOSTER Panayotis	E.N.S.E.E.G.

.../...

GALERIE Alain	E.N.S.E.E.G.
HAMMOU Abdelkader	E.N.S.E.E.G.
MALMEJAC Yves	E.N.S.E.E.G. (CENG)
MARTIN GARIN Régina	E.N.S.E.E.G.
NGUYEN TRUONG Bernadette	E.N.S.E.E.G.
RAVAINE Denis	E.N.S.E.E.G.
SAINFORT	E.N.S.E.E.G. (CENG)
SARRAZIN Pierre	E.N.S.E.E.G.
SIMON Jean-Paul	E.N.S.E.E.G.
TOUZAIN Philippe	E.N.S.E.E.G.
URBAIN Georges	E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON)
GUILHOT Bernard	E.N.S. Mines Saint Etienne
THOMAS Gérard	E.N.S. Mines Saint Etienne
DRIVER Julien	E.N.S. Mines Saint Etienne
BARIBAUD Michel	E.N.S.E.R.G.
BOREL Joseph	E.N.S.E.R.G.
CHOVET Alain	E.N.S.E.R.G.
CHEHIKIAN Alain	E.N.S.E.R.G.
DOLMAZON Jean-Marc	E.N.S.E.R.G.
HERAULT Jeanny	E.N.S.E.R.G.
MONLLOR Christian	E.N.S.E.R.G.
BORNARD Guy	E.N.S.I.E.G.
DESCHIZEAU Pierre	E.N.S.I.E.G.
GLANGEAUD François	E.N.S.I.E.G.
KOFMAN Walter	E.N.S.I.E.G.
LEJEUNE Gérard	E.N.S.I.E.G.
MAZUER Jean	E.N.S.I.E.G.
PERARD Jacques	E.N.S.I.E.G.
REINISCH Raymond	E.N.S.I.E.G.
ALEMANY Antoine	E.N.S.H.G.
BOIS Daniel	E.N.S.H.G.
DARVE Félix	E.N.S.H.G.
MICHEL Jean-Marie	E.N.S.H.G.
OBLED Charles	E.N.S.H.G.
ROWE Alain	E.N.S.H.G.
VAUCLIN Michel	E.N.S.H.G.
WACK Bernard	E.N.S.H.G.
BERT Didier	E.N.S.I.M.A.G.
CALMET Jacques	E.N.S.I.M.A.G.
COURTIN Jacques	E.N.S.I.M.A.G.
COURTOIS Bernard	E.N.S.I.M.A.G.
DELLA DORA Jean	E.N.S.I.M.A.G.
FONLUPT Jean	E.N.S.I.M.A.G.
SIFAKIS Joseph	E.N.S.I.M.A.G.
CHARUEL Robert	U.E.R.M.C.P.P.
CADET Jean	C.E.N.G.
COEURE Philippe	C.E.N.G. (LETI)

.../...

DELHAYE Jean-Marc	C.E.N.G. (STT)
DUPUY Michel	C.E.N.G. (LETI)
JOUVE Hubert	C.E.N.G. (LETI)
NICOLAU Yvan	C.E.N.G. (LETI)
NIFENECKER Hervé	C.E.N.G.
PERROUD Paul	C.E.N.G.
PEUZIN Jean-Claude	C.E.N.G. (LETI)
TAIEB Maurice	C.E.N.G.
VINCENDON Marc	C.E.N.G.

LABORATOIRES EXTERIEURS

DEMOULIN Eric	C.N.E.T.
DEVINE	C.N.E.T. (R.A.B.)
GERBER Roland	C.N.E.T.
MERCKEL Gérard	C.N.E.T.
PAULEAU Yves	C.N.E.T.
GAUBERT C.	I.N.S.A. Lyon

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

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

Professeurs de 1ère Catégorie

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

Professeurs de 2ème catégorie

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

Directeur de recherche

LESBATS	Pierre	Métallurgie

Maîtres de recherche

BISCONDI	Michel	Métallurgie
DAVOINE	Philippe	Géologie
FOURDEUX	Angeline	Métallurgie
KOBYLANSKI	André	Métallurgie
LALAUZE	René	Chimie
LANCELOT	Francis	Chimie
LE COZE	Jean	Métallurgie
THEVENOT	François	Chimie
TRAN MINH	Canh	Chimie

Personnalités habilitées à diriger des travaux de recherche

DRIVER	Julian	Métallurgie
GUILHOT	Bernard	Chimie
THOMAS	Gérard	Chimie

Professeur à l'UER de Sciences de Saint-Etienne

VERGNAUD	Jean-Maurice	Chimie des Matériaux & chimie industrielle

****

A certains

"Je suis un clown qui recueille des instants"

H. BOLL - La grimace.

Remerciements

Je tiens à exprimer toute ma reconnaissance à Madame Gabrièle SAUCIER, Professeur à l'ENSIMAG, pour m'avoir accueilli dans son laboratoire et permis de préparer dans d'excellentes conditions le travail exposé ici.

Je tiens à remercier

Monsieur le Professeur M. CHEIN de me faire l'honneur de présider le jury de cette thèse,

Monsieur Y. FRANCILLON, responsable CAO électronique, SA MATRA et Monsieur J. Cl. LAGARDE, Directeur Unité Circuits Intégrés, CIT ALCATEL, d'avoir accepté de faire partie de ce jury,

Monsieur G. MICHEL, Responsable Département Architectures des Micro-Systèmes, CNET, qui m'a fait profiter de ses remarques tout au long de la réalisation de ce projet et a accepté d'être rapporteur de cette thèse ainsi que membre de ce jury.

Ce travail est le résultat d'une étroite et agréable collaboration avec J. RARIVOMANANA, S. HANRIAT et F. TIAR, qu'ils trouvent ici, ainsi que mes collègues du Laboratoire Circuits et Systèmes, l'expression de ma profonde sympathie.

Je tiens enfin à remercier S. HUMBLOT d'avoir assuré la frappe de cette thèse ainsi que le service de reprographie de l'IMAG d'en avoir effectué le tirage.

TABLE DES MATIERES

INTRODUCTION

L'origine du travail présenté ici est la définition d'un langage adapté à la spécification et à la description du matériel informatique (ou CHDL pour Computer Hardware Description Language). Parmi les nombreux langages actuellement existants ou proposés sur ce sujet, on peut distinguer trois grandes classes : les langages orientés preuve, ceux orientés synthèse et ceux orientés simulation ; nous montrerons au cours de cette étude les incompatibilités entre ces trois orientations qui supposent l'utilisation de langages totalement différents dans leurs principes, indiquons simplement ici que l'objectif premier de ce travail est la définition d'un langage de simulation et éventuellement de synthèse et que nous intéresserons surtout aux circuits digitaux.

Dans une première partie, aprés avoir défini un certain nombre de termes couramment employés, nous nous intéresserons aux deux approches possibles de définition d'un tel langage : la première approche prend appui sur des langages développés pour d'autres buts (langages de processus, langages temps réel,...) et tente, par des mécanismes d'extension du langage initial, de prendre en compte la réalité des circuits et des technologies. La deuxième approche possible revient à considérer que la description du matériel est un problème spécifique qu'il n'est pas possible de traiter correctement par simple extension d'outils non spécifiques et donc qu'il est nécessaire de définir un langage propre à ce domaine. Dans une première partie, donc, des exemples pour les deux approches seront exposés, ces exemples seront accompagnés d'un certain nombre de remarques et de critiques sur la capacité des langages considérés à donner des modélisations satisfaisantes en ce qui concerne le domaine de la circuiterie en général et de la spécification et description en vue de simulation en particulier.

Dans une deuxième partie, nous exposerons le langage spécifique à la description du matériel que nous proposons : le langage CADOC.LD qui satisfait les contraintes propres à la description du matériel et prend en compte les remarques formulées lors de la première partie. Ce langage est le centre d'un système complet de CAO qui sera présenté ensuite.
Deux autres systèmes, l'un d'évaluation de la testabilité de circuits et de cartes, l'autre de génération de programmes de test pour microprocesseurs seront brièvement introduits ainsi que l'intérêt de leur interfaçage avec le système CADOC.

Nous reviendrons dans une troisième et dernière partie sur le problème de la modélisation de dispositifs propres à la technologie actuellement prédominante dans le domaine de la circuiterie à très haute densité d'intégration : la technologie MOS ; en effet, s'il est assez simple de modéliser des circuits de façon "abstraite" (nous reviendrons sur ce terme au cours du texte), certains dispositifs que l'on peut qualifier de bas-niveau sont difficiles à modéliser car ils entrainent la prise en compte de certains points délicats (bidirectionnalité des échanges, variables partagées,...) propres au domaine de la description et de la spécification de circuits.
Nous ferons un survol des différents simulateurs classiques existants, depuis les simulateurs électriques continus -les plus proches de la réalité (au sens connectique) des circuits- jusqu'aux simulateurs hybrides. Pour chacun de ces modes de simulation, nous montrerons la manière dont sont traités les problèmes précédents et nous conclurons en présentant la méthode proposée dans le langage CADOC.LD ainsi que les extensions envisagées en vue d'augmenter la précision temporelle des descriptions. Cette amélioration de la finesse des descriptions peut se faire par prise en compte de données supplémentaires, par exemple les valeurs des capacités à piloter par une sortie d'un circuit en fonction de son environnement, et permet au langage CADOC.LD de couvrir une gamme significative de niveaux de description.

Le système CADOC est actuellement opérationnel ; il a été développé dans le cadre d'un contrat INPG-DAII (n° 82-35-064) puis du projet européen CVT (CAD VLSI for Telecommunications) et a été mis à la disposition des autres partenaires de ce projet.

Le travail a été réalisé en collaboration avec J. RARIVOMANANA qui présente plus particulièrement dans sa thèse ([RAR 85]) l'utilisation du langage CADOC.LD pour la génération de test fonctionnel.

PREMIERE PARTIE

LANGAGES DE DESCRIPTION ET DE SPECIFICATION DU MATERIEL

I - DEFINITIONS

II- LANGAGES NON SPECIFIQUES A LA DESCRIPTION DU MATERIEL

II.1 - LANGAGES IMPERATIFS

II.1.1 - (ADLIB + SDL) → HELIX
II.1.2 - ADLIB
II.1.3 - SDL
II.1.4 - Descriptions de circuits en ADLIB/SDL,HELIX
II.1.5 - Manipulation du temps
II.1.6 - Remarques sur ADLIB/SDL → HELIX
II.1.7 - Autres approches

II.2 - LANGAGES PARALLELES

II.2.1 - Modèle CSP
II.2.2 - OCCAM
II.2.3 - ADA

II.3 - LANGAGES FONCTIONNELS ET LANGAGES DECLARATIFS

II.3.1 - Introduction
II.3.2 - LUSTRE
II.3.3 - LTS
II.3.4 - Remarques

II.4 - LANGAGES DE L'INTELLIGENCE ARTIFICIELLE

II.4.1 - LISP
II.4.2 - PROLOG

III - LANGAGES SPECIFIQUES A LA DESCRIPTION DU MATERIEL

I - DEFINITIONS

Dans les chapitres suivants, nous allons utiliser un certain nombre de termes que nous allons définir rapidement ; ces termes sont utilisés couramment dans les langages de description de matériel (ou CHDLs).
Ces définitions sont orientées "langage", la signification de certains termes pouvant varier lorsque l'on s'intéresse à d'autres domaines d'application (simulation, test, synthèse, ...) ; on peut même noter que ces termes reçoivent parfois des interprétations différentes dans le domaine des langages, selon les auteurs.

I.1 - Domaines de description : fonctionnel, structurel et physique

Etant donné un circuit, sa description peut schématiquement se faire selon trois axes ou domaines :

- le domaine structurel dans lequel le circuit est représenté comme assemblage d'éléments matériels primitifs,
- le domaine fonctionnel dans lequel la fonction réalisée par le circuit (son comportement) est représentée de manière abstraite,
- le domaine topologique dans lequel le circuit est représenté comme un assemblage de figures sans comportement associé.

Pour chacun de ces domaines, on peut distinguer un certain nombre de niveaux. Par ordre d'abstraction croissant on peut définir les niveaux suivants (pour chacun d'eux on indiquera les objets manipulés dans les domaines structurel et fonctionnel) :

- niveau électrique : le comportement du circuit est une fonction continue du temps et est défini au travers d'un ensemble d'équations différentielles ; sa structure est définie par assemblage d'éléments du type transistor, résistance,....

- niveau logique : le comportement du circuit est une fonction discrète du temps et s'exprime sous forme d'un ensemble d'équations booléennes. Dans le domaine structurel, le circuit est décrit comme assemblage de portes logiques (NOR, NAND,...).

- niveau transfert de registre : le comportement du circuit s'exprime à partir de manipulations arithmétiques sur un ensemble de variables qui sont associées aux points de mémorisation du circuit sous sa forme structurelle ; à ce niveau interviennnent les notions d'état et de séquencement d'états. Une description structurelle niveau transfert de registre est définie à partir d'éléments du type registre, multiplexeur, unité arithmétique et logique,...

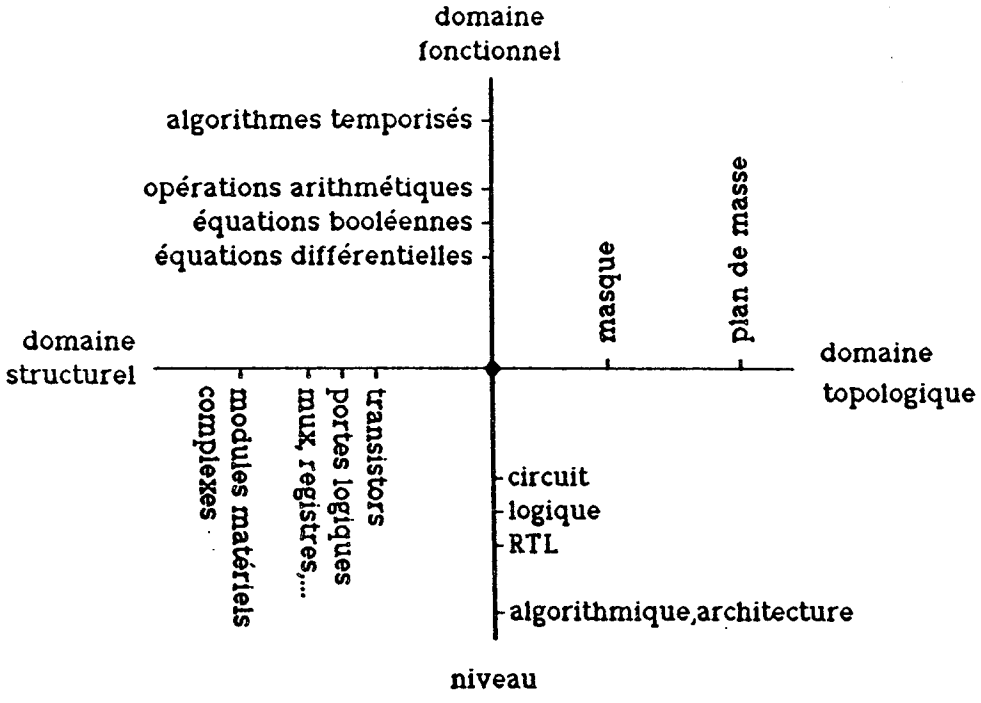
- niveau algorithmique, processus : le comportement du circuit est décrit par un algorithme éventuellement temporisé ; il n'existe pas de correspondance immédiate entre ce niveau du domaine fonctionnel et un niveau du domaine structurel. A ce niveau, une description structurelle du circuit se ferait en termes d'éléments complexes du type mémoire, processeur,....

Remarques

- En ce qui concerne le domaine topologique, la notion de niveau est beaucoup plus floue : indiquons simplement que l'on peut distinguer les niveaux masque (le plus prés de la réalité du circuit) et plan de masse (plus abstrait).

- Le passage automatique d'un domaine de représentation à un autre peut se faire par l'intermédiaire d'outils soit de synthèse, soit d'extraction.

Ces trois domaines et leurs niveaux respectifs peuvent se représenter ([WAL 85]) par :



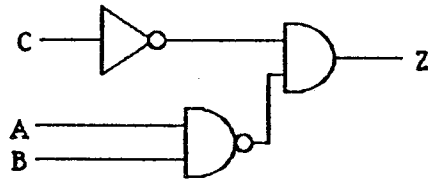
Remarque

La frontière entre niveaux consécutifs n'est pas parfaitement définie en particulier lorsque l'on arrive dans les niveaux d'abstraction élevée.

A partir de maintenant, nous allons oublier le niveau topologique pour ne plus nous intéresser qu'aux domaines fonctionnel et structurel.
Le premier point à noter est que les niveaux de ces deux domaines sont en correspondance biunivoque : le passage d'un domaine à l'autre étant assimilable à un simple changement de notation.

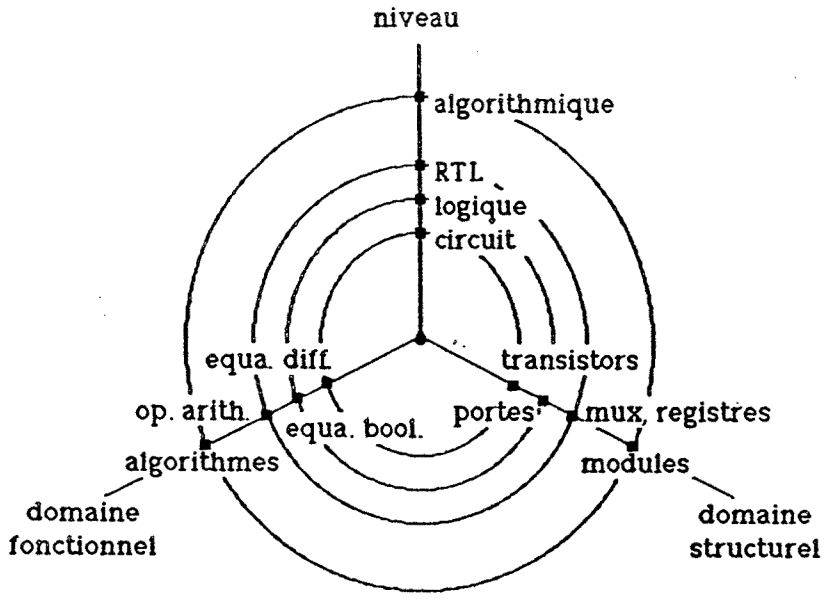
Exemple

Une représentation fonctionnelle du circuit dont la représentation structurelle au niveau logique est



sera **Z = non C et non (A et B)**

Si l'on considère la représentation des domaines fonctionnel et structurel et de leurs niveaux associés donnée précedemment, la correspondance entre niveaux peut se représenter par



On peut alors remarquer que la notion de synthèse d'un circuit à partir d'une description de niveau abstrait peut se ramener à un parcours depuis la périphérie de la figure précédente jusqu'à son centre. Le processus de synthèse pouvant se dérouler en parallèle sur les deux domaines ou procéder par changements successifs sur les domaines. Les transformations sur un même niveau sont de simples changements de notations et ne correspondant pas à un enrichissement de l'information disponible.

I.2 - Langages de description fonctionnelle et structurelle

Un circuit peut être défini à tous les niveaux comme un assemblage de circuits déjà existants (construits ou primitifs). Nous entendrons par langage de description fonctionnelle un langage ne possédant pas d'objets primitifs prédéfinis (au sens description non accessible ni modifiable), mais offrant en revanche aux utilisateurs des mécanismes de définition d'objets primitifs. Cette définition nous permettant de classer les langages RTL comme non fonctionnels ; de plus, d'après cette définition, il n'existe pas de niveau spécifique associé à un langage fonctionnel (un langage fonctionnel est implicitement lié à une bibliothèque de descriptions et exclu tout ensemble d'objets prédéfinis).

Une autre façon de voir les choses est de dire qu'un langage fonctionnel n'utilise pas de primitives en correspondance directe avec des primitives appartenant au domaine structurel.

I.3 - Hiérarchie et multiniveau

Par hiérarchie, nous entendrons description d'un circuit comme un assemblage de sous-circuit, eux-même décrits hiérarchiquement ou non.
Si les circuits feuilles de la hiérarchie sont décrits à des niveaux d'abstraction différents (logique, électrique par exemple), nous parlerons de composition hiérarchisée multiniveau (ou composition multiniveau).

De part la définition d'un langage de description fonctionnelle, toute description hiérarchique utilisant un tel langage est implicitement multiniveau.

I.4 - Procédural et non procédural

Ces deux termes font référence à l'ordonnancement en vue d'exécution des instructions apparaissant dans la représentation fonctionnelle d'un circuit ; si l'ordre d'exécution est celui dans lequel sont données les instructions, on parlera de langage procédural (notion de ";" et de "goto"), sinon on parlera de langage non-procédural (notion d'état défini comme étiquette conditionnelle).

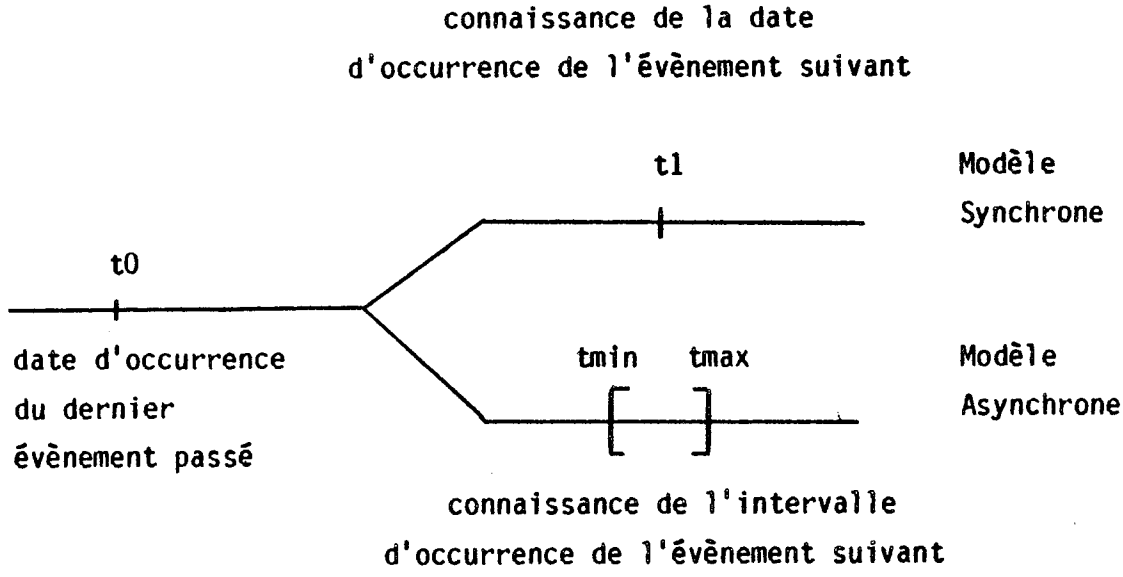
Sur ce sujet, on pourra se reporter au paragraphe II consacrée aux langages de descriptions de matériel dérivés de langage de programmation.

I.5 - Synchrone et asynchrone

Les termes de "synchrone" et "asynchrone" sont les plus difficiles à définir de manière précise. Pour commencer, on peut remarquer que leur existence commence avec la rupture entre le caractère continu des évolutions des systèmes réels (modulo des quantifications fines hors de propos ici) et la discrétisation du temps dans leurs modélisations.

La notion de temps discret implique que toute évolution du système (évènement) peut se ramener, au moins a posteriori à une date d'occurrence définie sur un ensemble isomorphe à l'ensemble des naturels, les notions de modèles synchrones et asynchrones étant deux abstractions différentes d'une même réalité.
Modélisation synchrone et asynchrone peuvent se définir alors par : connaissant l'histoire passée d'un système jusqu'à une date t0, si l'on spécifie l'occurrence de l'évènement suivant sur ce système sous forme d'une date précise (dans le futur) alors on a affaire à une modélisation synchrone ; modélisation asynchrone correspondant alors à une date d'occurrence définie comme appartenant à un intervalle de temps ($\lfloor tmin, tmax \rfloor$), sans plus de précision.

On peut considérer le schéma suivant :



Remarques

- Un modèle asynchrone est plus proche de la réalité qu'un modèle synchrone ; par contre, sur le plan de la précision au sens classique, ce dernier est moins précis que le précédent.

- Les modèles synchrones peuvent être considérés comme des cas particuliers des modèles asynchrones pour lesquels on a tmin = tmax.

- Quel que soit le modèle, on garde une relation de causalité stricte.

- La définition de synchrone en référence à des objets particuliers du type horloge n'a pas de sens autre que celui donné par l'habitude : on peut envisager des modélisations asynchrones dans lesquelles les intervalles ⌊tmin,tmax⌋ sont repérés par une horloge dont la seule "fonction" est d'assurer le passage entre temps continu et temps discret (échantillonage du temps).

- Un des domaines privilégiés des langages asynchrones (au sens de langage permettant des modélisations asynchrones) est le temps réel pour lequel on doit spécifier des contraintes du type "envoyer la commande C dans au moins 5 secondes et au plus 7 secondes".

Exemples

- Pour les modèles asynchrones, on peut citer les réseaux de Pétri classiques ainsi que le GRAFCET muni des notions d'action différée et de franchissement à échéance limite ([MOA 81]). D'un autre coté on trouve les langages dérivés des concepts de CSP : OCCAM et ADA (nous reviendrons sur ces langages au cours des paragraphes suivants).

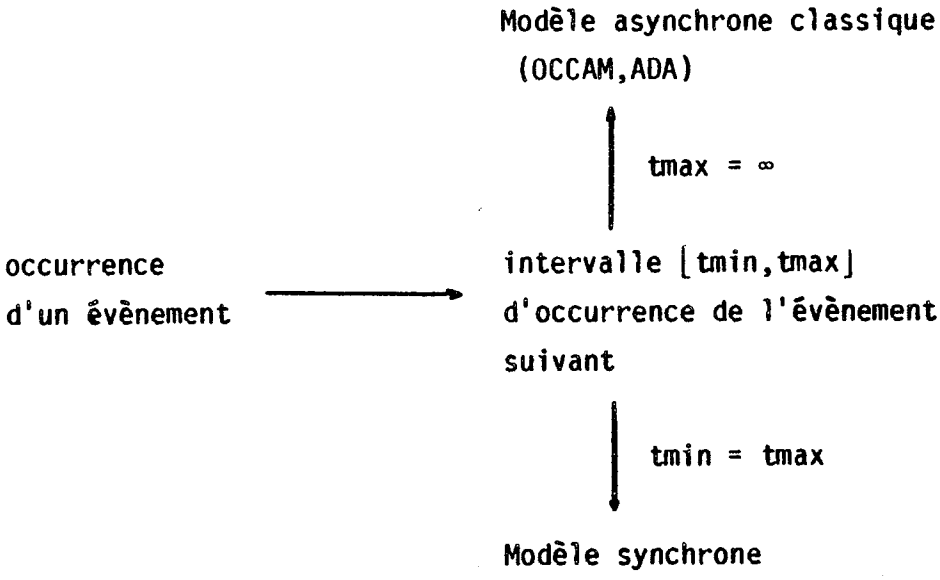
- Pour les modèles synchrones, on peut citer les langages LUSTRE et CADOC sur lesquels nous reviendrons aussi.

Remarques

- Les langages asynchrones classiques implémentent l'asynchronisme au travers d'opérateurs du type WAIT, ce qui correspond à la réalisation du tmin de l'intervalle (notion de "au moins à la date t"). Par contre la notion de "au plus à la date t" est en général non implémentée ; cette implémentation étant non triviale car spécifier une date maximale d'occurrence pour un évènement

implique l'existence d'un mécanisme s'activant à cette date et générant l'évènement à la date tmax s'il n'a pas été généré avant. Sachant que la durée d'exécution ε d'une action (action de génération de l'évènement par exemple) est non nulle même si infiniment petite (négligeable), si le processus de génération de l'évènement est activé à tmax, l'évènement proprement dit ne sera émis qu'à tmax + n.ε, n dépendant du nombre de calculs élémentaires à effectuer. Le problème du temps réel se ramène à savoir quand n.ε devient non négligeable. En d'autres termes, la notion de "au plus" est vérifiable a posteriori mais non calculable.

- On peut considerer le schéma suivant :



- Toutes ces notions peuvent s'illustrer sur l'équivalence entre

WAIT 2s;WAIT 3s et **WAIT 5s**

- On peut parler de synchronisation dans les modèles asynchrones (mécanismes de rendez vous) mais il est sans signification de parler de synchronisation dans des modèles synchrones.

- Asynchronisme et indéterminisme : ces deux termes ne sont pas équivalents ; on peut avoir des modèles asynchrones non déterministes (OCCAM) ainsi que des

modèles synchrones non déterministes (Reseaux de Pétri avec interprétation synchrone). Dans un modèle indéterministe asynchrone, on a deux niveaux de non connaisance : non connaissance de l'action à effectuer et non connaissance des dates d'exécution de l'action choisie.

- Pour revenir au problème de la circuiterie, un modèle synchrone peut être rendu asynchrone par introduction des valeurs min et max des délais d'évolution en remplacement des valeurs typiques.

II - LANGAGES NON SPECIFIQUES A LA DESCRIPTION DU MATERIEL

II.1 - Langages Impératifs

Nous nous bornerons ici à présenter un exemple d'application à la description du matériel d'un langage impératif classique : Pascal

II.1.1 - (ADLIB +SDL) - HELIX

Le système HELIX ([SIL 83]) a pour base deux outils développés à l'Université de Stanford : ADLIB et SDL ([HIL 79, HIL 80, COR 81]) ; le premier de ces outils est une extension de Pascal et est utilisé pour la modélisation du comportement des circuits (descriptions de modules), l'autre (SDL) est utilisé uniquement pour décrire l'interconnexion des modules (aspect structurel) ; l'interconnexion entre les deux outils se faisant par l'intermédiaire du système SABLE. La structure du système complet est donnée figure 1.

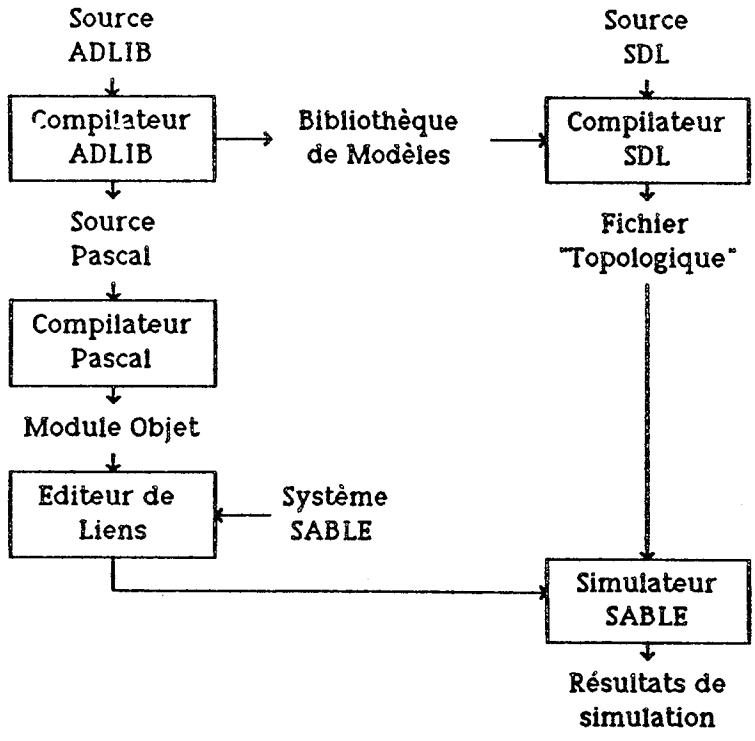


figure 1 : ADLIB, SDL, SABLE : organisation générale du système

Dans une première partie, nous présenterons le langage ADLIB puis nous introduirons rapidement les facilités de structuration offertes par SDL ; la

conclusion de cette partie sera constituée de remarques sur (ADLIB - SDL) ou HELIX.

II.1.2 - ADLIB

Le choix de Pascal comme base d'un langage de description du matériel a été motivé par les raisons suivantes : importante communauté d'utilisateurs, typage fort permettant des vérifications statiques poussées, facilité d'utilisation (lecture, écriture et compréhension des descriptions) et possibilité d'ouverture vers la vérification automatique de circuits à partir des techniques de preuve de programmes.
ADLIB étant une extension de Pascal, nous ne décrirons pas ce qui concerne la partie déclarative d'une description ADLIB (types, variables et vérifications effectuées à la compilation sont directement dérivées de Pascal) mais nous allons présenter les primitives de contrôle (instructions) disponibles. Ces primitives peuvent être regroupées en 2 catégories :

i) Primitives Pascal

Ce sont les primitives clasiques : if-then-else, case-of, while-do, repeat-until, for-do et goto ; la sémantique de ces primitives étant classique nous n'insisterons pas.

ii) Primitives ADLIB

* **Affectation avec délai**

assign ‹expr› to ‹nom_de_port› ‹clause_temporelle›

ou :

- ‹expr› est une expression évaluée à l'instant courant, stockée dans une mémoire tampon et affectée à la variable spécifié par ‹nom_de_port› à l'instant indiqué par la clause temporelle ; cette expression peut contenir des appels à des fonctions prédéfinies ou construites.

- ‹nom_de_port› désigne un port d'E/S du circuit
(une variable interne sera affectée par la commande classique :=)

- ‹clause_temporelle› spécifie l'instant relatif au temps courant d'affection de la valeur de ‹expr› à la variable ‹nom_de_port› ; par défaut ce temps est 0.0 . Cette clause peut faire référence à la variable time representant le temps courant de simulation ou à une horloge et à ses phases définies par l'utilisateur (sync et phase).

Exemple

assign r.carry to line_1 sync ck1 phase 1

signifie que la valeur portée par r.carry sera affectée à line_1 au premier front montant de la phase 1 de l'horloge ck1.

*** Attente d'évènement**

waitfor ‹expr_booléenne› ‹clause de contrôle›

Cette primitive bloque l'évolution courante tant que la condition spécifiée dans ‹expr_booléenne› est fausse ; la clause de contrôle servant à spécifier les instants d'évaluation de l'expression : instants définis soit par une période d'échantillonnage (delay), soit par une évènement (sync et phase), soit par une liste de ports d'Entrée/Sortie (check) ; dans ce dernier cas, toute variation sur l'un des ports spécifiés entrainera l'évaluation de ‹expr_booleenne›.

Exemples

```
Waitfor current > 0.01   delay période_échantillon
Waitfor ack = 1          Sync ck phase 3
Waitfor a and b          Check (a,b)
```

*** Masquage - démasquage - blocage**

les procédures de démasquage (sensitize) ou de masquage (desensitize) sont utilisées pour rendre visible ou masquer les variations de certains ports d'E/S à une description.
La procédure detach bloque l'exécution d'une description tant qu'il n'y a pas de variations sur une des variables auxquelles la description est sensible.

Exemple Porte NAND (entrées a et b, sortie s)

```
begin
  sensitise (a,b) ; ← sensibilisation de la porte à ses entrées.
  repeat
    detach                          ← attente de variation de a ou b.
    assign not (a and b) to y
  until false
end
```

Remarque
Ces instructions peuvent être décrites en termes de l'instruction d'attente d'évènement (waitfor), le choix se faisant uniquement sur des critères d'efficacité de simulation.

*** Notion de sous-processus**

Si l'on considère la description d'un module comme un processus, il est alors possible de définir des sous processus, réalisant des fonctions simples et executés indépendamment du processus principal.

Les sous processus sont de 2 types : upon et transmit et sont désignés par un nom.

- upon ‹expr_booleenne › ‹liste_de_ports› do ‹instructions›

à chaque variation d'un port spécifié dans la liste, l'expression booleenne est évaluée, si elle est vérifiée la liste d'instructions associée est executée

- transmit ‹expr› ‹liste_de_ports› to ‹nom_de_port› ‹clause temporelle›

cette construction est équivalent à

upon true ‹liste_de_ports› do assign ‹expr› to ‹nom de port›

L'activation des sous processus est contrôlable par l'intermédiaire des procédures d'activation et d'inhibition (permit, inhibit) dont le paramètre est un nom de sous processus. Une fois un processus activé, l'exécution des instructions le composant est non contrôlable.

II.1.3 - SDL

La prise en compte de l'aspect structurel d'une description se fait par l'intermédiaire de SDL (Structural Design Language) ; la saisie d'un circuit composé se faisant au travers d'outils graphiques, ce langage est transparent à l'utilisateur.

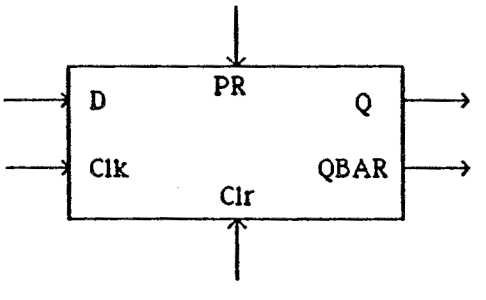
II.1.4 - DESCRIPTIONS DE CIRCUITS EN ADLIB/SDL

Un circuit est décrit comme un assemblage structurel (via SDL) de modules (via ADLIB) ; un modèle de circuit est appelé comptype, un comptype fonctionnel étant formé de la liste des instructions ADLIB modélisant son comportement.

La structure d'une description ADLIB est la suivante :

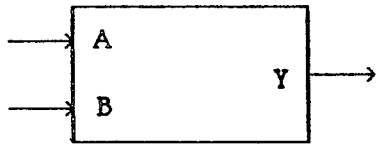
```
comptype <nom> <liste des paramètres>
     default
        ... ← valeur par défaut des paramètres
     type
        ... ← déclaration des types construits (Pascal)
     inward
        ...
     outward
        ...
     bothward
        ...
     internal
        ...
     var
        ...
     label
        ... ←  étiquettes de branchement des instructions goto
     ...    ← déclaration des procédures et fonctions locales
     subprocess
     ... ← déclaration des sous processus
     begin
     ... ← corps de la description
     end
```

Exemple 1 : Bascule D avec Preset et Clear ([SIL 83])



```
comptype DFF
     inward CLR,PR,D,CLK : boolean ;
     outward Q,QBAR : boolean ;
     subprocess
       clock-output :
          (* propagation de D sur Q et QBAR au front montant de CLK *)
          upon CLK and PR and CLR check CLK do
             begin
                assign D to Q delay 65 ;
                assign not D to QBAR delay 65 ;
             end ;
       preset-output :
          upon not PR check PR do
             begin
                assign true to Q delay 80 ;
                assign false to QBAR delay 50 ;
             end ;
       clear-output :
          upon not CLR check CLR do
             begin
                assign false to Q delay 50 ;
                assign true to QBAR delay 80 ;
             end ;
     begin
       assign false to Q ;
       assign true to QBAR ;
         (* initialisation de Q et QBAR *)
         (* l'évolution se fera par les sous processus *)
```

Exemple 2 : porte NOR à 2 entrées



```
Comptype NOR 2 (td : integer)
     default
       td = 0
     inward A,B : logic_net (* type enuméré prédéfini {Hi,Lo,Unk,Z} *)
     outward Y : Logic_net
     Subprocess
       nor2_action :
          upon true check A,B do
            begin
              if (A=Unk) or (A=Z) or (B=Z) or (B=Unk) then
                assign Unk to y delay td
              else
                if (A=Hi) or (B=Hi) then
                  assign Lo to Y delay td
                else assign Hi to Y delay td
            end ;
```

La modélisation proposée dans [COR 81] étant :

```
Comptype NOR ;
     inward A,B : boolnet ;
     outward Y : boolnet ;
     begin
       while true do begin
         assign not (A or B) to y ;
         waitfor true check a,b ;
       end ;
     end ;
```

Remarque : dans les modélisations proposées récemment, chaque mode d'activation du circuit est représenté par un sous-processus, le corps de la description étant uniquement utilisé pour les initiations.

II.1.5 - MANIPULATION DU TEMPS

La manipulation explicite du temps se fait en ADLIB au travers de l'existence d'une horloge de base délivrant la suite de valeurs 0,1,2,... A partir de cette référence absolue le concepteur peut définir des horloges locales utilisées au travers des primitives sync et phase.

Exemple

clock phi (10,4)

définit une horloge phi à 4 phases de durée 10.

II.1.6 - REMARQUES SUR ADLIB/SDL/HELIX

- Le masquage de l'aspect structurel des descriptions par les outils de saisie graphique est justifié par le fait que l'écriture manuelle de listes de connexions est source importante d'erreurs.

- La notion de **traducteurs** définie dans ([HIL 80, COR 81]) permet dans le cadre d'une conception descendante de passer d'un type abstrait (entier par exemple) à des types plus proches du matériel (booléen, logic net,...).

- Différents comptypes ayant des caractéristiques communes peuvent être regroupés en modules ; dans un module on trouve la définition de la correspondance entre horloge de base et temps extérieur, la définition des types globaux et des horloges. Cette facilité permet la création de bibliothèques de descriptions suivant les technologies utilisées.

- Dans les exemples de descriptions de circuits on a déjà remarqué les variations du style de modélisation, une des possibilités étant d'avoir autant de processus parallèles que de modes d'activation du circuit. Il faut noter que les mécanismes de communication et de synchronisation entre sous processus sont peu explicites (résultat de l'activation simultanée de deux sous processus travaillant sur des variables communes ?), ce type de problème étant liés à l'utilisation sous jacente de Pascal qui ne possède pas les notions de parallélisme de synchronisation de sous processus. Pour terminer ajoutons que la spécification du parallélisme n'est pas naturelle (création puis activation de sous processus).

- Aucun mécanisme de vérification autre que ceux liés à la syntaxe Pascal ne semble prévu, les vérifications étant effectuées en amont de la simulation (pas de notions d'assertions en cours de simulation).

- Il n'existe pas d'équivalent satisfaisant de la notion d'affectation de chronogrammes, les affectations étant faites au travers d'une syntaxe assez lourde (assign, delay, check) et portant uniquement sur un couple (valeur, date). Sur ce point on se reportera à la notion d'affectation généralisée définie dans CADOC.LD.

- Le fait d'utiliser Pascal comme base entraine la nécessité de redéfinir des types plus adéquats pour la modélisation de variables à signification matérielle ainsi que les opérations sur ces types.

- La description de la circuiterie MOS et des problèmes y afférent n'est pas abordée (cf. chapitre 3).

II.1.7 - AUTRES APPROCHES

Parmi les applications de langages impératifs, on peut citer XI ([FELD 83]) basé sur C ainsi que l'utilisation de modula-2 dans [ROB 83]. Notons simplement que Modula 2 est un langage purement asynchrone, sans aucun mécanisme prédéfini de synchronisation, la seule notion existante étant celle de variable partagée.

II.2 - Langages Parallèles

Dans la catégorie des langages conçus pour exprimer les notions de processus et de communications entre processus nous distinguerons deux classes : les langages du type réseau de Pétri (RdP) et les langages parallèles. Une étude des différentes approches possibles illustrées sur l'exemple du protocle du bit alterné peut être trouvé dans [ROI 84], une étude des formalismes Grafcet et RdP dans [MOA 81]. En ce qui concerne les langages du type RdP nous présenterons ultérieurement le langage CAP/DSDL ainsi que CADOC.LD, objet de cette étude.
Dans un premier paragraphe, nous allons présenter brièvement le modèle CSP ([HOA 78], [DIJ 75], [ROI 84]), puis nous introduirons OCCAM ([MUN 83], [MAY 83]) et nous terminerons par le langage ADA ([VER 82]).

II.2.1 - MODELE CSP

Dans le modèle proposé par Hoare, la notion de base est la notion de processus, différents processus pouvant être regroupés en commandes parallèles ; lors de l'activation d'une commande parallèle, tous les processus la composant seront exécutés en parallèle, leurs vitesses d'exécution relatives étant arbitraires. Pour éviter des conflits d'accès sur des variables communes, les processus apparaissant dans une commande parallèle ne peuvent pas utiliser de variables partagées (processus disjoints) ; la communication entre processus se faisant par des processus spécialisés.

1) Communication entre processus

La communication entre 2 processus se fait par l'intermédiaire de 2 processus (commandes) spécialisés : un processus d'entrée noté ? et un processus de sortie noté !, ces processus travaillant sur des variables.

Une communication entre 2 processus parallèles P1 et P2 est établie lorsque :
- une commande d'entrée de P1 spécifie P2 comme source,

- une commande de sortie de P2 spécifie P1 comme source,
- l'expression générée par la commande de sortie est compatible avec la (ou les) variable(s) spécifiée(s) dans la commande d'entrée.

Dans le cas ou l'un des 2 processus n'est pas prêt pour la communication, l'autre processus se bloque en attente.

Exemple :

Processus P	Processus Q
.	.
Q?(X)	P!(exp)
.	.

Si exp est compatible avec X, le résultat de l'exécution simultanée des 2 commandes est équivalent à l'exécution de l'affectation classique X := exp.

ii) non déterminisme

Le concept de non déterminisme dans l'exécution d'une liste d'actions [DIJ 75] est rendu possible par l'introduction de commandes gardées.

Une commande gardée est composée :
- d'une garde (condition booléenne)
- d'une liste de commandes associées.

Les commandes gardées apparaissent dans les constructions conditionnelles ([‹liste de commandes gardées›]) et répétitives (* ‹liste de commandes gardées› enddo) ; deux commandes gardées dans une même liste étant séparées par le symbole [ ].

Dans la construction conditionnelle, on sélectionne de manière non déterministe une des commandes gardées à garde vraie ; après exécution de la liste de commandes associée, la commande conditionnelle est terminée.

Dans la construction répétitive, tant qu'il existe des commandes gardées à garde vraie, on les exécute de manière indéterministe ; la commande conditionnelle étant terminée lorsqu'il n'y a plus de gardes vérifiées.

Nous ne chercherons pas ici à donner plus de détail sur CSP et nous allons présenter maintenant une des implémentations existantes : le langage OCCAM.

II.2.2 - OCCAM

Le langage OCCAM est une réalisation directe des notions développées dans CSP et s'articule autour des notions de processus et de communication entre processus.

Un programme OCCAM est construit à partir de processus élémentaires et de constructeurs, deux processus communiquant par l'intermédiaire de composants élémentaires de synchronisation (ou canaux) qui sont les seuls objets pouvant être partagés entre processus.

1) Processus élémentaires

Les 4 processus élémentaires en OCCAM sont l'affectation, l'entrée, la sortie et l'arrêt.

Exemples

* "x := 1" :
  processus se terminant après avoir attribué la valeur 1 à la variable locale x.

* "cardreader?cardimage" :
  processus qui attend pour s'exécuter qu'un processus de sortie utilisant le canal cardreader soit prêt à s'exécuter en parallèle.

* "C[i]!a" :
  processus qui lorsqu'une communication sera établie avec un autre processus ecrira la valeur courante de a sur le ième élément du canal c.

* "wait now after delay" :

processus qui sera bloqué tant que la valeur du temps local sera inférieur à la valeur de "delay". Nous reviendrons ultérieurement sur la notion de temps en OCCAM.

ii) **Constructeurs**

A partir des processus précédents on peut construire des processus plus complexes en utilisant les constructeurs suivants :

- SEQ : constructeur séquentiel,
- WHILE : constructeur répétitif,
- PAR : constructeur parallèle,
- ALT : alternatif,
- IF : conditionnel,
- FOR : multiplexeur

Dans le cas d'un constructeur alternatif, chacune des alternatives est une commande gardée (ou processus gardé), la sélection de la commande exécutée sera faite de manière indéterministe (il en est de même pour le constructeur répétitif);

En OCCAM, la garde d'un processus est une expression booléenne complétée éventuellement d'une commande d'entrée.

iii) **Déclarations/typage**

OCCAM ne possède pas de notion de type de données explicite ; à chaque processus est associé l'ensemble de ses canaux (CAN), l'ensemble des variables locales (VAR), l'ensemble des constantes locales à ce processus (CON), l'ensemble des processus locaux (PROC).

iv) **Descriptions de circuits en OCCAM**

Si le formalisme OCCAM est originellement destiné à la description de processus à un niveau d'abstraction assez élévé (protocoles de communications entre systèmes,...), certaines recherches sont faites pour essayer d'appliquer ce langage à la description d'architectures de circuits. La

présentation que nous allons donner ici de descriptions de circuits en OCCAM est inspiré de [MUN 85] et est asez fragmentaire ; pour des raisons de place, il nous est impossible de citer ce qui est actuellement fait par exemple dans le cadre d'architectures systoliques.

Nous allons considérer une cellule d'additionneur composée d'une porte NAND et de deux demi additionneurs (figure 2a) eux même composés de 4 portes NAND (figure 2b). Cet exemple va ensuite nous permettre de faire un certain nombre de remarques.

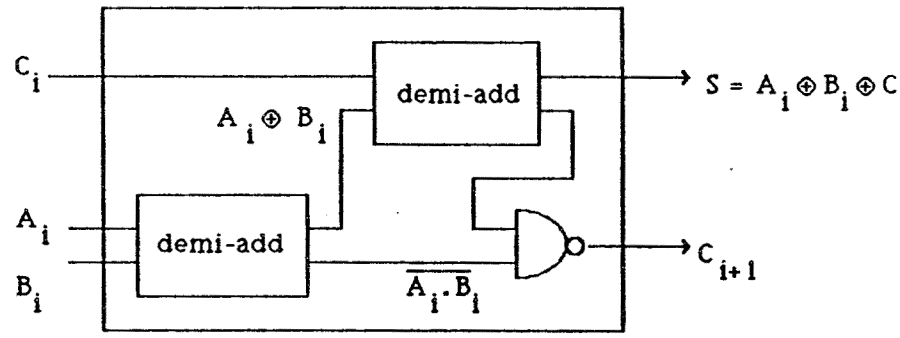


Figure 2a : Additionneur

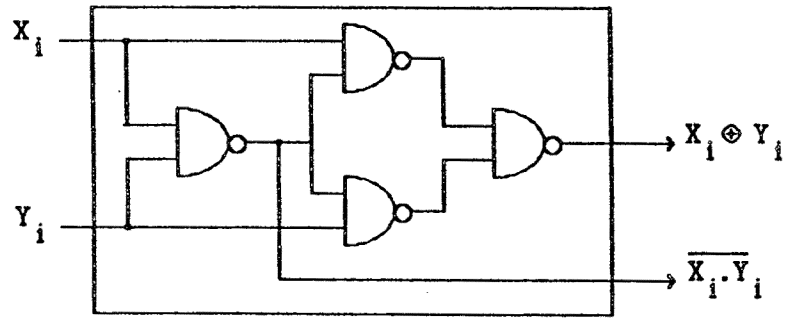


Figure 2b : Demi additionneur

* **description d'une porte NAND**

La description OCCAM d'une porte NAND fait intervenir la définition de 4 canaux (figure 3) : 3 canaux d'E/S classiques (E0,E1,S) et un canal pour arrêter le processus associé (STOPPER).

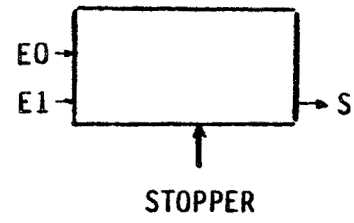


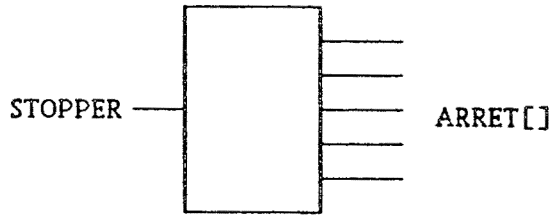
figure 3 : porte NAND

```
proc NAND (chan STOPPER,E0,E1,S) =

  def TAB.NAND = table [1,1,1,1,0,2,1,2,2]
     -- table modélisant les opérations effectuées par la porte :
     -- 1 ← VRAI, 0 ← FAUX, 2 ← U
  var A,B,FINI -- variables locales

  seq -- les processus suivants vont s'exécuter en séquence
     A := 2
     B := 2 -- A et B mis dans l'état U.
     STOPPER ? FINI -- lecture sur le canal "STOPPER",
                    -- résultat mis dans la variable locale FINI
     while not FINI
           seq
             S!TAB-NAND [(A*3)+B]
                    -- génération de la sortie
                    -- ex. : A = 2 et B = 2 → S = TAB_NAND [8] = 2
                    --         → non U.U = 1
                    --        A = 0 et B = 1 →S = 1
                    --         → non 0.1 = 1
             E0?A
             E1?B -- lecture des canaux d'entrée
             STOPPER?FINI
```

* Description d'un demi-additionneur

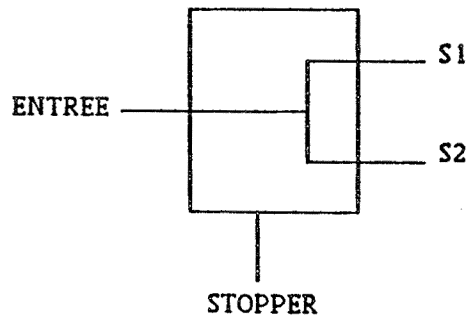
Avant de donner la description du demi additionneur nous allons d'abord définir 2 processus ARRETER et CONNEXION dont la nécessité sera expliquée ultérieurement.

- processus ARRETER




```
proc ARRETER (value NB, chan STOPPER, ARRET [ ]) =
     -- NB est un paramètre formel
  var FINI
  seq
     FINI := false
     while not FINI
            seq
              STOPPER ? FINI
              par I = [0 TO NB.BIT.MAX]
                    if I < NB → ARRET [I]!FINI
                    true →    SKIP :
                      -- éclatement de l'information lue sur
                      -- le canal STOPPER sur NB canaux ARRET
```

- processus CONNEXION



```
proc CONNEXION (chan STOPPER, ENTREE, S1, S2) =
  var FINI,A
  seq
     STOPPER ? FINI
     while not FINI
            seq
              ENTREE ? A
              par
                    S1 ! A
                    S2 ! A
              STOPPER ! FINI ;
                 -- éclatement de l'information lue sur ENTREE vers S1 et S2
```

Tout ceci étant défini on peut alors définir le processus modélisant un demi additionneur (figure 4) par :

```
proc DEMI.ADD (chan STOPPER,E0,E1,Q,S) =
  def NB.ARRET = 8  -- constante (8 sous processus)
  chan ARRET [NB.ARRET] :
  chan EQUI [8[,C1,C2 :  -- canaux internes
  par
     ARRETER (NB.ARRET, STOPPER, ARRET)
     NAND (ARRET[0], EQUI[0], EQUI[1], C1)
     NAND (ARRET[1], EQUI[2], EQUI[3], EQUI[6])
     NAND (ARRET[2], EQUI[4], EQUI[5], EQUI[7])
     NAND (ARRET[3], EQUI[6], EQUI[7], S)
     CONNEXION (ARRET[4], C1, C2, Q)
     CONNEXION (ARRET[5], C2, EQUI[3], EQUI[4])
     CONNEXION (ARRET[6], E0, EQUI[0], EQUI[2])
     CONNEXION (ARRET[7], E1, EQUI[1], EQUI[5]) :
```

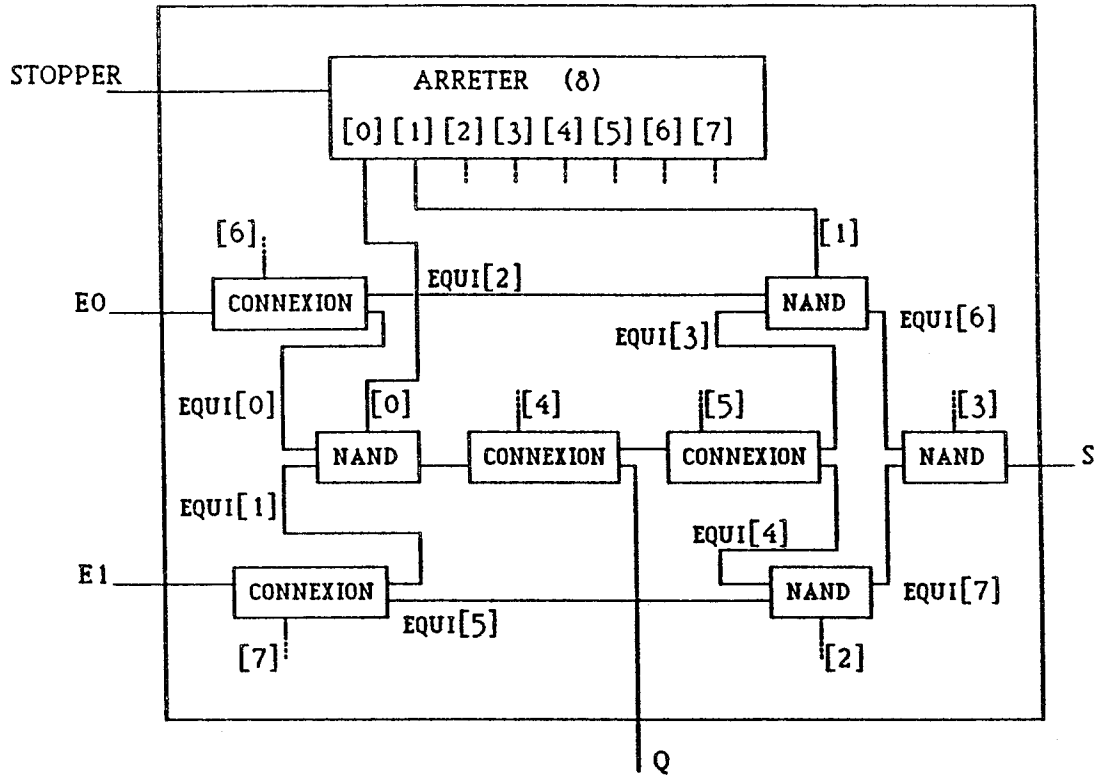


Figure 4

Décomposition du demi additionneur en terme de ses sous processus OCCAM

* Description d'un additionneur

Le processus modélisant un additionneur 1 bit (figure 5) est décrit par :

```
proc ADD.1 BIT (chan STOPPER,A,B,CE,CS,S) =
   def NB.ARRET = 3 :
   chan ARRET [NB.ARRET] :
   chan EQUI [3] :
   par
         ARRETER (NB.ARRET,STOPPER,ARRET)
         DEMI.ADD (ARRET[0],A,B,EQUI[0],EQUI[1]
         DEMI.ADD (ARRET[1],EQUI[1],CE,EQUI[2],S)
         NAND (ARRET[2],EQUI[0],EQUI[2],CS) :
```

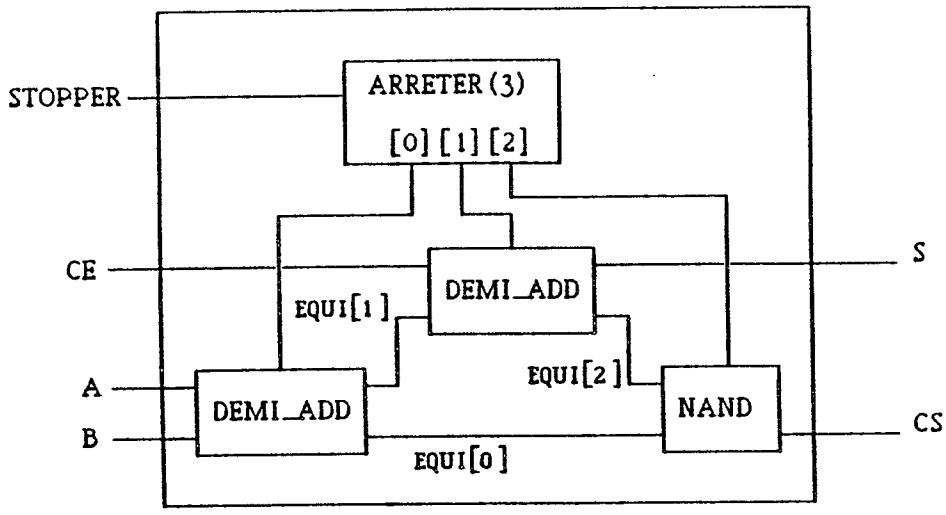


Figure 5

Decomposition de l'additionneur 1 bit en terme de ses sous processus OCCAM

v) Remarques

*** nécessité des processus de CONNEXION**

La sémantique d'OCCAM impose la duplication de l'information devant être émise vers plusieurs processus dans la mesure où ces processus doivent tous (et non un seul choisi de manière non déterministe) utiliser l'information considérée.

Considérons le schéma donné figure 6a, si on le décrit comme assemblage des processus OCCAM donné figure 6b (même canal partagé en entrée par B et C) alors l'évènement généré par le processus A sera consommé par le premier processus parmi B,C qui sera prêt à faire l'échange A!X → B?X ou C?X. La modélisation correcte nécessite la duplication de l'information générée par A et donc l'utilisation d'un processus de connexion (fgure 6c).

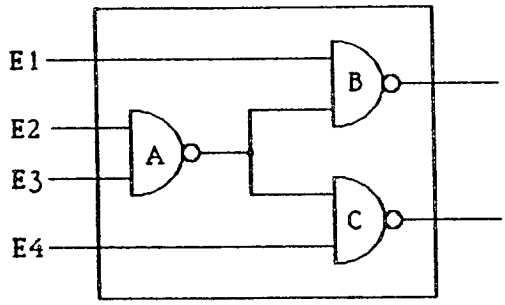


Figure 6a
Représentation logique

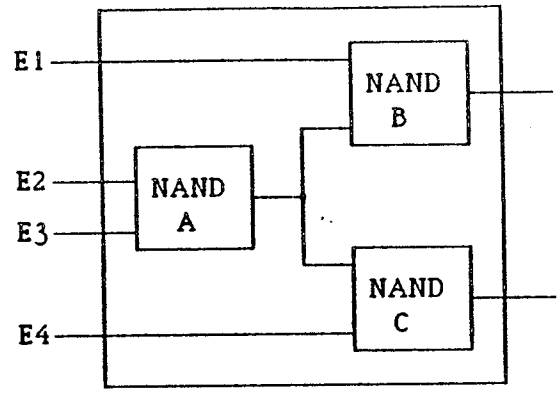


Figure 6b
Modélisation OCCAM "incorrecte"

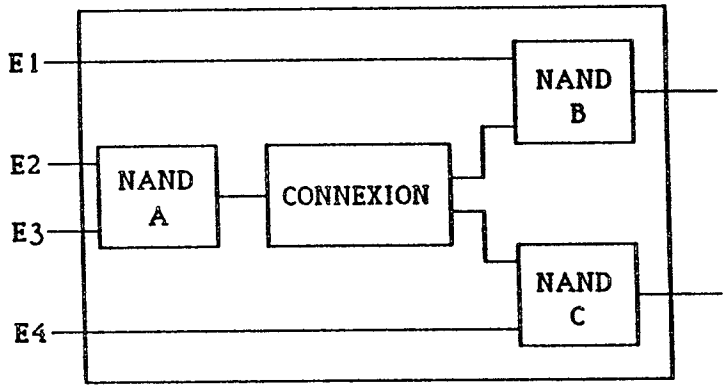


Figure 6c : Modélisation OCCAM "correcte"

La nécessité des processus CONNEXION s'explique par la différence entre la réalité électrique d'une valeur de connexion, valeur qui persiste dans le temps jusqu'à ce qu'une cause précise la fasse varier et la définition OCCAM d'un canal qui ne transporte que des évènements qui, une fois consommés par un autre processus, cessent d'exister sur le canal qui n'a plus alors de valeur (aspect instantané des valeurs des canaux OCCAM). Cette différence fondamentale, qui doit être contournée de façon artificielle (les processus de connexion pouvant être générés automatiquement lors d'une saisie du schéma à décrire, avec les difficultés évidentes dans le cas de dispositifs bidirectionnels) rend difficile l'utilisation de OCCAM pour les circuits fortement couplés.

*** Contrôle de la simulation**

Le circuit proposé étant de nature purement combinatoire, sa description OCCAM est une composition parallèle des processus décrivant chaque sous circuits (demi additionneurs puis portes nand). Par définition le processus global ne s'arrêtera que lorsque tous les sous processus parallèles seront terminés. Les sous processus étant des modèles de circuits combinatoires sont donc du type boucle infinie, pour qu'ils puissent se terminer il faut impérativement leur ajouter un canal de contrôle ; ce contrôle devant arriver à tous les processus entraîne la nécessité des processus ARRETER qui transmettent le contrôle d'arrêt de simulation jusqu'aux processus feuilles de la hiérarchie.
Cette nécessité peut être rapprochée de l'absence de manipulation explicite du temps en OCCAM (voir plus loin).

*** Absence de type**

Tous les canaux ou variables sont d'un même type qui correspond à une chaine de bits en machine, la valeur TRUE (resp. FALSE) étant définie comme le nombre où tous les bits sont à 1 (resp. à 0). Le seul type structuré est le type tableau.

Ceci entraine en particulier de devoir passer par des tables du type TAB.NAND donné précédemment pour modéliser certains types énumérés, méthode qui est fortement source d'erreurs et est peu lisible. De même l'absence de type structuré est sensible lorsque l'on veut donner des modèles de niveau d'abstraction élevé, plus lisible lorsque l'on dispose de la notion d'enregistrement (cas des structures microprogrammées). Pour terminer, le non typage des objets enlève toute les possibilités de vérification de la cohérence des connexions (on peut très bien affecter la valeur TRUE à un canal puis ensuite manipuler cette valeur comme un entier).

*** Modélisation du temps**

Les seules références au temps (horloge de référence) se font au travers de l'opérateur NOW ; la valeur retournée par NOW est un entier sur un mot machine qui est incrémenté de façon régulière mais de manière dépendante de la machine hôte.

NOW peut être utilisé comme horloge locale à un processus mais aucune relation ne peut être définie entre les valeurs produites par l'opérateur NOW dans deux processus parallèles.

Cela est conforme au besoin d'OCCAM en tant que langage de description de processus (communications par rendez-vous, processus parallèles sans variables communes autres que les canaux,...) qui ne nécessite pas de notion de temps global [OCC 83] mais entraîne qu'il est impossible de donner des descriptions de dispositifs faisant explicitement intervenir un temps absolu en utilisant le langage OCCAM.

II.2.4 - ADA

Au vu des richesses de ADA (parallélisme, mécanisme de compilation séparée bien adapté à la création de bibliothèques de programmes et donc de descriptions de circuits,...) des études sont faites ([BAR 84, BAR 85]) pour appliquer ce langage tel quel à la description de circuits.

Une étude précise de ADA ([VER 82]) est hors de propos ici : pour ce qui nous concerne seules quelques notions vont être présentées dans une première partie ; des exemples de descriptions de circuits et quelques remarques seront donnés ensuite.

1) Présentation rapide du langage

Un programme au sens ADA est composé d'un ensemble d'unités pouvant être compilées séparement.

Une unité peut être :

- une spécification de paquetage (ou de sous programme)
- un corps de paquetage (ou de sous programme)
- une sous unité (incluse dans une unité mère mais compilée séparement)

Une sous unité pouvant être :

- un corps de paquetage (ou de sous programme)
- un corps de tâche.

Du point de vue de la compilation on distingue les unités de compilation autonomes et séparées.
Une unité autonome (ou unité de bibliothèque) est développée de façon indépendante, l'accès à cette unité se faisant au travers de la clause WITH suivie du nom de l'unitée et placée en tête de l'unité utilisatrice.
Une unité séparée est une unitée normalement incluse dans l'unité mère, les occurences de l'unité séparée sont remplacées par une déclaration (souche) correspondant à cette unité ; une unité séparée est indiquée par la clause SEPARATE suivie de l'identificateur de l'unité mère, (sur le plan de la compilation, l'effet de la clause SEPARATE est d'inclure le contexte de l'unité mère dans le contexte de l'unité séparée).

Il est à noter que ces notions entrainent l'existence d'un ordre partiel à respecter au cours de la compilation ([TIA 85])

ii) Exemples de descriptions de circuits en ADA

De manière classique, un circuit va être décrit en ADA comme occurence d'un objet abstrait : un paquetage va définir la description d'un circuit donné et les opérations permises (création d'une occurence, construction d'objets composés, simulation,...) seront incluses dans la partie publique du paquetage

Exemple 1

Une description ADA d'un inverseur dont les ports sont "entrée" et "sortie" est la suivante ([BAR 84]) :

* Description de la spécification du paquetage associé à l'inverseur

```
with gestion_port ; use gestion_port
     (* cette clause donne l'accès aux procédures de gestion des ports d'E/S
     (* des circuits : affectation connexion, deconnexion (voir plus loin)

package gestion_inverseur is
  (* spécification (déclaration) des connexions externes de l'objet abstrait
  (* inverseur", déclaration de la fonction d'instantiation de cet objet et
  (* de la procédure de simulation associée

type
  inverseur_enreg is
           record
             entrée : port ;
             sortie : port   (* le type port étant défini dans gestion_port
            end record ;
type
  inverseur is access inverseur_enreg
     (* déclaration de la structure contenant les informations  associées au
     (* modèle d'inverseur (inverseur_enreg) et d'un  pointeur sur cette (*
     (* structure (inverseur)

function creation return inverseur_enreg ;
     (* fonction de création d'une occurence d'un inverseur

procedure simulate (v : in inverseur) ;
     (* cette procédure reçoit en entrée un pointeur sur une description
     (* d'inverseur, elle lit la valeur de  v.entrée et calcule v.sortie *)
```

Corps du paquetage associé à l'inverseur

```
with gestion_port ; use gestion_port
package body gestion_inverseur is

  function création return inverseur is
     begin
           return new (inverseur_enreg) ;
     end creation ;

  procedure simulation (v : in inverseur) is
    begin
       case valeur (v.entrée) is
         when haut => affecter (v.output, bas) ;
         when bas  => affecter (v.output, haut) ;
         when indéfini => null ; (* pas d'actions à effectuer *)
       end case
        (* les fonctions valeur() et affecter() permettent d'accèder à la
        (* valeur d'un port et de modifier cette valeur propagation à tous
        (* les ports connectés ; ces fonctions sont définies dans gestion
        (* port ainsi que le type énuméré (haut, bas, indéfini)
    end simulation
end gestion_inverseur
```

*** Remarques**

- Le mécanisme de simulation est spécifié de manière directe dans la description ; à la différence des langages de description du matériel du type CADOC.LD il n'y a pas à proprement parler de simulateur : la simulation d'une description ADA revient à exécuter la procédure associée ; nous verrons ultérieurement dans un exemple plus complexe les difficultés qui peuvent se présenter pour écrire
les procédures de simulation associées à des dispositifs composés.

- Sur le plan de l'exactitude du modèle proposé il serait préférable de transformer "when indéfini → null" par "when indéfini → affecter (v.output, indéfini).

- On peut déjà remarquer la difficulté d'utilisation de ce langage liée à sa syntaxe complexe.

exemple 2

Nous allons nous intéresser ici à la description en ADA d'un verrou D dont la structure interne est donnée ci-dessous (figure 7).

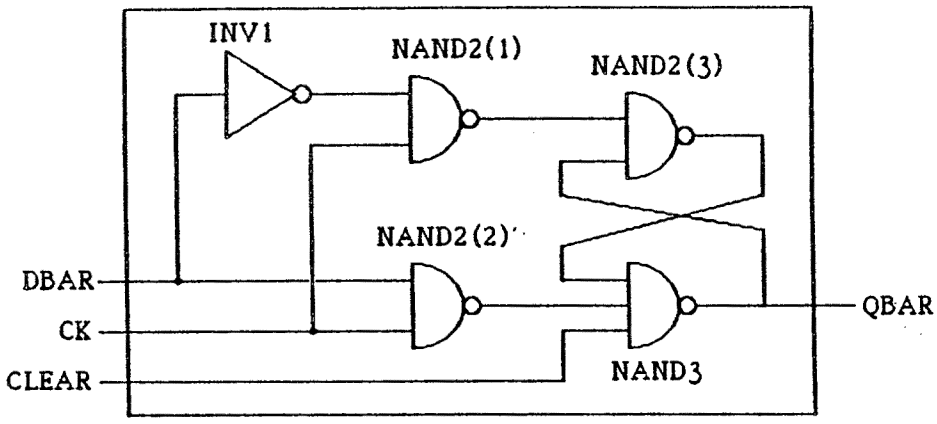


figure 7 : Verrou D

En supposant que les modules suivants aient été décrits :

- gestion_port : paquetage de gestion des ports d'interconnexion,
- gestion_inverseur : paquetage de description d'un inversseur (voir exemple 1),
- gestion_nand2 : paquetage de description d'un nand à 2 entrées,
- gestion_nand3 : paquetage de description d'un nand à 3 entrées.

* **Spécification du paquetage associé au verrou D**

```
package gestion_D latch  is
  type composants_Dlatch is private
       (* la structure interne est masquée à l'utilisateur
  type Dlatch_enreg is
       record
         ck : port ;
         dbar : port ;
         dbar : port ;
         qbar : port ;
         composants : Dlatch_composants ;
       end record ;
  type Dlatch is access Dlatch_enreg ;

  function creation return Dlatch ;
  procedure construction (Dlt : in Dlatch) ;
      (* la construction d'un verrou D se fera par création (des éléments
      (* constituants) puis par construction (interconnexion de ces éléments

  procédure simulation (Dlt : in Dlatch) ;
  private
     type nand2_tab is array (1..3) of gestion_nand2 . nand2
     type composants_Dlatch is
        record
          nand2_int : nand2_tab ;
          nand3_int : gestion_nand3.nand3 ;
          inverseur_int : gestion_inverseur.inverseur
        end record ;
end gestion_Dlatch
```

* **Corps du paquetage associé au verrou D**

```
package body gestion_Dlatch is
  function creation return Dlatch is
     begin
       return new (Dlatch_enreg) ;
     end creation ;
  procédure construction (Dlt : in Dlatch) is
  begin
     -- création des sous circuits
     for i in 1...3 loop
       Dlt.composant.nand2_int(i) := création ;
     end loop ;
     Dlt.composants.inverseur_int := création ;
     Dlt.composants.nand3_int := création ;

     -- renommage de certains ports (procédure défini dans le paquetage
     -- gestion port)
      renommer (Dlt.dbar, Dlt.composants.inverseur_int.entrée) ;
      renommer (Dlt.dbar, Dlt.composants.nand2_int(2).entrée(2)) ;
      ...

      -- interconnexions entre ports
      connecter (Dlt.composants.inverseur_int.sortie,
                 Dlt.composants.nand2_int(1).entrée(1°° ;
      ...
  end construction ;
```

```
procédure simulation (Dlt : in Dlatch) ;
  begin
     -- l'ordre de ce qui suit est important
     simulation (Dlt.composants.inverseur_int) ;
     simulation (dlt.composants.nand2_int(1)) ;
     simulation (Dlt.composants.nand2_int(2)) ;
     -- traitement des rebouclages combinatoires
     simulation (dlt.composants.nand2_int(3)) ;
     simulation (Dlt.composants.nand3_int) ;
     -- retraitement des premieres portes des rebouclages combinatoires
     -- voir remarques plus loin
     simulation (Dlt.composants.nand2_int(3)) ;
  end simulation ;
end gestion_dlatch ;
```

Avant de faire quelques remarques générales, nous allons rapidement présenter le mécanisme de gestion des ports et des interconnexions entre ports proposé dans ([BAR 84], [BAR 85]).
Dans cette approche, tous les problèmes de connectique sont résolus au travers d'une structure globale assurant la correspondance entre une connexion et la liste des ports qu'elle relie ; toutes les opérations sur les ports et les interconnexions sont réalisées dans un paquetage de gestion de connectique (gestion_port) offrant les opérations publiques (la structure des données manipulée est masquée) suivantes :

- **connecter** : connexion de deux ports, création éventuelle d'une interconnexion et mise de ces deux ports dans la liste des ports qu'elle relie.
- **déconnecter** : déconnexion de deux ports.

Du fait des mécanismes de visibilité en ADA, l'opération de déconnexion est en général inutile ; la connexion étant faite au même niveau de la hiérarchie de la construction de l'objet composé. La seule utilisation de la deconnexion est d'autoriser la spécialisation (spécification) d'un objet à partir d'un modèle générique par rupture de connexions internes mais cette méthode est en conflit avec le principe d'une structure interne masquée et les notions classiques de visibilité et de hiérarchie.

- **renommer** : renommage d'un port, association d'un nom de port interne d'un objet (nom formel) avec un port externe (nom effectif).
- **denommer** : opération inverse du renommage : cette opération est définie dans [BAR 84] comme équivalente à une déconnexion.
- **valeur** : retourne la valeur d'un port (de l'équipotentielle associée).
- **affecter** : affectation d'une valeur à un port.
- **fan_out** : nombre de ports interconnectés à un port donné.

iii) Remarques

* surcharge

L'utilisation de procédures portant le même nom dans différents paquetages (création, simulation,...) est autorisée grâce aux mécanismes de gestion des conflits développés en ADA ([VER 82, chap. 7.2]). L'intérêt d'une telle surcharge est qu'il est possible de donner le même nom à des fonctions différent non par le traitement effectué mais par les types des données manipulées ; remarquons simplement que l'utilisation des surcharges peut nuire à la lisibilité des descriptions et que la compréhension des mécanismes de visibilité et de masquage rend leur utilisation difficile pour un utilisateur non informé.

* Mécanisme de simulation

Nous allons revenir ici sur la manière dont est écrite la procédure de simulation associée au verrou D (exemple 2 précédent).
Le principe est de simuler dans l'ordre l'inverseur INV1, la porte NAND2(1), et la porte NAND2(2) puis de simuler ensuite une première fois NAND2(3) ainsi que NAND3 ; la procédure se termine après avoir simulé une deuxième fois la porte NAND2(3) du fait du rebouclage combinatoire.
L'ordre de simulation a donc été :

- INV1
- NAND2(1)
- NAND2(2)
- NAND2(3)
- NAND3 → à ce niveau, en supposant qu'il n'y ait pas d'oscillations (?) QBAR est correct.
- NAND2(3)

Il est évident qu'une telle méthode suppose la connaissance (l'analyse) du flot de données pour connaitre l'ordre d'appel des différentes procédures de simulation ainsi que la connaissance, dans le cas des circuits fortement couplés, du nombre de boucles à effectuer pour avoir stabilisation. La complexité du problème devenant inextricable dans le cas de boucles imbriquées ou autres structures plus complexes.
Ces problèmes sont liés au fait que les mécanismes de simulation sont inclus dans la description du circuit : il n'existe pas comme dans les systèmes spécialisés type CADOC.LD de programme de simulation qui se charge des problèmes liés à l'interconnexions de différents sous circuits.
Une des conséquences est qu'une utilisation du type définition de bibliothèques de descriptions et interconnexions de cellules est impossible : la connaissance du **comportement local** des cellules n'entraine pas la connaissance du **comportement global** du circuit défini comme interconnexion de ces cellules. en d'autres termes, la fonction d'un circuit composé n'est pas la juxtaposition des comportements des circuits composants.

*** Initialisation des simulations**

Le mécanisme d'interface (description des stimuli extérieurs) n'est pas spécifié.

*** Création et Construction**

Les auteurs soulignent l'intérêt d'utiliser les mécanismes de ADA pour masquer par exemple les structures internes des circuits à l'utilisateur et de n'accéder aux modèles qu'au travers d'un ensemble de procédures (notion d'opérateurs sur des types abstraits). on peut alors se demander qu'elle est la raison de fournir à l'utilisateur les deux procédures de création et de construction alors que ces deux procédures son intrinséquement liées ; il semblerait plus logique de masquer la construction à l'intérieur de la création.

*** Topologie des circuits**

Les auteurs proposent la prise en compte des caractéristiques spatiales (dimensions, forme) d'un circuit par définition de nouveaux champs associés au type décrivant le circuit considéré. De façon similaire à ce qui est proposé dans CADOC.LD, la définition de nouveaux champs permet de prendre en compte des contraintes physiques telles que capacités d'entrée, délais,....
On se reportera au paragraphe sur les extensions de CADOC.LD pour les limitations de cette aproche.

*** Modélisation du temps**

Dans ce qui précéde, l'accent a été mis sur les possibilités offertes par ADA en ce qui concerne les descriptions structurelles et fonctionnelles non temporisées d'un circuit. Des modifications sur le paquetage de gestion des ports ([BAR 84]) permet d'associer des délais aux modèles.

Exemple : inverseur de l'exemple 1

```
procédure simulation (inv : in inverseur) is
  begin
     case valeur (inv.entrée,10) is
       when bas → affecter (inv.sortie,haut)
       ...
```

A la différence des simulateurs classiques "orientés vers le futur", les mécanismes proposés ici calculent les valeurs courantes par référence aux valeurs passées.

*** Parallélisme**

Il n'est pas fait référence dans les articles étudiés aux possibilités de parallélisme offertes dans ADA au travers des taches (on se reportera à [VER 82]), ces facilités étant proches de celles existantes dans le modèle CSP présenté précédemment.

*** Autres approches basées sur ADA**

Dans ce qui précède, nous avons vu une application directe de ADA à la description du matériel ; on se reportera au paragraphe III.4 pour la présentation du langage VHDL basé sur des concepts de type ADA mais possédant une syntaxe propre.

*** Conclusion sur ADA en tant que CHDL**

Ce qui a été présenté dans cette partie est basé uniquement sur l'approche proposée dans ([BAR 84], [BAR 85]) et ne peut donc être considéré comme un jugement définitif sur l'applicabilité de ADA à la description du

matériel ; ce qui a été discuté n'est pas le langage lui même mais l'utilisation présentée par les auteurs. .

En ce qui concerne donc l'approche présentée, elle semble être encore au niveau de la spécification ; les inconvénients présentés n'ont pas de contre-parties susceptibles de justifier l'utilisation de ADA dans ce domaine.

II.3 - Langages fonctionnels et langages déclaratifs

II.3.1 - INTRODUCTION

A la différence des langages évoqués précédemment (langages impératifs séquentiels) les langages fonctionnels et les langages déclaratifs n'ont pas de notion d'état, de séqentialité ou d'affectation (ce qui a pour conséquence la suppression des effets de bords) : un programme dans un langage déclaratif est un ensemble d'équations, un programme dans un langage fonctionnel est une fonction composée.

En ce qui concerne les langages fonctionnels on peut citer LISP (dans la mesure ou l'on n'introduit pas la primitive SET équivalente à l'affectation) et FP ([BAC 81]) qui utilise des formes fonctionnelles et définit une algèbre de programmes.

Notons simplement ici que ces langages se prètent bien à la manipulation mathématique des équations ou fonctions définissant un programme et sont donc adaptées à la vérification formelle (preuve) soit de programmes, soit de circuits ; ce dernier type d'application étant d'ailleurs rarement traité par les langages fonctionnels ou déclaratifs dont les buts initiaux sont autres.

Nous allons développer ce chapitre avec la présentation du langage déclaratif LUSTRE développé à partir des notions d'algèbre d'évènements ([HAL 84]), nous parlerons ensuite du langage LTS qui est une approche équivalente développée dans [MIL 84].

II.3.2 - LUSTRE

Conçu pour des applications temps réel, le langage LUSTRE ([BER 85]) est susceptible d'application à la modélisation et -dans une moindre part- à la simulation de circuits. Une des caractéristiques principales de LUSTRE est le fait qu'à une variable X est associée une suite potentiellement infinie de valeurs x1,...,xn,... (chacune de ses valeurs apartenant à un domaine D(x) défini par le type de x et étendu avec la valeur indéfinie nil).

En LUSTRE, toute variable doit être définie par une équation (langage déclaratif) de la forme "x = E" ou E est une expression définissant une

séquence $e_1,\ldots,e_n,\ldots$ de valeurs dans D(x) ; l'interprétation de "X = E" étant pour tout i=1,2,... $x_i = e_i$, ou : à chaque "instant" i, la valeur de X est égale à celle de E.

Il est important de souligner qu'une équation en LUSTRE est assimilable à une propriété qui sera vérifiée sur toute l'histoire des variables et non à un instant donné comme dans les langages impératifs classiques.

1) Expression LUSTRE

Une expression en LUSTRE est construite à partir de variables au sens précédent, de constantes (séquence infinie de la même valeur) et d'opérateurs.

Tous les opérateurs classiques (booléens, arithmétiques, choix, choix multiple) possèdent la signification habituelle mais étendue aux suites de valeurs.

Exemple
L'expression **si X > Y alors Z + T sinon U** représente une séquence dont le nième terme est égal à :

- nil si (x_n = nil) ou (y_n = nil)
- nil si ($x_n > y_n$) et ((z_n = nil) ou (t_n = nil))
- $z_n + t_n$ si ($x_n > y_n$) et ($z_n \neq$ nil) et ($t_n \neq$ nil)
- u_n si $x_n < y_n$

Aux opérateurs classiques s'ajoutent un certain nombre d'opérateurs spécialisés travaillant sur les histoires des variables.

* opérateur pre

Si E est une expression définissant la séquence $e_1,\ldots,e_n,\ldots$ alors pre(E) définit la séquence $nil,e_1,e_2,\ldots,e_n,\ldots$

* opérateur → (followed by)

Si $F = f_1,\ldots$ est de même type que $E = e_1,\ldots$
alors E → F définit la séquence $e_1,f_2,f_3,\ldots$
L'opérateur pre est en général utilisé pour l'initialisation d'histoires.

Exemples :

- "X = 1 → pre(X) + 1" définit une variable X dont la nième valeur x_n satisfait

$x_n = 1$ si $n = 1$

$x_n = x_{n-1} + 1$ si $n > 2$ (et donc $x_n = n$)

- "H = C → C and not pre(C)"
  indique les fronts montants de la variable booléenne C

*** opérateur when**

Si C est une variable booléenne, E une expression alors when (C) (E) est une expression définissant la séquence dont le nième terme est la valeur de E au nième instant ou C est vraie.

Exemple

C	T	F	F	T	T	F	F	T
E	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
X = when(C)(E)	e_1			e_4	e_5			e_8
	x_1			x_2	x_3			x_4

Remarques

- When est un opérateur de renumérotation ou de filtrage et permet d'associer une variable X et une horloge C ; X étant exprimé à partir de l'horloge C, la seule notion de temps accessible à X est la séquence des instants ou C est vraie (en conséquence, la question : quelle est la valeur de X quand C est fausse n'a pas de sens).

- On peut ajouter à la remarque précédente que 2 variables peuvent avoir la même séquence de valeurs sans être égales : une variable est donc caractérisée par sa séquence de valeurs et par son horloge associée (qui peut être toujours vraie : horloge fondamentale).

*** opérateur current**

L'opérateur current s'applique uniquement aux expressions faisant référence à une horloge et correspond à l'échantillonnage d'une variable filtrée.

Exemple

	C	T	F	T	T	F	F	T
	E	e_1	e_2	e_3	e_4	e_5	e_6	e_7
when(C)(E)		e_1		e_3	e_4			e_7
X = current(when(C)(E))		e_1	e_1	e_3	e_4	e_4	e_4	e_7
		x_1	x_2	x_3	x_4	x_5	x_6	x_7

*** opérateur every**

l'opérateur every permet à partir d'une expression E et d'une variable booléenne C de définir une expression (every (C)(E)) dont la séquence de valeurs est définie de manière non déterministe par : la nième valeur de every(C)(E) est l'une des valeurs prises par E entre le nième front montant (inclus) de C et le front descendant suivant (inclus) : nième cycle

Exemple

C	F	T	F	F	T	T	F
E	e_1	e_2	e_3	e_4	e_5	e_6	e_7

1^{er} cycle $2^{ème}$ cycle

X = every (C)(E) → x1 = e2 ou e3, x2 = e5 ou e6 ou e7.

ii) Structure d'un programme LUSTRE

La notion de base en LUSTRE est le noeud : un noeud est un sous programme recevant des variables d'entrée et définissant des variables de sortie (éventuellement des variables locales) par l'intermédiaire d'un système d'équations définies grâce aux opérateurs précédents.

L'appel à un noeud se fait sous forme fonctionnelle : si N est un noeud dont l'entête est :

$$\text{node } N\ (I_1:T_1;\ldots,I_p:T_p)\ \text{returns}\ (J_1:S_1;\ldots;J_q:S_q)$$

et si $E_1,\ldots E_p$ sont des expressions de type $T_1,\ldots,T_p$ alors $N(E_1,\ldots E_p)$ est un t-uple d'expressions de type $(S_1,\ldots S_q)$ calculées à partir des paramètres d'entrées $E_1,\ldots E_p$.

iii) Exemple de description de circuits en LUSTRE

Un des intérêts d'un langage du type LUSTRE étant de permettre la preuve de description de circuits, il nous semble intéressant d'illustrer rapidement cette possibilité sur une architecture systolique réalisant un produit de convolution ([PIL 85b]).

La conception d'une architecture systolique ([QUI 83]) à partir d'un ensemble d'équations définissant le problème peut se faire en suivant les étapes suivantes :

- réécriture des équations initiales sous forme d'équations récurrentes uniformes (ce qualificatif sera précisé ultérieurement).
- choix d'une fonction de cadencement des calculs.
- choix d'une architecture.

L'étape de réécriture se faisant dans l'objectif d'obtenir un système de calculs élémentaires communiquants de manière simple.

Pour chacune de ces étapes, nous allons donner la description LUSTRE associée et prouver formellement l'équivalence entre les descriptions associées à 2 étapes successives.

* Equation initiale

A partir de la suite x(0), x(1),... et des coefficients constants w(k), k = 0,...K on veut calculer les termes y(1),y(2),... définis par :

$$y(i) = \sum_{k=0}^{K} w(k)\, x(i-k) \qquad \textbf{équation (1)}$$

Remarque

y(i) est non défini pour $i < K$

Description en LUSTRE (modélisation (1))

```
var w array [0...K] of real
var x,y : real
```

$$Y = \sum_{k=0}^{K} \text{pre}^{\,k}(x)\, w(k) \qquad \textbf{modélisation (1)}$$

-- l'opérateur pre^k étant défini de manière récurrente par

$$pre^n(X) = pre^{n-1}(pre(X)).$$
$$pre^0(X) = X$$

-- les i premières valeurs sont à nil.

*** Première transformation**

Afin de mettre en évidence les calculs élémentaires (réalisés par les cellules du circuit systolique) et les communications entre ces calculs, l'équation précédente est réécrite sous la forme :

$$y(i,k) = y(i,k-1) + w(k)\ x(i-k) \quad k=0,1,...K$$
$$y(i,-1) = 0 \qquad i > 0 \qquad \textbf{équation(2)}$$
$$x(i-k) = 0 \qquad i < K$$

Les transferts de données à effectuer dans l'architecture réalisant ces calculs n'étant pas locaux et réguliers (équation non uniforme) on réécrira l'équation (2) sous la forme :

$$y(i,k) = y(i,k-1) + w(i-1,k)\ x(i-1,k-1) \quad i > 0 \text{ et } k = 0,1,...K$$
$$w(i,k) = w(i-1,k) = w(k)$$
$$x(i,k) = x(i-1,k-1) \qquad \textbf{équation (3)}$$

Ce qui revient à faire circuler w(k) et x(i-k) de calcul en calcul. L'équation (3) s'écrit en lustre.

```
w array [0..K] of real
x,y array [0..K+1] of real
for k in [0..K+1]
  let
    y(k) = y(k-1)+x(h) pre(x(k-1))        modélisation (2)
    x(k) = pre(x(k-1))
  tel
x(0) = x
y(0) = 0
y = y(K+1)
```

L'équivalence entre les modélisations(1) et (2) étant immédiate (définition de variables supplémentaires et réécriture).

* **Cadencement des calculs et choix d'une architecture**

Les instants d'exécution des différents calculs sont spécifiés par une fonction temporelle t respectant la contrainte de dépendance entre les calculs.
En ce qui nous concerne la fonction t définie par $t(i,k) = i + k$ respecte cette contrainte. Il reste ensuite à définir une fonction d'allocation entre les calculs et les cellules les réalisant ; dans l'exemple choisi nous allons considérer K + 1 cellules réalisant les calculs au points (i,k) : $a(i,k) = k$
$k = 0,...K$

Le circuit final est donc une composition de cellules de base dont la structure est donnée ci-dessous (figure 8).

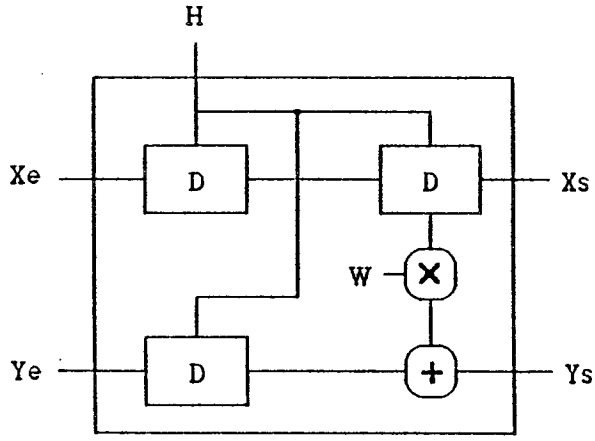


Figure 8 : Cellule de base

La description en LUSTRE du circuit final étant :

```
w array [0...K] of real
X array [0...K+1] of real
Y array [0...K+1] of real
node cellule (Xe,Ye,W : real) returns (Xs,Ys : real)
  let
    Ys = pre (Ye) + W pre(pre(Xe))
    Xs = pre (pre(Xe))
  tel
Y[0] = 0
X[0] = x
y = Y[K+1]
for J in [1...K]
  let
    X(J),Y(J) = CELLULE (X(J-1),Y(J-1),W(J-1))
  tel
```

figure 9 : description finale (modélisation 3)

*** <u>Vérification de cohérence</u>**

Pour assurer la conformité du circuit et donc de sa description donnée figure 8 avec les spécifications initiales, il nous faut prouver l'équivalence entre les modélisations (2) et (3).

Les équations de la modélisation (2) sont :

$$Y(k) = Y(k-1) + w(k)\ pre(X(k-1))$$
$$X(k) = pre\ (X(k-1))$$

On introduit ensuite le changement de variable :

$$Y'(k) = pre^k(Y(k))$$
$$X'(k) = pre^k(X(k))$$

On en déduit :

$$Y'(k) = pre^k(Y(k))$$
$$= pre^k(Y(k-1)+w(k)\ pre(X(k-1)))$$

$= pre(pre^{k-1}(Y(k-1))) + pre^{k}(w(k))\ pre(pre(pre^{k-1}(X(k-1))))$

$= pre\ Y'(k-1)+w(k)\ pre(pre(X(k-1)))$

Ce qui est bien l'équation de base de la modélisation (3).

iv) Remarques sur LUSTRE

* Le type d'approche développé dans ce langage est à l'opposé des approches classiques : au lieu de proposer une syntaxe complexe, ce qui peut être un moyen permettant de modéliser aisement tout type de circuit, la syntaxe de LUSTRE est réduite mais en contrepartie sa sémantique est parfaitement définie et permet d'utiliser la puissance des manipulations mathématiques sur des expressions.

* On peut établir un parallèle entre la notion d'invariant dans les langages de programmation et un programme lustre : un invariant dans un langage de programmation est une équation qui est vérifiée localement, un programme LUSTRE peut être considéré comme un ensemble d'invariants globaux. On peut aussi rapprocher ces notions de la notion d'assertion introduite dans le langage CADOC.LD.

* L'applicabilité de LUSTRE à la description de matériel n'est pas le but initial de ce langage initialement conçu pour la spécification et la validation de programme temps réel ; une des raisons classique de l'extension à la modélisation des circuits est qu'il est intéressant de pouvoir décrire l'environnement du programme temps réel : décrire à la fois le logiciel et le matériel. L'étude de l'application de LUSTRE à la modélisation de circuits étant récente, on ne peut conclure s'il est possible de l'utiliser pour des problèmes de circuiterie particulière (ce point sera développé dans la troisième partie) ou si la syntaxe réduite ne le restreint pas implicitement à des modélisations très abstraites.

II.3.3 - LTS

Le langage LTS (Layout and Timing for Structure) présenté dans [MIL 84] est basé sur les mêmes idées que le langage LUSTRE présenté précédemment (langage déclaratif, notion d'histoire d'une variable, sémantique formelle,...) ; pour cette raison, nous n'allons pas le décrire en détail mais simplement discuter certains points particuliers.

i) Modèle temporel

Le temps en LTS est un temps discret, (suite des valeurs 0,1,2...) en conséquence, une variable (ou signal) peut être considérée soit comme une fonction de l'ensemble des instants dans un ensemble de valeurs, soit comme une séquence de valeurs.
Ce temps discret est assimilable à une variable non échantillonnée et toujours vraie de LUSTRE.

ii) analogie LTS LUSTRE

Travaillant sur l'histoire des variables, LTS définit un certain nombre d'opérateurs sur ces histoires, ces opérateurs ont soit un équivalent immédiat en LUSTRE, soit peuvent être exprimés comme composition d'opérateurs LUSTRE.

* Opérateur last

L'opérateur last appliqué à un signal x définit un signal dont la valeur à tout instant est la valeur de x à l'instant précédent :

x	T	F	F	T	F
last(x)	T	T	F	F	T	F

instant initial

Si l'on reprend les notations utilisées dans le paragraphe récédent on a :

si x → e1,e2,...,en,...
alors last(x) → e1,e1,e2,...,en,...

à rapprocher de pre(x) → nil,e1,e2,...,en,... (LUSTRE).

Remarquons que la valeur intiale de last(x) est la valeur intiale de x ce qui parait peu justifiable et rend difficile l'initialisation des signaux qui est faite en utilisant une fonction de manipulation explicite du temps permettant de désigner les instants absolus.

Exemple : La description d'une horloge de période 2 sera :

```
clock =
   if time mod 2 is
     0 then true
     1 then false
end
```

qui est à rapprocher de la définition LUSTRE suivante :

```
clock = T → not(pre(clock)).
```

*** opérateur atmostrecent**

Cette opérateur est définit par

```
atmostrecent (x,y) =
     if x is
       true then y
       false then atmostrecent (last(x), last(y))
     end
```

et retourne la valeur de y au temps le plus récent auquel la valeur de x était vraie ; on a l'équivalence :

```
atmostrecent (x,y) = current (when(x)(y)).
```

Il faut noter à ce niveau que LTS utilise fortement la notion de récursion (inconnue en LUSTRE) pour définir de nouvelles fonctions : un autre exemple étant la définition d'un délai de n unités de temps :

```
delay(n)(x) =
   if n is
     0 then x
     as m:1..? then delay(m-1)last(x)
end
```

L'inconvénient de la récursion étant qu'elle est synonyme d'appels imbriqués et donc de temps d'évaluation important ; cette remarque n'étant pas à prendre en compte au niveau des caractéristiques du langage mais est importante si ce langage vise les applications temps réel.

Pour terminer cette présentation rapide il faut noter en LTS :

- La notion de groupe de définitions (équivalent à un noeud LUSTRE ou un paquetage ADA)
- Les notions de typage automatique ou manuel, de typage d'une fonction
- La possibilité de définir des fonctions complexes (de la forme map(f) (x,y) (n), définie comme f(x(n),y(n))), possibilité classique pour les langages fonctionnels et de lisibilité parfois discutable.

iii) Modélisation de dispositifs particuliers

[MIL 84] présente la manière de décrire en LTS un certain nombre de dispositifs propres au circuit et en particulier l'article s'intéresse à la modélisation des liaisons (connexions) et des portes de transfert bidirectionnelles (sur ce sujet, on se reportera à la troisième partie de cet exposé).

* Connexions

La modélisation d'un lien bidirectionnel entre 2 signaux est faite par l'intermédiaire d'un opérateur spécial noté :› et prononcé "dépend de".
La définition formelle de cet opérateur semble floue et n'est pas donnée dans [MIL 84] ou il est simplement indiqué que :› est assimilable au := des langages conventionnels.

Exemple
Une liaison bidirectionnelle entre 2 signaux x et y est définie par :

```
biwire (x,y) = begin x :› y, y :› x end
```

Cette définition imprécise semble contradictoire avec le caractère fonctionnel (déclaratif) initial de LTS car elle introduit la notion d'affectation, donc l'existence d'effets de bord (on peut rapprocher le :› de LTS au SET de Lisp).

* Porte de transfert

En admettant l'opérateur :› indiqué précédemment la description d'une porte de transfert (transistor) est la suivante :

```
transistor (gate) (source, drain) =
   begin
     (source,drain) :*
         if gate is
           true then (drain, source)
           false then (source, drain)
     end
   end
avec l'équivalence (x,y) :* (z,t) ↔ x :* z, y :* t.
```

II.3.4 - REMARQUES SUR LES LANGAGES FONCTIONNELS ET DECLARATIFS

Les définitions proposées pour les expressions "langage fonctionnel" et "langage déclaratif" ne sont pas parfaitement rigoureuses, cela est justifiable par le fait que nous ne nous sommes pas intéressés à ces langages en tant que tels mais à leur application à la description de dispositifs matériels.

La description de circuit n'était pas a priori le but initial des langages évoqués dans cette partie, leur orientation initiale étant plutôt la validation ou la preuve. Il est donc évident qu'ils ne peuvent être comparés à des langages conçus spécifiquement pour la modélisation de circuits ; si une telle comparaison est hors de propos on peut quand même noter que dans le cas de LTS, l'application du langage à la description de circuits a entrainé l'introduction d'opérateurs en contradiction avec la philosophie à la base du langage, ce qui semble confimer la remarque selon laquelle la modélisation "précise" et la vérification formelle sont encore des domaines en conflit.

Nous conluerons cette partie en soulignant que les frontières entre langages fonctionnels, parallèles et autres ne sont pas parfaitement claires et qu'il existe des approches mixtes comme FP2 (functional Parallel Programming) [CIS 84] utilisant les caractéristiques de plusieurs classes de langages.

II.4 - LANGAGES DE L'INTELLIGENCE ARTIFICIELLE

Nous aurions pu évoquer les deux langages dont nous allons parler (LISP et PROLOG) dans les chapitres sur les langages fonctionnels et logiques respectivement ; pour des raisons d'habitude nous regroupons ces deux langages dans une rubrique commune des langages de l'intelligence artificielle. Une autre raison pour ce classement étant que nous n'allons pas présenter les principes de la programmation fonctionnelle (voir paragraphe précédent) ni de la programmation logique mais uniquement leur application à la description du matériel.

II.4.1 - PROLOG

En ce qui concerne PROLOG, un exemple d'application est donné dans [SUZ 85], le dialecte utilisé étant une version parallèle de Prolog offrant la possibilité de définir des processus parallèles communiquant par l'intermédiaire de variables partagées ; chaque circuit étant représenté par un prédicat (expression en concurrent PROLOG).

Exemple

La description du circuit générateur de code correcteur d'erreur de la figure 9a et donné figure 9b.

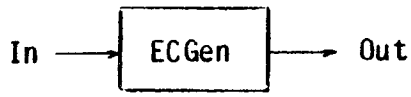


Figure 9a : générateur de code correcteur d'erreur

```
ECGen ([In/InQueue],[Out/OutQueue])
 :- generate (In,Out),                    (1)
    ECGen (InQueue?,OutQueue)             (2)
ECGen ([],[])                             (3)
```

Figure 9b : description PROLOG

La notation [X/Y] représentant une liste constituée des valeurs successives sur un port donné. Le symbole ":-" introduit la définition du comportement du circuit ECGen : en réponse à l'entrée [In/InQueue], on génère d'abord (1) le premier élément de la sortie (Out) à partir du premier élément de la liste d'entrée (In). La description est ensuite rappelée récursivement sur la suite des listes (2) ; le processus s'arrêtant lorsque la liste des entrées est vide (3).

Les manipulations explicites du temps se font par l'intermédiaire d'un processus générant une horloge globale produisant la séquence infinie d'entiers [0,1,2,3,...].

Les exemples donnés dans [SUZ 85], seule référence disponible, sont uniquement sur des circuits de complexité élevée (générateur de code correcteur d'erreur, mémoire principale,...) avec un niveau d'abstraction élevée. Les descriptions sont difficiles à manipuler dans la mesure ou le concepteur ne maitrise pas une syntaxe PROLOG-équivalente ; cette complexité est encore augmentée lorsque l'on souhaite manipuler explicitement le temps.

Une des raisons pouvant justifier l'étude entreprise étant le fait que concurrent PROLOG est le langage central du projet d'ordinateurs de la cinquième génération ; son utilisation en tant que CHDL donnerait alors accès à un environnement de développement considérable et serait en théorie accessible à une importante communauté d'utilisateurs. Cet argument a été dans le passé avancé pour le langage ADLIB (dont est dérivé HELIX, cf. II.1) basé sur Pascal et est avancé actuellement pour le langage ADA.

Pour conclure, notons que l'intérêt principal d'utiliser PROLOG (ou tout autre langage de la même famille) est l'orientation en vue de la vérification de circuits : la méthodologie employée se situe entre la simulation fonctionnelle et la preuve formelle ; elle utilise le mécanisme de réfutation d'énoncés pour vérifier ou infirmer une assertion donnée à partir d'expressions de base par exploitation des mécanismes de déduction propres à l'environnement PROLOG.

II.4.2 - LISP

Le langage LISP a été choisi par le MIT comme support de ses recherches sur la compilation de Silicium ([SOU 83], [SIS 82]) ; de manière simplifiée une description (ou programme) MacPitts est constituée d'un ensemble de processus exécutables en parallèle, chaque processus étant divisé en états exécutés séquentiellement. Les opérations exécutées dans un état étant spécifiées par une forme LISP.
La sémantique associée au langage est fortement orientée par l'architecture cible de la synthèse (partie contrôle/partie opérative).

Exemple

Les registres réalisant les points de mémorisation sont du type Maitre/esclave ; ils sont lus avant d'être écrits ce qui autorise l'écriture suivante d'un échange entre deux registres (le symbole "par" indiquant une évaluation parallèle) :

```
(par (setq (a,b)
     (setq (b,a))
```

La modélisation proposée est fortement synchrone : toutes les évolutions se font en référence aux cycles d'une horloge de base. Le système proposé est orienté synthèse, le langage de modélisation est en conséquence assez pauvre et devient insuffisant dès que l'on sort du modèle partie opérative/partie contrôle, en particuliers pour des modélisations en vue de simulation.

Remarques

- L'utilisation du "set" ôte au LISP utilisé son caractère de langage fonctionnel.

- On peut noter dans [BAT 81], l'utilisation de LISP en tant que langage de description au niveau layout, la sémantique associée étant une sémantique de synthèse.

III - LANGAGES SPECIFIQUES A LA DESCRIPTION DU MATERIEL

Nous avons vu dans le chapitre précédent que l'adaptation de langages non spécifiques à la description de circuit présente certes des avantages (compilateurs déjà écrits, environnements importants, ...), mais aussi des inconvénients en particulier sur les points suivants qui sont propres aux systèmes logiques : modélisation du parallélisme et manipulation explicite de la dimension temporelle.

Après le précurseur CDL [CHU 65], les premiers langages spécifiquement conçus pour la description du matériel sont apparus vers les années 74 : DDL [DIE 74], AHPL [HIL 74], ISP [SIE 74] ; à la différence de ces langages qui étaient associés à un niveau de description donné, les langages proposés actuellement sont en général multiniveaux.
Dans ce paragraphe, nous allons nous intéresser à cinq approches différentes (cinq langages) ; le choix de ces langages parmi ceux existants dans la littérature a été fait sur des critères plus ou moins arbitraires : HILO MARK II illustre le mécanisme d'extension utilisé pour passer d'un langage mononiveau existant à un langage multiniveau, CAP/DSDL a été choisi pour ses ressemblances avec CADOC.LD, MODLAN introduit la notion d'assertion, VHDL est une illustration de l'application des concepts d'un langage non spécifique (ADA) à la définition d'un langage spécifique. CONLAN propose une approche totalement différente qui sera présentée en dernière partie.
Pour ces cinq approches nous donnerons un aperçu rapide basé sur des exemples du langage considéré et nous ferons un certain nombre de remarques.

III.1 - HILO MARK II

III.1.1 - GENERALITES

Les concepteurs de ce langage proposent une nouvelle version multiniveau par extension d'une version antérieure qui est mono-niveau : HILO logique.

Un circuit peut être décrit en HILO II comme une composition hiérarchisée d'éléments qui sont des objets prédéfinis et/ou des fonctions. Les communications entre les unités sont décrites par des variables de connections et/ou par l'intermédiaire d'événements.

Il existe deux types de description :

- description structurelle
- description fonctionnelle

III.1.2 - DESCRIPTION STRUCTURELLE

La description structurelle consiste à décrire le circuit comme une interconnection d'éléments prédéfinis (portes logiques, portes trois-états, ...), de circuits ou de fonctions, l'interconnection étant faite par l'intermédiaire de variables ; l'ensemble des valeurs de ces variables constituant l'état du circuit. Cette méthode est en général utilisée pour les descriptions structurelles au niveau logique.

III.1.3 - DESCRIPTION FONCTIONNELLE

La description fonctionnelle d'un circuit en HILO II utilise des objets plus complexes qui sont :

- les événements déclenchant le calcul des nouveaux états des variables.
- les registres activés par des opérateurs prédéfinis .

Exemple

R := B loadif1 A

L'opérateur loadif1 permet d'affecter au registre R la valeur de B si A = 1.

Avec la sémantique de ces opérateurs, on ne modélise que les registres sensibles au niveau. Pour les registres sensibles au front, il faut utiliser une autre construction qui est l'instruction à événement dont la syntaxe est :

when événement do listes_actions .

Les actions de listes_actions seront exécutées lorsque la partie événement sera satisfaite

i) actions

Les actions peuvent être :

- des actions conditionnelles (if-then-else)
- des transferts entre registres
- des créations d'événements.

ii) Manipulation du temps

La partie événement d' une instruction spécifie soit un changement de la valeur d'une variable, soit une temporisation, soit une séquence d'événements.

Exemple 1

Le comportement d'un registre sensible au front montant est défini par :

when CLK (0 to 1) do Q := D ;

exemple 2

when 12 * (CKA then CKB) then (EVA or EVB) wait 5 do A[0:7] := B[0:7] ;

Cet exemple illustre la manipulation du temps dans le langage, l'exécution d'une action peut être temporisée en utilisant des séquences d'événements et

de retards (wait). Pour l'exemple donné ci-dessus, le transfert de registre entre A et B sera exécuté après 12 séquences des événements CKA puis CKB, suivies de l'occurrence de l'événement EVA ou de l'événement EVB et après une attente de 5 unités de temps.
Notons que EVA, EVB, CKA, CKB, sont des événements créés explicitement par une ou plusieurs instructions d'une description et que déroulement du protocole d'exécution de l'action peut être interrompu en spécifiant une clause RESET.

III.1.4 - REMARQUES

- Le mode de spécification des protocoles d'accès à une action est difficilement exploitable : la définition d'un modèle graphique aurait été préférable pour des raisons de lisibilité ; cette remarque est confirmée par l'utilisation dans [FLA 81] d'un équivalent graphique pour expliquer le comportement d'un protocole exprimé sous forme textuelle.

- La temporisation est utilisée pour spécifier l'instant d'exécution de l'action mais il est impossible de temporiser l'action elle-même. La notion simple d'affectation de chronogrammes n'est pas définie dans le langage.

- Pour pouvoir faire une simulation efficace au point de vue "temps de simulation", un ensemble d'événements peut être associé à chaque élément du circuit. Ces événements sont rattachés aux variables de connexions ; L'évaluation des nouveaux états de ces variables ne sera faite que sur occurrence de ces événements. Cette démarche est semblable à celle utilisée dans ADLIB au travers des opérateurs SENSITIZE et DESENSITIZE.

- HILO II est un exemple caractéristique des langages ayant évolué depuis une version de base mononiveau RTL par introduction de primitives fonctionnelles ; le résultat étant un langage "logico-fonctionnel" à définition assez floue.

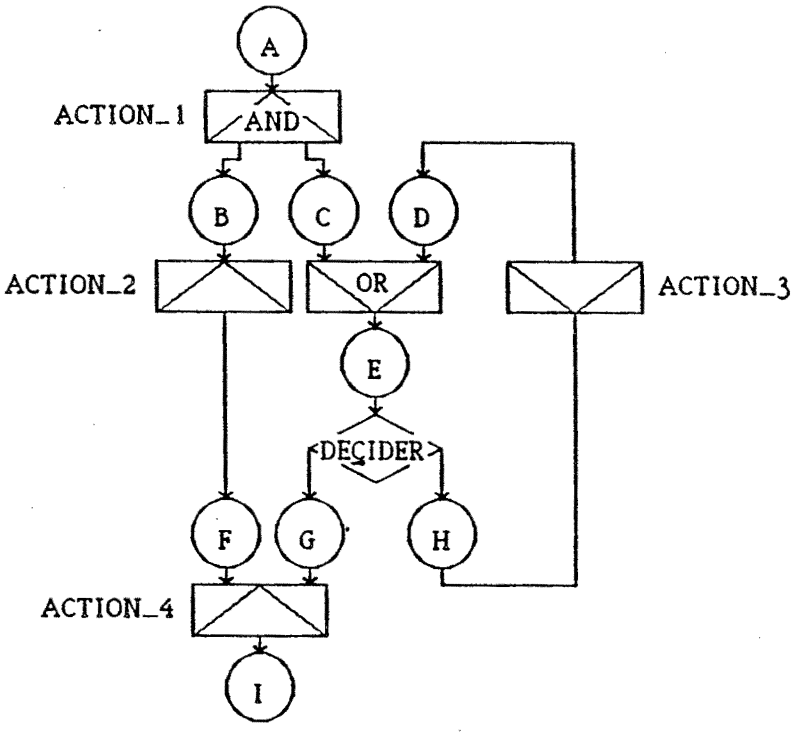
III.2 -CAP/DSDL

III.2.1 - GENERALITES

Une des caractéristiques de ce langage est qu'il est basé sur un modèle de type Réseau de Pétri temporisé et interprété. Ce modèle est utilisé pour la description fonctionnelle d'un circuit ou d'une partie de circuit.

Les transitions sont de trois types : AND, OR, DECIDER. Les actions sont associées aux transitions (comme nous le verrons ultérieurement, ce principe est différent de celui choisi pour CADOC.LD dans lequel les actions sont associées aux places ; l'équivalence entre les modèles se ramenant à l'équivalence des modèles de Mealy et de Moore en théorie des automates.

Exemple



Le format textuel de ce graphe est le suivant :

```
var A, B, C, D, E, F, G, H, I : place
net
  on (A) do mark (B & C) ACTION_1 ;
  on (B) do mark (F) ACTION_2 ;
  on (C  D) do mark (E) ;
  on (E) do if CO then mark (H)
                  else mark (G) ;
  on (H) do mark (D) ACTION_3 ;
  on (F & G) do mark (I) ACTION_4 ;
end
```

figure 10 : représentation textuelle CAP/DSDL

En parallèle du modèle graphique, CAP/DSDL offre aussi aux utilisateurs des constructions algorithmiques structurées qui sont :

- seqbegin ... end : les instructions comprises dans le bloc seront exécutées séquentiellement.
- cobegin ... end : les instructions comprises dans le bloc seront exécutées parallèlement. L'exécution du bloc sera considérée comme terminée si toutes ses instructions le sont.

Exemple

Le fonctionnement du circuit décrit par le graphe de l'exemple précédent peut être décrit en utilisant ces constructions structurées par :

```
seqbegin
   ACTION_1 ;
   cobegin
     ACTION_2 ;
     while CO do ACTION_3 ;
   end
   ACTION_4 ;
end ;
```

Remarque

Ces constructions structurées n'ont pas la puissance du réseau de Pétri. Mais elles permettent d'assurer certaines propriétés de fonctionnement : réseau sauf, sans blocage, réinitialisable,...

III.2.2 - VARIABLE ET TYPES DE DONNEES

Deux classes de variables sont définies en CAP/DSDL :

- variables "explicites"
- variables "implicites".

A une variable "explicite" (à mémorisation) est explicitement associée un signal qui active leur fonction de mémorisation.
Une variables "implicite" (sans mémorisation) prend une nouvelle valeur à chaque fois qu'une variable figurant en partie droite de l'expression la définissant change de valeur ; ces expressions sont données dans une section non procédurale de la description.

Remarque

En d'autres termes les variables explicites font référence aux éléments matériel qui sont les bascules, les registres, les mémoires, ...Cette approche est celle rencontrée dans les langages RTL.
D'autre part, les variables implicites décrivent des connexions (ou les terminaux dans les langages RTL) et des parties combinatoires du circuit. Un temps de réaction peut être associé à ces variables.

Exemple 1 : bascule RS sur un bit :

```
var R,S,N,Q : implicit bit ;
impdef
   Q := R nor NQ (up 5 to 7, down 4 to 6) ;
   NQ := S nor Q (up 6 to 8, down 3 to 6) ;
```

Exemple 2 : registre RS de 16 bits :

```
var R, S, Q, NQ : implicit bit (16) ;
impdef
   Q := R nor NQ ;
   NQ := S nor Q ;
```

Le type de données de base du langage est le "bit (0,1)", étendu aux chaînes de bits. Dans l'exemple 2, les variables R, S, Q et NQ sont des nappes de 16 fils.
La manipulation de données au niveau "bit" est intéressante pour les expressions booléennes (les opérations booléennes classiques étant étendues au cas des chaines de bits), par contre, elle est fastidieuse pour les manipulations d'expressions arithmétiques.
Les autres types de données existants sont des types structurés : tableaux, enregistrements.

III.2.3 - MANIPULATION DU TEMPS

Pour la manipulation du temps, le langage offre des primitives tels les "retards" (DELAY) et les objets de type "événement".
Un retard définit la période entre l'évaluation de l'expression donnant la valeur à affecter à une variable et l'affectation effective.
Les événements peuvent être le niveau d'un signal ou le changement de valeur d'une variable [RAM 83a]. Ces événements sont utilisés dans les deux constructions suivantes :

```
at signal do action
when signal do action
```

La première construction permet de spécifier une action déclenchée sur l'occurrence d'un changement de valeur sur la variable ‹signal›.
La deuxième permet de spécifier une action qui est exécutée pendant le niveau "VRAI" de la variable ‹signal›.

III.2.4 - REMARQUES

- Nous reviendrons sur l'application de CAP/DSDL à la modélisation des dispositifs MOS (partie 3, IV.1), remarquons simplement ici que ces modélisations se font par introduction d'objets à signification matérielle prédéfinis (PULLUP, PULLDOWN, TRANSFER) ce qui est en contradiction avec le caractère jusqu'alors fonctionnel du langage ; le modèle des Réseaux de Pétri (RdP) possède une sémantique suffisante pour permettre la description de tout type de circuits et donc de ne pas avoir à prédéfinir un certain nombre d'objets (ceci sera illustré sur CADOC.LD dans la partie 3, paragraphe IV.2).

- En ce qui concerne la manipulation du temps, CAP/DSDL ne permet pas la modélisation de chronogrammes, les primitives ne permettant de spécifier uniquement un couple (valeur affectée, date d'affectation).

- Plus important à noter est la confusion entre condition booléenne et événement : si l'on considére une instruction du type
at (state = "00" & ck) do ....
on a fusion entre une condition au sens classique (state = "00") et l'échantillonnage de cette condition (ck).

- Les types de données manipulables (bits et chaines de bits), adaptés à la modélisation niveau RTL sont peu compatibles avec des niveaux d'abstraction plus élevés nécessitant des types de base plus complexes (entier, types énumérés, ...).

III.3 - MODLAN

III.3.1 - GENERALITES

Les notions de base définies dans le langage MODLAN ([PAW 82]) sont :

- module : une partie d'un système logique
- terminal : un point de communication du module avec l'extérieur.
- interconnexion : une voie de transfert d'informations entre les terminaux.

Un module est décrit dans un des trois types de description suivants :

- SM (description structurelle)
- FM (description fonctionnelle)
- CM (description avec contrôle explicite)

De plus, le langage possède des modules primitifs (PM) : portes logiques, retards, buffers trois-états, bus bidirectionnels, ...

Un terminal peut recevoir un des deux attributs suivants :

- OC : collecteur ouvert
- TSL : terminal trois-états.

Remarque :

Le mécanisme d'attributs est intéressant et permet de définir des informations supplémentaires associées aux objets déclarés en vue de traitements particuliers ; par contre, le fait d'avoir prédéfini ces attributs - correspondants à des technologies particulières - dans le langage est discutable : il aurait été préférable de laisser le choix de leur définition à l'utilisateur.

III.1.2 - DESCRIPTION STRUCTURELLE (SM)

Dans ce cas, le module est décrit comme une interconnexion de sous-modules primitifs (PM) ou d'autres sous-modules décrits en SM, FM ou CM. La récursivité dans la description de type SM permet la hiérarchisation de la description d'un modèle.

* **valeurs manipulées**

Les 7 valeurs manipulées dans ce type de description sont :

- 0 : niveau bas
- 1 : niveau haut
- R : transition de 0 à 1 ou front montant
- F : transition de 1 à 0 ou front descendant
- ? : erreur de potentiel ou état indéterminé à l'initialisation
- "?" : haute impédance
- V : conflit sur un bus bidirectionnel.

III.3.3 - DESCRIPTION FONCTIONNELLE (FM)

Ce type de description est utilisé pour définir un module par ses fonctions. Le fonctionnement du module est décrit en utilisant un ensemble d'instructions manipulant des événements (langage type RTL). Ces instructions se trouvent soit dans une partie procédurale de la description, soit dans une partie non-procédurale.

*** valeurs manipulées**

Les valeurs des variables dans une description FM sont : 0, 1, ?.

*** variables et types de données**

Les variables doivent toutes être déclarées et typées. Le type de base est le "bit"(ou booléen). D'autres types sont définis et sont paramétrés par la taille (nombre de bits) : registres, mémoires, ...

Remarque

La déclaration de variables "matérielles" et l'utilisation de blocs non procéduraux apparentent le langage MODLAN aux langages RTL.

*** manipulation du temps**

Au point de vue manipulation du temps, le langage offre un module primitif "DELAY" avec :

- soit deux paramètres de retards nominaux (d_{LHnom}, d_{HLnom})

- soit quatre paramètres (d_{LHmin}, d_{LHmax}, d_{HLmin}, d_{HLmax}) pour la simulation de pire cas.

L'utilisateur peut aussi spécifier des contraintes temporelles par des déclarations de type CHECKS. Ces déclarations concernent la stabilité des entrées pendant un intervalle de temps donné.
Dans l'exemple de la figure 11, on veut exprimer les conditions de stabilité du signal SDI. TTDS est "vrai" si le signal a été bien positionné pendant le temps TDS, TTDH est "vrai" si le signal SDI a été maintenu pendant une période TDH à partir du front montant de HLDA.

En ce qui concerne la temporisation des actions, en particulier celle des affectations, on peut associer aux variables receptrices une valeur de retard. Dans l'exemple ci-dessous, la variable INTA prend la valeur ? après un retard de THD unités de temps.

La combinaison de la déclaration de stabilité et de l'instruction d'affectation correspond aux chronogrammes de la figure 11.

```
checks TTDS = SDI/TDS                    %temps de positionnement de SDI%
       TTDH = HLDA↑.SDI/TDH              %temps de maintien%

    if (HLDA↑ & TTDS) then INTA/THD/ ← ?
```

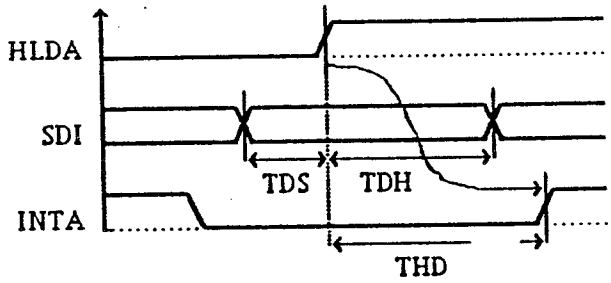
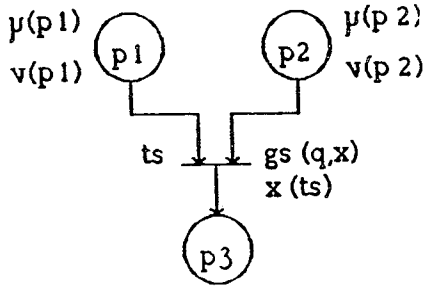


Figure 11 : assertions MODLAN

III.3.4 - DESCRIPTION A CONTROLE EXPLICITE (CM)

Le modèle utilisé dans les descriptions à contrôle explicite est un graphe de type Réseaux de Pétri synchronisés.



$\mu(p)$: actions associées à p
$\nu(p)$: temps pendant lequel p reste active
$x(t)$: temps de transition
$g_s(q,x)$: condition de transition
x = ensemble des variables d'entrées
q = ensemble de variables logiques associées aux places.

* **actions**

Les actions associées aux places sont décrites en utilisant les constructions définies pourle type FM. L'activation de la place p_i correspond à l'exécution de l'action $\mu(p_i)$.

* **transitions**

Le graphe défini explicitement le contrôle de l'exécution des actions : le franchissement des transitions engendre l'activation des places en aval.

III.3.5 - REMARQUES

- Pour couvrir tous les niveaux de description de circuit, MODLAN utilise une approche multitype de description : une description de type SM pour décrire le circuit au niveau porte logique architectural, une description de type FM ou CM pour décrire le circuit au niveau RTL. Nous avons déjà remarqué avec CAP/DSDL qu'un modèle de type RdP est suffisamment puissant pour décrire tous les modèles de circuits. L'addition des mécanismes d'interconnexion et d'hiérarchisation permettant la descripton de la structure et du fonctionnement de tout type de circuit, la nécéssité de 4 modèles est en conséquence non justifiée.

- Il faut noter la définition dans le langage d'attributs permettant de modéliser des dispositifs propres à une technologie (collecteur ouvert, logique trois-états). La technique utilisée (définition de mots clef) rendant le langage dépendant des technologies traitées, sans possibilité de nouvelles définitions par l' utilisateur;

III.4 - VHDL

III.4.1 - GENERALITES

Le langage VHDL (VHSIC Hardware Description Language, [SHA 85]) a été défini comme le langage de spécification de matériel pour le support de tous les outils du projet VHSIC (Very High Speed Integrated Circuits).

Les principaux points que ce langage doit satisfaire sont les suivants :

- utilisation des concepts du langage ADA (mécanisme de compilation séparée),
- support pour les outils de conception, de documentation et de simulation à différents niveaux,
- indépendance par rapport à l'évolution de la technologie et la méthodologie de conception, tout en laissant aux utilisateurs la possibilté de prendre en compte d'autres informations dans leurs descriptions (technologie utilisée, fan-in/fan-out,...), informations qui seront utilisées pour des traitements spécifiques (cette méthode est trés semblable à celle proposée dans CADOC.LD (partie 3, IV.2.2).

La notion de base du langage est "l'entité de conception". Elle représente un circuit ou une partie de circuit. Une entité de conception est composée :

- d'une interface
- d'un ou plusieurs corps.

III.4.2 - INTERFACE

L'interface d'une entité de conception décrit tous les ports d'entrées/sorties et les paramètres génériques de la description. Ces informations sont visibles (accessibles) à l'extérieur de l'entité. L'interface comporte une partie invisible de l'extérieur de l'entité mais connue par tous les corps de l'entité. Cette partie est constituée d'un

ensemble de déclarations : les types de données, les définitions d'attributs, les assertions.

i) Ports d'entrée/sortie

Les ports d'entrées/sorties sont tous déclarés et typés et appartiennent à l'une des trois classes suivantes :

- entrée
- sortie
- bidirectionnel

ii) Paramètres génériques

Les paramètres génériques sont utilisés pour définir une classe de composants : c'est-à-dire un modèle. Ces paramètres peuvent être liés à une technologie, à des caractéristiques temporelles, à des dimensions, ou à des attributs de type fan-in/fan-out. La valeur de ces paramètres doit être fixée lorsque l'entité est instantiée.

Exemple

L'additionneur 4-bits représenté figure 12 a comme interface :

```
with ADDER_RESOURCE ;
  -- spécification de l'interface
entity FOUR_BIT_ADDER
     (A, B : in BIT_VECTOR (3..0); -- ports
     cin : in BIT
     cout : out BIT
     sum : out BIT_VECTOR (3..0))
generic (DELAY : TIME := 36ns) -- paramètre et valeur par défaut
assertion DELAY > 3ns
          SUM'FANOUT < MAX_FANOUT
end FOUR_BIT_ADDER
```

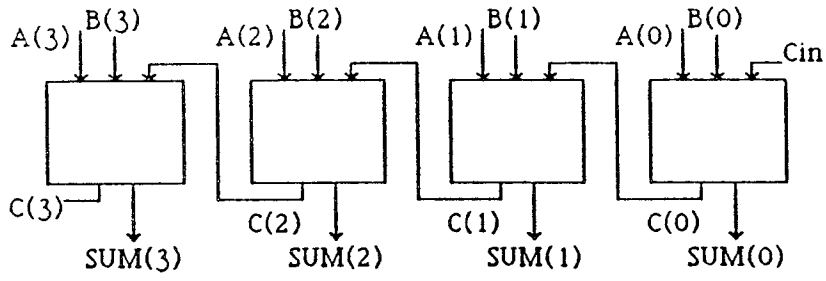


Figure 12 : additionneur 4 bits

Le paramètre générique "DELAY" de type TIME etant fixé à une valeur de 36 ns, l'entité est donc unique.

iii) Types de données

Les types de données définis dans VHDL sont :

- les types prédéfinis : entier, réel, booléen, vecteurs de bits
- les types composés définis par l'utilisateur : énumération, tableau, enregistrement.
- les types physiques : longueur, temps, voltage, ... ; à ces types es sont associés des unités de unités de mesure : dans l'exemple précédent le paramètre DELAY est de type "TIME" dont la définition est donnée figure 13.

```
type TIME is range 0..1E20
     units
     fs ;                  - femtosecond
     ps = 1000fs ;         - picosecond
     ns = 1000ps ;         - nanosecond
     µs = 1000ns ;         - microsecond
     ms = 1000µs ;         - millisecond
     s  = 1000ms ;         - second
```

Figure 13 : définition d'un type physique VHDL

Remarque

L'utilisateur peut définir par énumération ses propres valeurs logiques ainsi que les opérations de manipulation de ces valeurs. cela permet par exemple de définir un type logique étendu ({0, 1, Z, U, X} par exemple) plus adapté à la modélisations de dispositifs matériels.

iv) Définition d'attributs

Au lieu de prédéfinir des attributs d'objets dans le langage, VHDL possède des mécanismes permettant aux utilisateurs de les définir. Ils pourront ainsi ajouter dans leur description des informations qui seront exploitées par des outils spécifiques. Les attributs sont typés en vue d'une vérification. Ils sont aussi rattachés à une classe de ports d'entrée/sortie. Par exemple l'attribut "fan-out" est rattaché à tous les signaux de sortie.

Remarque

Le rattachement des attributs aux classes d'E/S est limitative : la capacité d'entrée n'est pas identique pour tous les ports, de même pour les sortances.

v) Assertions

Les assertions permettent la spécification des conditions d'utilisation d'une entité qui doivent être satisfaites pendant la conception du circuit

(cf. assertions statiques de CADOC.LD). Elles peuvent être aussi utilisées pour exprimer certains résultats de simulation attendus pour mieux localiser les erreurs (cf. assertions dynamiques de CADOC.LD).

Exemple

```
(SUM'FANOUT < MAX_FANOUT)
report "SUM a trop de charge à piloter"
```

Si pendant la simulation l'attribut "fanout" de SUM a une valeur plus grande que "MAX_FANOUT", l'erreur sera signalée.

III.4.3 - CORPS D'UNE ENTITE DE CONCEPTION

Le corps d'une entité est une unité de compilation au sens ADA. Une entité peut être décrite soit comportementalement, soit structurellement. Il existe deux types de corps :

- "behavioral body" (spécification comportementale)
- "architectural body" (spécification architecturale)

i) Modèle temporel

Le modèle temporel de VHDL est classique (cf. III.5) ; il existe deux types d'instants :

- micro-instants,
- macro-instants.

Les macro-instants correspondent à des instants réels mesurables, par contre les micro-instants sont des instants non mesurables et inaccessibles à l'utilisateur.

ii) Spécification comportementale

Dans une spécification comportementale toutes les variables utilisées sont déclarées et typées. Elles n'ont pas de relation avec le temps : elles ne peuvent prendre qu'une seule valeur et leurs précédentes valeurs sont détruites. Il existe des variables statiques et dynamiques. Les variables statiques mémorisent leur valeur d'un changement à un autre, par contre celles qui sont dynamiques doivent être réinitialisées à chaque instant.
Les instructions sont exécutées séquentiellement.

Exemple

```
behavioral  body ADDER4B of FOUR_BIT_ADDER is
  variable t, ta, tb : HEXSUM
  begin
     ta := int (A) ;
     tb := int (B) ;
     t := ta + tb ;
     cout & SUM ← bin (t) after DELAY ;
  end ADDER4B ;
```

iii) Spécification architecturale

Dans une spécification architecturale (ou structurelle) toutes les entités utilisées doivent être déclarées et sont interconnectées par des "signaux".

Un signal est un objet qui peut contenir une suite de valeurs : les valeurs dans le passé étant accessibles mais non modifiables, et les valeurs dans le futur étant uniquement modifiables. Un signal peut prendre une séquence de valeurs avec leur date respective (chronogramme).

Les instructions définies dans une spécification structurelle sont exécutées parallèlement.

Exemple

```
architectural body PURE_STRUCTURE of FOUR_BIT_ADDER is
  signal C: BIT_VECTOR (3..0)

  component FULL_ADDER (CIN, I1, I2 : in BIT ;
                        COUT, RES : out BIT );

  begin
     for i in 3..0 generate
          if i = 0  generate
               FULL_ADDER (CIN, A(i), B(i), C(i), SUM((i));
          endgenerate

          if i ≠ 0 generate
               FULL_ADDER (C(i-1), A(i), B(i), C(i), SUM(i));
          endgenerate
     endgenerate
  COUT ≤ C(3)
end PURE_STRUCTURE ;
```

III.3.4 - REMARQUES

Ce langage possède des caractéristiques intéressantes : description comportementale, description structurelle, spécification d'attributs et d'assertions et manipulation de chronogrammes.
Contrairement aux langages précédents, VHDL ne possède aucun type prédéfini ce qui correspond à l'approche choisie pour CADOC.LD. Notons que la référence au langage ADA n'est pas nécessaire : les notions de visibilité et de modularité étant parfaitement classiques.

III.5 - Approche CONLAN

III.5.1 - GENERALITES

Le projet CONLAN ([BOR 81], [PIL 83]) a les objectifs suivants :

- possiblité de description d'un système à plusieurs niveaux d'abstraction (comportement et structure), de vérification de cohérence entre les descriptions.
- extensiblité de l'ensemble CONLAN
- définition très précise des résultats attendus dans l'interprétation des primitives de description (simulation).

L'approche CONLAN consiste à définir un langage de base à partir duquel on peut dériver des familles de langages ; chaque famille sera particularisée pour un niveau de description. Un ensemble de règles de dérivation a été défini pour permettre ces extensions. D'une manière générale, la définition d'un nouveau langage est faite en définissant les objets de base du langage à partir d'un langage de référence. Toutes les familles de langages sont dérivées à partir d'un langage de base appelé BASE CONLAN (BCL), lui-même défini à partir d'un langage plus élémentaire : PRIMITIF CONLAN (PCL).

CONLAN s'adresse à trois types d'utilisateurs :

- les concepteurs de langage d'application
- les concepteurs de logiciels basés sur un langage de l'ensemble CONLAN (simulateur, outil de synthèse, outil d'analyse, ...)
- concepteurs de systèmes logiques qui sont des utilisateurs d'un langage de CONLAN.

III.5.2 - MODULARITE

Le concept de modularité dans CONLAN est basé sur la notion de segments. Il existe six types de segments :

- TYPE : définition d'un nouveau type d'objet.
- CLASSE : définition d'un ensemble de types.
- FUNCTION : définition d'une opération qui renvoie un résultat.
- ACTIVITY : définition d'une opération qui modifie un ou plusieurs objets passés en paramètres.
- DESCRIPTION : description d'un système logique ou d'une partie d'un système logique
- CONLAN : définition d'un nouveau langage ou/et d'une bibliothèque.

La syntaxe d'un segment est unifiée pour tous les segments.

```
[REFLAN <identificateur_langage_référence> ]

  <type_de_segment> <identificateur> [( <paramètre> )] /en-tête/

    [ASSERT <assertions> ENDASSERT ]

    BODY
      [définition de segments locaux]
      [déclaration d'objets internes]
      [invocations d'opérations]
      [modification de syntaxe]
  END <identificateur>
```

III.5.3 - LANGAGE BCL (Base CONLAN Language)

L'ensemble CONLAN est défini sur un univers d'objets primitifs qui sont divisés en deux classes :

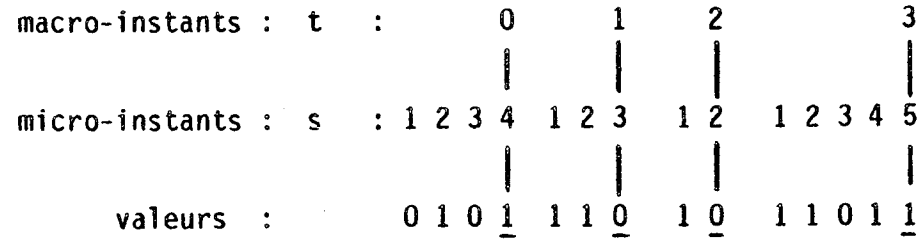
- les valeurs constantes, entières, chaînes de caractères.
- les porteuses qui peuvent contenir une valeur.

1) Modèle temporel

Le modèle temporel défini pour l'ensemble CONLAN est constitué de deux types d'instants : (cf. III.4.3)

- les macro-instants
- les micro-instants

Exemple



Les valeurs soulignées représentent les valeurs stabilisées de la porteuse à la fin de l'évolution interne des micro-instants ; ces micro-instants correspondent à des boucles de stabilisation lors de la simulation.

ii) Types de valeurs

Les types de valeurs de BCL sont :

- entier, booléen, chaîne de caractères qui sont hérités de PRIMITIF CONLAN.
- signal qui est défini pour BCL avec les opérateurs logiques classiques.

Des sous-types peuvent être définis à partir des types de base. Dans BCL, on a défini les sous-types suivants : pint (entier strictement positif), nint (entier strictement négatif), bint (m,n : int). Ce dernier type est le type "entier borné" paramétré par les valeurs des bornes m et n.

Exemple

Le sous-type pint est défini comme suit :

```
SUBTYPE pint
   BODY
        ALL x : int with  x > 0 ENDALL
   END pint
```

Le type SIGNAL représente une séquence de valeurs utilisant le modèle temporel de CONLAN. C'est un type générique qui peut engendrer un type SIGNAL particulier suivant le type de valeurs de la séquence.

Exemple

TYPE signal(v: val_type) est la définition du type générique "SIGNAL". Dans le cas où v est booléen, on définit un type signal de booléens.

Les opérations sur ce type SIGNAL sont définies par des segments FUNCTION ou ACTIVITY.

Exemple

La fonction de retard sur un signal est définie comme suit :

```
FUNCTION delay (x : signal(v) ; d : pint) : signal(v);
```

Le signal(v) retourné par cette fonction est décalé de 'd' unités de temps par rapport au signal(x).

La fonction STABLE peut alors être définie en utilisant la fonction "delay" :

```
FUNCTION stable (x : signal (v); n : pint) : boolean ;
   RETURN
      if n = 1 then 1
      else [delay(x,n) = delay(x, n-1)]  & stable(x,n-1)
      endif
END stable
```

Si l'expression booléenne est vérifiée pour n ≠ 1, la valeur retournée est VRAI.

iii) Types de porteuses

Trois types de porteuses sont définis :
- porteuse type terminal,
- porteuse type variable,
- porteuse type rt_variable (rt pour register_transfer).

Initialement, les porteuses ne contiennent aucun signal. Pour chaque type de porteuse il a été défini au moins une opération de type "ACTIVITY" :
- pour les terminaux, l'opération "CONNECT" notée ".="
- pour les variables, l'opération classique d'affectation notée ":="
- pour les rt_variables, l'opération transfert notée "←"

iv) Structurations de données

Les types de structures de données sont les types classiques : tableaux, enregistrement.

v) Exemples

Dans les sous-paragraphes précédents, nous avons défini les langage BCL. Dans ce sous-paragraphe nous nous plaçons en tant qu'utilisateur de BCL pour décrire des éléments de circuits.
Le premier exemple est une description structurelle d'une bascule "RS" en termes de portes logiques NOR. Le deuxième exemple sera une description fonctionnelle d'une bascule "D" à partir de son graphe d'états.

Exemple 1

```
REFLAN BCL
DESCRIPTION rsff (IN r,s : signal (bool); OUT q,nq : btm0) ;

  BODY
     nq .= nor (r%1, q%1)
     q  .= nor (s%1, nq%1)
  END rsff;
```

Le type btm0 est un sous-type de terminal avec des valeurs de type booléen et une valeur par défaut 0 ; le signe % est la syntaxe simplifiée de l'appel de la fonction DELAY.

Remarque

Cette description utilise des signaux et des terminaux. Les instructions de calcul des états des sorties de la bascule sont exécutées parallèlement. Après des boucles de stabilisation, les valeurs des sorties sont déterminées.

Exemple 2

```
REFLAN BCL
DESCRIPTION dff(tsu, th, tp : pint)
              (IN d, ck : signal (bool) ;
               OUT q, nq : variable (bool,0));

ASSERT tp > th ENDASSERT
BODY
  ASSERT
     if (ck%th) & not(ck%(th+1)) then
           stable(d, tsu+th) & stable0(ck%th, tsu)
           & stable1(ck, th)
     else 1
     endif
  ENDASSERT
  if (ck%(tp -1)) & not(ck%tp) then q := d, nq := not d;
  endif
END dff
```

Remarques

- cette description est paramétrée par "tsu, th, tp" (trois paramètres temporels). Ces paramètres doivent être fixés lors d'une instantiation.

- deux assertions ont été données dans cette description. La première se situe en dehors du corps de segment. C'est une assertion concernant les paramètres de description et sera vérifiée lors de l'instantiation du modèle (cf. assertion statique de CADOC.LD). La seconde concerne la stabilité des entrées de la bascule, à savoir l'horloge "ck" et l'entrée "d". Cette assertion se trouve dans le corps du segment et sera vérifiée pendant la simulation : c'est la condition temporelle d'activation de la bascule (cf. assertion dynamique de CADOC.LD).

- La dernière instruction "if" spécifie la condition de changement d'état de la bascule. Cette condition étant un front montant de ck : ck vaut 1 à (tp-1) et 0 à tp (tp = temps de propagation).

- Les descriptions fonctionnelles utilisent des objets de type VARIABLE. Les instructions sont exécutées si les conditions associées (if ...) sont vérifiées (on retrouve la sémantique des langages de description comportementale ou fonctionnelle). Ce type de description permet une meilleure efficacité durant la simulation : les actions ne seront exécutées que lorsque les conditions d'activations seront vérifiées.

III.5.4 - EXEMPLE DE DERIVATION DE LANGAGE : WISLAN

Comme application des mécanismes de dérivation de langage, nous allons brièvement présenter le langage WISLAN [DIE 83].

WISLAN est un langage orienté réseaux prédiffusés. Il est directement dérivé de BCL. Un ensemble de logiciels ([VAI 83]) travaille à partir de WISLAN pour optimiser un réseau de portes en respectant les contraintes technologiques, et pour l'interfaçage de la decription originale ou modifiée en vue d'une simulation.

Le niveau de description de WISLAN est donc le niveau porte logique. La description abstraite du comportement du circuit est très limitée.
Le segment CONLAN spécifiant WISLAN à partir de BCL contient les objets suivants :

- portes génériques : AND, OR, XOR, NAND, NOR, XNOR. Ces portes sont définies avec un paramètre générique qui est leur "fan-in" (de type pint),
- inverseur NOT,
- multiplexeurs et décodeurs génériques dont la taille est paramètre générique,
- additionneur générique dont les paramètres génériques sont la taille de l'additionneur et le nombre de bits utilisés pour le calcul d'anticipation de la retenue,
- éléments de mémorisation (NOR-LATCH, NAND-LATCH, RS-ATCH,LSSD-LATCH, MS-JK-FLOP, MS-D-FLOP, MS-T-FLOP, D-FLOP) qui sont décrits comme une interconnexion de portes logiques, leur structure interne étant inconnue de l'utilisateur.

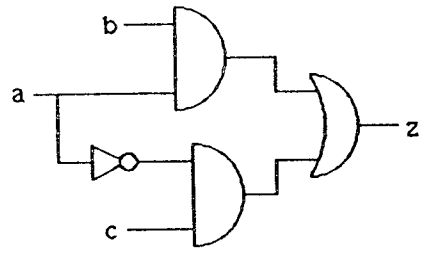
Les utilisateurs de WISLAN décrivent donc leur circuit structurellement à partir de ces primitives. Les facilités de définir des "FUNCTION", des "ACTIVITY" et des segments "CONLAN" ne sont pas disponibles dans WISLAN. Par contre les types terminal, entier, tableau sont hérités du BCL, et sont donc connus dans WISLAN.
Le seul type de segment hérité et disponible pour les utilisateurs est le type "DESCRIPTION".

L'activité "connect" symbolisé par ".=" est le type d'affectation dans WISLAN. L'affectation conditionnelle est possible en utilisant l'instruction "if ... then "

Exemple

```
z .= if a then b
     else c
```

L'interprétation en portes étant :



III.5.5 - REMARQUES

L'approche CONLAN est une approche différente de toutes les approches utilisées dans les langages de description de matériel : elle offre un environnement permettant de définir des langages spécifiques pour des outils de synthèse ou d'analyse bien déterminés, tout en se basant sur un ensemble de concepts communs.

Cette facilité de dérivation de langage à partir d'un autre langage, par exemple BCL, est très séduisante. Mais actuellement, peu de langages semblent être conçus à partir de CONLAN ; de plus certains membres du groupe CONLAN étudient la possibilité d'utiliser des langages comme ADA (cf. II.2.3).

Pour terminer cette partie, nous allons simplement signaler le système CASCADE ([BOR 85], [MER 85]) qui propose 6 langages de description différents, en fonction du niveau de modélisation désiré (LASSO : niveau système, LASCAR : niveau architecture, CASSANDRE : niveau RTL, POLO : niveau logique, CASTOR : niveau interrupteur et IMAG : niveau électrique) ; de plus la spécification d'actions séquentielles classiques se fait par définition de procédures ou fonctions FORTRAN. Le niveau électrique mis à part, le

concepteur doit donc maitriser 6 langages différents pour pouvoir exploiter complétement le système proposé. Notons que certains de ces langages ont été définis par dérivation à partir de BASE CONLAN et qu'un simulateur acceptant certaines descriptions mixtes a été défini.

Nous ne nous attarderons pas sur cette approche multilangage qui est à l'opposé de l'approche que nous allons introduire maintenant avec le langage CADOC.LD.

DEUXIEME PARTIE

LE SYSTEME ET LE LANGAGE CADOC.LD

III - LE SYSTEME CADOC

III.1 - OUTILS D'EXPLOITATION DU LANGAGE CADOC.LD

III.1.1 - Compilateur
III.1.2 - Editeur de liens
III.1.3 - Simulateur

III.2 - OUTILS ASSOCIES AU LANGAGE

III.2.1 - Test fonctionnel
III.2.2 - Synthèse de contrôleurs

III.3 - OUTILS INTERFACABLES AVEC LE SYSTEME CADOC

III.3.1 - Liaison CATA/CADOC : analyse de testabilité
III.3.2 - Liaison GAPT/CADOC : description de microprocesseurs

IV - CONCLUSION DE LA DEUXIEME PARTIE

Dans la partie précédente, nous avons étudié quelques langages de description du matériel, de cette étude, nous avons retiré un certain nombre d'idées et constaté des faiblesses sur certains points. La suite de cette étude a été la définition d'un langage spécifique à la description du matériel : le langage CADOC.LD.

Dans un premier paragraphe, nous allons présenter ce langage de façon informelle de manière à en définir ses grandes lignes ; une présentation détaillée sera faite dans une deuxième partie.

Nous présenterons ensuite le système de CAO organisé autour du langage CADOC.LD : nous énumérerons d'abord les outils immédiatement associés au langage (compilateur, simulateur, ...) puis les outils associés (test fonctionnel, synthèse de contrôleur,...).

Pour terminer, nous présenterons deux outils développés dans l'équipe et nous étudierons l'intérêt de leur interfaçage avec le système CADOC.

I - INTRODUCTION INFORMELLE

Dans ce paragraphe, nous allons simplement présenter les grandes lignes du langage CADOC.LD ([AMB 83], [AMB 84], [BEL 85]), une présentation complète sera donnée paragraphe II et la carte syntaxique est renvoyée en annexe.

Le langage CADOC.LD est un langage fonctionnel de description du matériel qui a été conçu pour satisfaire les objectifs suivants :

- indépendance vis à vis des technologies,
- pas de limitation liée à la complexité des circuits,
- facilité d'utilisation,
- indépendance vis à vis du système hôte,
- les descriptions doivent pouvoir servir de base à d'autres systèmes.

I.1 - Indépendance vis à vis des technologies

Par indépendance vis à vis des technologies, il faut entendre indépendance vis à vis de la réalisation physique du circuit dans une technologie donnée (MOS, TTL, ...). Une des conséquences de ce point étant qu'un langage voulant respecter cette contrainte doit offrir aux utilisateurs, non la manipulation d'un ensemble d'objets prédéfinis mais un ensemble de constructions suffisamment puissantes pour décrire les modèles de ces objets. Il est bien entendu évident que la définition de bibliothèques de descriptions et l'utilisation des facilités de structuration (à l'aide de programmmes de saisie de schémas électroniques) permet de décrire des circuits sans utiliser les mécanismes de base de modélisation de comportements.

I.2 - Pas de limitation liée à la complexité des circuits

De manière schématique, l'idéal serait de pouvoir décrire tout circuit au niveau d'abstraction le plus bas possible (niveau circuit : potentiels et courants fonctions continues du temps,...).

Cet idéal est d'abord irréalisable pour des raisons évidentes de quantité d'information à traiter par des méthodes complexes (sur ce point, on se reportera à la partie 3 de ce document) et de plus fournirait une masse inexploitable de résultats : parallèlement à l'augmentation de la complexité du circuit, on observe une nécessité d'abstraction des résultats fournis (les bus adresses et données par exemple, ne sont plus vus comme une concaténation de fils mais de manière globale comme une variable entière).
La gestion de la complexité passe donc par l'utilisation de modèles représentant "la" réalité physique d'un circuit avec un degré d'abstraction plus ou moins grand ; ces modèles pouvant être soit prédéfinis, soit construits à partir d'un langage de base. Cette dernière possiblité qui elle seule permet une "infinité" de niveaux d'abstraction est celle retenue dans le langage CADOC.LD.

I.3 - Facilité d'utilisation

Un des premiers critères de jugement pour un langage de description de circuits est la facilité de modélisation qu'il peut offrir ; cette facilité étant liée aux étapes de traduction à franchir par l'utilisateur lorsqu'il veut exprimer dans le langage les bases du fonctionnement de son circuit.
De manière simplifiée, les point suivants doivent avoir un équivalent immédiat dans le langage :

- manipulation explicite du temps,
- manipulation de chronogrammes,
- ordonnancement des actions,
- gestion du parallélisme.

On peut par exemple, considérer que ADLIB/SDL (HELIX) permet la modélisation du parallélisme (par l'intermédiaire de sous processus) mais que cette prise en compte se fait sous un formalisme très éloigné d'un formalisme "naturel" (faire A en parallèle avec B). Un autre exemple est l'impossibilité de la manipulation explicite du temps dans les langages de contrôle de processus (dérivés de CSP : OCCAM, ADA).

Pour résumer, nous avons retenu les caractéristiques suivantes lors de la définition du langage CADOC.LD :

- manipulation explicite du temps,
- ordonnancement des actions représenté sous forme d'un graphe temporisé,
- expression du parallélisme immédiate (parallélisme du graphe ou des actions),
- utilisation de la notion de chronogramme en remplacement de celle classique d'expression.

I.4 - Descriptions utilisables comme base pour d'autres outils

Nous verrons dans la troisième partie de cet exposé que le langage CADOC.LD peut servir de base à un ensemble d'outils de CAO : outils de test fonctionnel, de génération de parties contrôle et nous étudierons un interfaçage possible avec des outils déja existants : analyse de testabilité, génération de programmes de test pour microprocesseurs.

En conclusion de cette introduction informelle, la figure 1 résume les principes de base du langage CADOC.LD :

- un circuit est décomposable de manière hiérarchisée en un ensemble de sous circuits,

- la communication entre sous circuits se fait par l'intermédiaire des ports d'E/S, les mécanismes d'activation en fonction des valeurs de ces ports étant définis dans les ressources elles mêmes,

- le langage de description des comportements est un langage fonctionnel (au sens donné dans les définitions figurant en partie 1) qui ne comporte donc pas de modèles prédéfinis ; la définition du comportement d'un circuit feuille est basée sur un modèle graphique assurant une bonne lisibilité des descriptions ; le langage autorise le travail au niveau des chronogrammes et la manipulation explicite du temps est immédiate.

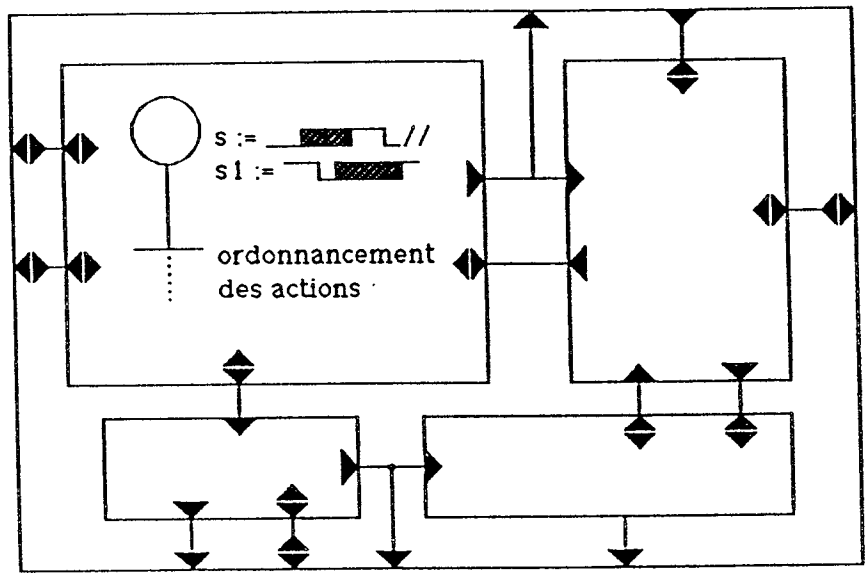


figure 1
aspect schématique d'une description d'un circuit en CADOC.LD

II - LE LANGAGE CADOC.LD

II.1 - Modèles de circuits en CADOC.LD

Dans le langage CADOC.LD, un circuit (ressource, ressource fonctionnelle) est défini formellement comme l'occurrence d'un modèle générique paramètré (**Ressource Générique Fonctionnelle** ou **RGF**) ; le mécanisme d'instantiation étant donné figure 2.

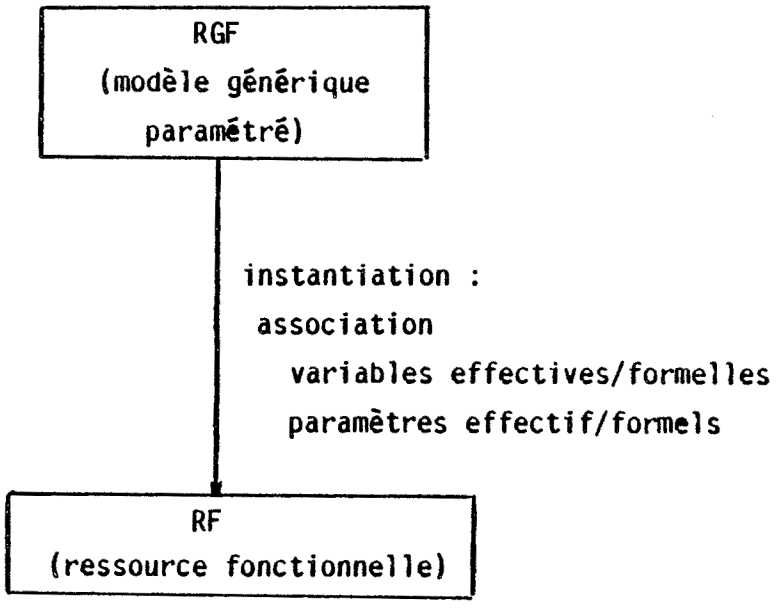


Figure 1 : RF = occurrence d'une RGF

La description d'une ressource fonctionnelle qui est un modèle du circuit est composée de 3 parties :

- un en-tête,
- une partie déclarative,
- une partie fonction.

Avant de présenter séparément ces 3 rubriques nous allons d'abord définir les types de données manipulées en CADOC.LD, ainsi que les opérateurs sur ces types.

Remarques

Pour des raisons de facilité d'écriture lors de la modélisation de circuits effectuant des opérations complexes, il a été nécessaire d'introduire la notion de **ressource algorithmique** qui est la traduction immédiate de la notion de fonction dans les langages de programmation classique.
Nous reviendrons sur les ressources algorithmiques au paragraphes VI.

II.2 - Types de données et opérateurs sur ces types

Comme nous le verrons ultérieurement, toute variable utilisée dans une RGF doit être déclarée et typée (cette "contrainte" permettant des vérifications de cohérence en particulier lors des interconnexions de différentes resources).

Nous allons d'abord présenter les types de base (ou types simples) ; des types structurés pouvant être créés à l'aide de constructeurs qui seront présentés ultérieurement.

II.2.1 - TYPES SIMPLES

Les types simples sont les types booléen, entier, front et énuméré ; les trois premiers types étant étendu par des valeurs symboliques X,Z,U dont la signification sera indiquée plus loin. Pour chaque type, on donne entre parenthèse le(s) mot(s) réservé(s) le spécifiant.

i) Type booléen (BOOL, BOOLEEN)

Ce type est un type énuméré dont les cinq valeurs sont :

- V (respectivement F) représentant la valeur "logique 0" (respectivement "1").
- X représente une valeur indifférente (soit V, soit F).
- U représente une valeur inconnue conséquence soit d'une non initialisation, soit d'un comportement de type transitoire du signal associé (si l'on considère les potentiels associés à ces valeurs on peut écrire $0 < U < 1$; il ne faut pas confondre ces inégalités avec les inégalités définissant l'ordonnancement logique des valeurs : $0 < U$ et $1 < U$).
- Z qui représente l'état haute impédance : la variable associée est déconnectée de toute source d'information.

* opérateurs sur le type booléen

Les opérateurs définis sur le type booléen sont :

- l'opérateur unaire de négation "non"
- les opérateurs binaires de comparaison <,< , =, >, >,<> .

Des exemples de tables donnant les résultats de ces opérateurs sont donnés en figure 3a, 3b, 3c et 3d.

Remarque

Lors de la définition de ces opérateurs, on est confronté au problème du choix de leur sémantique : soit d'une sémantique algorithmique, soit une sémantique matérielle.

Donner une sémantique matérielle à un opérateur revient à définir son comportement comme celui du dispositif matériel le réalisant : si par exemple on considère l'opérateur < et plus précisemment le résultat de V< U, de manière algorithmique et de par la définition des valeurs booléennes,le résultat est V ; par contre une réalisation de cet opérateur dans une technologie donnée peut donner un résultat différent (U par exemple).

Dans la mesure où les définitions choisies pour les opérateurs ne conviennent pas à une application donnée, l'utilisateur peut parfaitement redéfinir ses propres opérateurs par l'intermédiaire des constructeurs du type "choix".

A	V	F	X	Z	U
non A	F	V	X	U	U

figure 3a : opérateur non

<>	V	F	X	Z	U
V	F	V	F	V	V
F	V	F	F	V	V
X	F	F	F	V	V
Z	V	V	V	F	V
U	V	V	V	V	F

Figure 3b : opérateur <>

A < B

	V	F	X	Z	U
V	F	F	F	U	F
F	V	F	X	U	V
X	X	F	X	U	X
Z	U	U	U	F	U
U	V	F	X	U	F

Figure 2c : opérateur <

A ou B

	V	F	X	Z	U
V	V	V	V	V	V
F	V	F	X	U	U
X	V	X	X	U	U
Z	V	U	U	U	U
U	V	U	U	U	U

Figure 2d : opérateur OU

ii) Type entier (ENT, ENTIER)

Le type entier est une extension du type entier classique (entiers relatifs) avec les valeurs symboliques X,Z, et U ; X représentant toute valeur entière. Il est possible de définir des intervalles bornés ; un intervalle borné est de la forme [borne inférieure..borne supérieure].

*** Mode de représentation des entiers**

En entrée les entiers peuvent être donnés sous les formes suivantes :

- décimale signée : 328, -1024, ...
- hexadécimale signée : #0F = 15 en décimal
  #-FF = -1 en décimal
  -#FF = -255 en décimal
- binaire signée : &1110 = 14 en décimal
  &-1110 = -2 en décimal
  -&-1110 = 2 en décimal

En sortie les entiers sont donnés sous forme décimale signée.

* Opérateur sur le type entier

Les opérateurs définis sur les types entier sont :

- les opérateurs de comparaison $\leq, <, =, \geq, >, <>$.
- les opérateurs arithmétiques +, -, div, reste, *, **.

Ces opérateurs étant étendus pour prendre en compte les valeurs symboliques X,Z et U.

Remarques

- Un exemple de définition d'opérateur (+) sur le type entier est donné figure 4.
- Les entiers sont sans limitation de taille, indépendamment de la machine hôte (choix d'un mode de mémorisation en listes, transparent à l'utilisateur).

	valeurs non symboliques	X	Z	U
valeurs non symboliques	+ classique	X	U	U
X	X	X	U	U
Z	U	U	U	U
U	U	U	U	U

Figure 4 : opérateur +

Exemples

```
X : ENTIER                  /entier non borné/
Y : ⌊-2**15..2**15-1⌋ /entier en complément à 2 sur 16 bits/
```

iii) Type front (FRONT)

Le type front est un type énuméré dont les 2 valeurs sont :

- M (front montant)
- F (front descendant).

Ce type a été défini pour des raisons de facilité de descriptions de certains dispositifs sensibles non à des niveaux mais à des variations de niveaux des signaux (bascules sensibles au front,...)

Aucune opération n'est définie sur ce type.

iv) Type énuméré

Un type énuméré est défini de manière classique par la liste des éléments le composant ; cette liste étant étendue avec les valeurs symboliques X,Z et U.

Les seuls opérateurs définis pour les types énumérés sont les opérateurs d'égalité et de différence.

Exemple
Les modes d'adressage d'un microprocesseur classique peuvent être décrits par le type énuméré suivant :

(imm, dir, inx, ext, impl, rel).

Si x est une variable de ce type il est possible d'écrire des expressions de la forme "X = impl".

II.2.2 - Types structurés

Les types structurés sont construits par l'intermédiaire des constructeurs d'enregistrement et de tableaux.

i) Enregistrements

La définition d'un enregistrement se fait par énumération de ses champs, les champs d'un enregistrement étant nécessairement de type simple.
De manière similaire aux langages de programmation classique, il est possible de définir des champs conditionnels. Il faut noter que les champs conditionnels sont tous accessibles (en cours de simulation), quelles que soient les valeurs des sélecteurs de champs. Dans le cas d'enregistrements conditionnels, les sélecteurs sont du type de base.
L'accès aux champs d'un enregistrement se fait en postfixant le nom de la variable par le nom du champ.

Exemples

- Définition d'une micro-instruction composée de 2 champs : un champ mode d'adressage et un champ opération qui est un entier sur 16 bits :

```
ENREG
  adr : (imm, dir, inx, ext, impl, rel) ;
  op : [0..2**16-1] ;
FIN ;
```

- Enregistrement conditionnel :

```
ENREG
  CHOIX c : BOOL ;
      V : (c1,c2 : BOOL) ;
      F : (c3,c4 : ENTIER) ;
  FINCHOIX ;
  c5 : [0..255] ;
FIN ;
```

Remarques

- Dans les exemples précédents ainsi que dans ceux qui vont suivre, les mots clefs seront indiqués en majuscule ; cette convention est prise uniquement pour des raisons de lisibilité : le langage CADOC.LD ne faisant pas de distinction entre majuscules et minuscules
- les commentaires sont délimités par des "/".

ii)Tableaux

Le langage CADOC.LD autorise la définition de tableaux multidimensionnés ; lors de la définition d'un type tableau, seules les bornes supérieures sont données, les indices correspondants variant entre 0 et les valeurs maximales.

Le type d'un élément de tableau peut être un type simple ou un enregistrement, les tableaux de tableaux ne sont pas autorisés.

Exemple

Une mémoire de 512 mots, chacun des mots étant composé des 2 champs définis précédemment est definie par :

```
TABLEAU [511] DE
      ENREG
        adr : (imm, dir, inx, ext, impl, rel) ;
        op : [0..2**16-1] ;
      FIN ;
```

II.3 - entête d'une RGF

Dans l'entête d'une rgf on trouve :

- le nom de la rgf
- la liste de ses ports d'entrée sortie (cette liste est optionnelle : si elle est omise, elle sera générée automatiquement lors de l'analyse de la partie déclarative de la description)
- la liste des paramètres et de leurs types utilisés dans la description cette liste peut être vide.
- le type de la rgf (actuellement le type unique est le type opérateur ("op") ; en fonction de l'utilisation de la rgf (simulation, test, synthèse), des types différents permettant de faire certaines vérifications ou restrictions lors de la compilation sont envisageables).

Exemple

L'entête de l'unité arithmétique et logique donnée figure 5 est le suivant (les mots clefs sont en majuscule);

```
RGF alu_gen (ea, eb, cin, codop, s, cout ; PARAM : n : ENT) : OP ;
```

Si l'on omet la liste des ports d'E/S, cet entête devient :

```
RGF alu_gen (PARAM : n : ENT) : OP ;
```

Remarque

Les paramètres sont obligatoirement soit de type entier soit de type booléen.

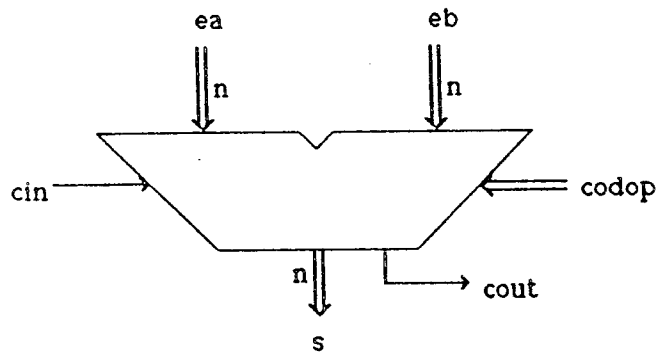


Figure 5 : UAL sur n bits

II.4 - Partie déclarative d'une Ressource Générique Fonctionnelle

II.4.1 - PARTIE DECLARATIVE DES RGF SIMPLES

Dans le cas d'une RGF simple, la partie déclarative contient :

- la déclaration des constantes,
- la déclaration des types utilisés,
- la déclaration des variables de la description,
- la déclaration des assertions statiques.

i) Déclaration des constantes (CONST)

Seule la déclaration de constantes entières est autorisée, la valeur de la constante pouvant être donnée dans l'une quelconque des représentations définies en III.1.2.

Exemples

```
ZERO = 0
DCC  = #FF
```

ii) Déclaration des types (TYPE)

Les types de base et les constructeurs ont été définis au paragraphe II.2 ; la déclaration d'un type permettant d'associer un identificateur de type à un type donné.

Exemples

```
entier_255 = ⌊0...255⌋ ;
entier_c2_255 = ⌊-128..127⌋ ;

adressage = (imm, dir, inx, ext, impl, rel) ;
micro_instr = ENREG
                  adr : adressage ;
                  op : ⌊0..2**16-1⌋ ;
              FIN ;
mem = TABLEAU ⌊511⌋ de micro_instr ;
```

iii) Déclaration des variables

Toute variable utilisée dans la description d'une RGF doit être déclarée et typée. Les types utilisés étant soit des types prédéfinis, soit des types complexes, soit des identificateurs de type définis dans la rubrique TYPE.

Une variable est déclarée dans l'une des 5 classes suivantes : ENTREE, SORTIE, BIDIR, VARINT, ALGO.

La classe **ENTREE** regroupe les variables apparaissant dans la liste des ports d'E/S de la RGF et utilisables uniquement en lecture.

La classe **SORTIE** regroupe les variables apparaissant dans la liste des ports d'E/S de la RGF et utilisables uniquement en écriture.

La classe **BIDIR** regroupe les variables apparaissant dans la liste des ports d'E/S de la RGF et utilisables uniquement en lecture/écriture.

La classe **VARINT** regroupe les variables internes n'apparaissant pas dans la liste des ports d'E/S de la RGF. Ces variables sont utilisées comme interconnexions entre ressources constituantes dans le cas de RGF composées.

La classe **ALGO** regroupe les variables internes utilisées uniquement dans les parties algorithmiques des descriptions. Une variables de classe ALGO mémorise uniquement sa valeur à l'instant courant (pas de mémorisation de l'histoire passée, ni de possibilités d'affectations généralisées : cf. II.5.2).

Remarques

- Une variable de classe autre que ALGO mémorise en cours de simulation l'histoire de ses valeurs au cours du temps. A une variable est donc associée un échéancier qui est une liste de couples (valeur de la variable, date de prise de cette valeur).
- De part la nécessité de déclarer et de typer toute variable, le langage CADOC est fortement typé. Ce choix a été fait afin de permettre le maximum de vérifications lors de la compilation des descriptions.

iv) Déclaration des assertions statiques (ASST)

Les assertions statiques associées à une RGF servent à vérifier si cette RGF est utilisée correctement ; on peut distinguer deux types d'utilisation illégale d'une RGF : les occurences ne respectant pas des contraintes sur les paramètres figurant dans la RGF et les occurences qui sont non compatibles avec l'environnement.

Une assertion statique est constituée de 2 partie : une expression booléenne et un message d'erreur, ce message sera imprimé si l'expression booléenne est vérifiée.

* Contraintes sur les paramètres

Une RGF étant un modèle paramétrable de circuit, ce modèle peut n'être valable que pour certaines valeurs des paramètres. On peut par exemple définir un modèle de mémoire paramétré par le nombre de mots et la taille des mots ; pour des raisons technologiques, ces deux paramètres peuvent être limités par des valeurs maximales données.
Les assertions statiques sont donc utilisées ici pour prévenir toute utilisation d'un modèle en dehors du domaine pour lequel il est défini.

Exemple

Si n1 est le nombre de mots de la mémoire et n2 la taille d'un mot, l'assertion AS1 donnée ci-dessous oblige la capacité totale à rester inférieure à 65536 bits.

```
ASST
 as1 :
   n1*n2 > 65536 :
   ECRIRE ('assertion as1, associée au modèle "mémoire" non vérifiée') ;
```

*** Contraintes d'environnement**

Les assertions statiques peuvent être utilisées pour effectuer des vérifications de fan-in/fan-out lors de l'interconnexion de différentes ressources.

Considérons par exemple le circuit donné figure 5, si la sortie O de R1 est capable de piloter 20pF de charges et si la somme des capacités d'entrée de I1,I2 et I3 dépasse cette valeur, une erreur crée par l'environnement peut être détectée par l'intermédiaire d'une assertion statique.

Nous reviendrons sur ce type de vérification dans la partie consacrée aux extensions du langage (partie 3, paragraphe IV.2.3).

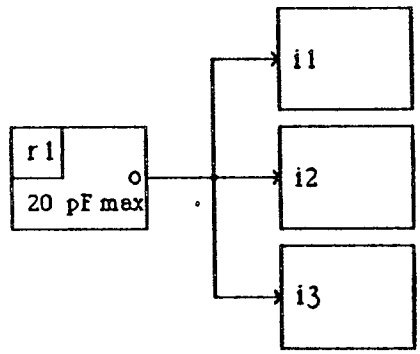


Figure 6 : Vérification de sortance

Remarque

Comme leur nom l'indique, les assertions statiques seront vérifiées en début de simulation, après la phase d'édition de liens ; nous verrons plus loin qu'il est possible de définir des assertions dynamiques qui seront vérifiées en cours de simulation.

v) **Exemple**

La partie déclarative de l'UAL donnée précédemment (figure 4) est la suivante

```
TYPE
  entier_n = ⌊-2**(N-1)..2**(N-1)-1⌋ ;
ENTREE
  ea,eb : entier_n ;
  c_in : ⌊0..1⌋ ;
  codop : (add,sub,neg) ; /type énuméré/
SORTIE
  c_out : [0..1] ;
  s : entier_n ;
ASST
 as1 : (n > 16):
       ECRIRE ("assertions AS1 du modèle alu_gen non vérifiée") ;
```

II.4.1 - PARTIE DECLARATIVE DES RGF COMPOSEES

Lorsque la description d'un circuit se fait par appel à des modèles de sous circuits, il est nécessaire de déclarer en plus de ce qui a été défini en II.4.1 :

- les entêtes des RGF utilisées avec la liste des variables et des paramètres qu'elles utilisent (rubrique RGF),
- les entêtes des ressources algorithmiques (RGA) utilisées (cf. II.6),
- les occurences de ces RGF avec la liste des variables et paramètres effectifs (rubrique RCONST),
- la liste (éventuelle) des connectiques explicites (rubrique CONNECT).

i) **Déclaration des RGF utilisées et de leurs occurrences**

Afin de simplifier les vérifications à effectuer lors de la phase d'édition de liens nécessaire dans le cas d'une ressource composée et pour rester cohérent avec l'optique de typage fort défini pour CADOC.LD, il est nécessaire de donner les entêtes des RGF utilisées.
Dans l'entête on trouvera une liste de ports d'E/S formels avec leurs types ainsi que la liste éventuelle des paramètres formels et leurs types.

La déclaration d'une occurrence d'une RGF déclarée se fera en donnant le nom de l'occurrence suivi du nom du modèle et de la liste des ports d'E/S et paramètres effectifs.

Exemple : Considérons le circuit composé donné figure 7.

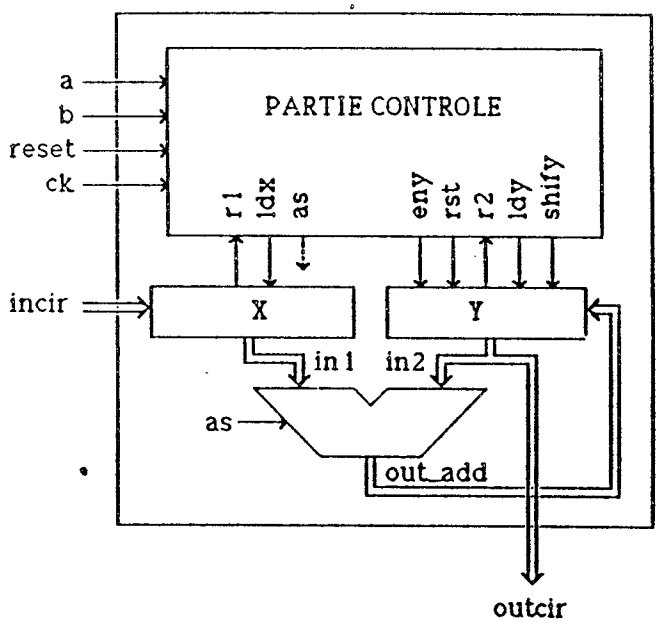


Figure 7 : Circuit composé

Sa partie déclarative est la suivante :

```
TYPE
  int_8 = [0..255] ;
ENTREE
  incir : int_8 ;
  a, b, reset : BOOL ;
  ck : FRONT ;
SORTIE
  outcir : int_8 ;
VARINT
  r1, r2 : BOOL ;
  ldx, ldy, rst, shfy, eny, as : BOOL ;
  out_add, in1, in2 : int_8 ;
RGF
  crtl_gen (ck1 : FRONT ;
            reset, in1, in2, loadx, loady, shift, outy, adsb : BOOL) ;
  srr_gen (load, shift, clear : BOOL ; in : int_8 ; msb  : ENTIER ;
           out : int_8 ; lsb : ENTIER ; PARAM n : ENTIER) ;
  ual_gen (in1, in2 : int_8 ; op : BOOL ; out : ENTIER) ;
  reg_reg (in, out : ENTIER ; load / BOOL ; PARAM n : ENTIER) ;
RCONST
  control : ctrl_gen
              (ck, reset, a, b, r1, r2, ldx, ldy, rst, shfy, eny, as) ;
  x : reg_gen (incir, in1, ldx ; 8) ;
  y : srr_gen (ldy, shfy, rst, out_add, zero, in2, - ; 8) ;
  add_sub : ual_gen (in1, in2, as, out_add) ;
CONNECT
  in2 = outcir ;
```

ii) Connectiques explicites

Dans l'exemple précédent, nous avons vu qu'une des manières possible pour effectuer une interconnection entre 2 ports distincts était de donner le même nom effectif à ces ports dans la déclaration des occurrences des ressources où ils figurent (figure 8a et 8b).
L'autre méthode revient à définir l'interconnexion entre deux variables dans la rubrique des connectiques explicites. Cette méthode est illustrée figure 9a et 9b.

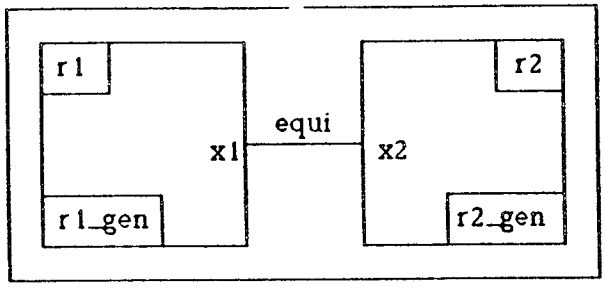


Figure 8a
connectique implicite,
schéma du circuit

```
...
VARINT
    equi : t1
...
RGF
    r1_gen (...; x1 : t1 ;...) ;
    r1_gen (...; x2 : t1 ;...) ;
RCONST
    r1 : r1_gen (...; equi ;...) ;
    r2 : r2_gen (...; equi ;...) ;
...
```

Figure 8b
connectique implicite,
partie déclarative

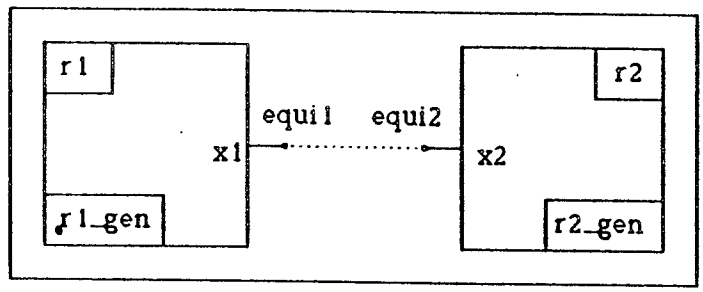


Figure 9a : Connectique explicite, schéma du circuit

```
...
VARINT
     equi1, equi2 : t1 ;
...
RGF
     r1_gen (...; x1 : t1 ;...) ;
     r2_gen (...; x2 : t1 ;...) ;
...
RCONST
     r1 : r1_gen (...; equi1 ;...) ;
     r2 : r2_gen (...; equi2 ;...) ;
...
CONNECT
     equi1 = equi2 ;
```

Figure 9b : Connectique explicite, partie déclarative

Remarques

- Les deux modes de description de la connectique sont équivalents.
- Les parties déclaratives des circuits composés peuvent être générées automatiquement à partir d'une représentation graphique du circuit et d'une bibliothèque de RGF, par l'intermédiaire de logiciels de capture de schémas électroniques ; cette solution est préférable à une description manuelle pénible et source d'erreurs.

II.5 - Partie fonction d'une RGF

II.5.1 - MODES DE DEFINITION DE LA FONCTION ASSOCIEE A UNE RGF

Le comportement d'un circuit peut être décrit en CADOC.LD de 3 manières :

- Juxtaposition des comportements des sous circuits composants (figure 10a): dans le cas d'une description hiérarchisée du circuit considéré, la fonction associée au circuit est obtenue par juxtaposition et communication entre les fonctions de tous les sous circuits. En conséquence, pour ce mode de description, la partie fonction proprement dite est vide.

- Utilisation des primitives de définition d'un comportement (figure 10b): dans le cas où le circuit considéré est un circuit feuille (ne faisant pas référence à des sous circuits), le comportement qui lui est associé sera décrit uniquement par l'intermédiaire d'un graphe interprété temporisé.

- Utilisation simultanée de sous-circuits et d'un graphe interprété temporisé (figure 10c) : dans certains cas un modèle de description mixte peut être souhaitable, le comportement est obtenu par la juxtaposition des comportements des sous circuits et de la partie fonction propre de la ressource.

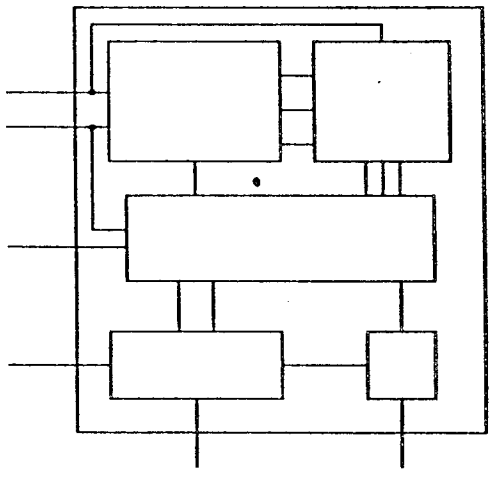


Figure 10a
Comportement décrit comme
juxtaposition des sous comportements

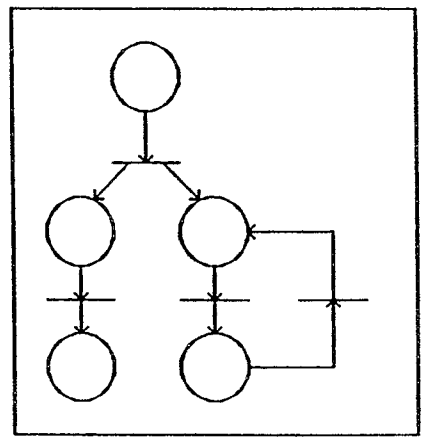


Figure 10b
Comportement décrit par un
un graphe interprété temporisé

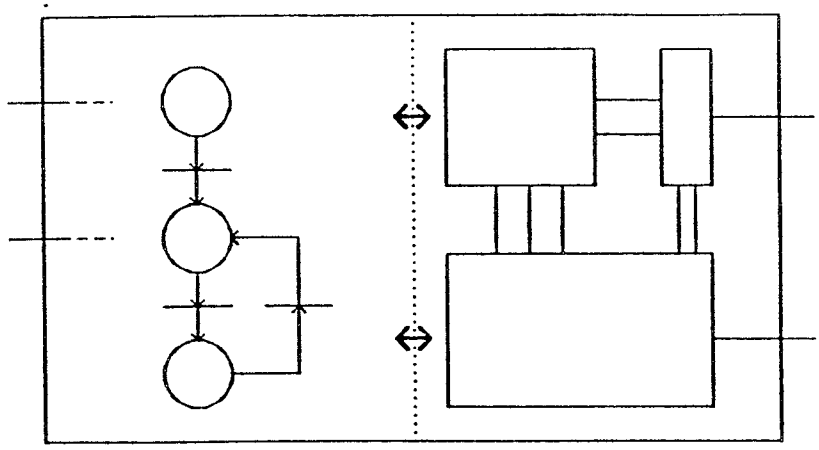


Figure 10c : Comportement décrit de manière mixte

Remarques
- Dans tous les modes de description, on peut définir des assertions dynamiques (cf. II.5.3).
- Dans le cas d'un comportement décrit par juxtaposition des comportements des sous-circuits, on se reportera au paragraphe II.4.2 concernant la partie déclarative des RGF composées.

II.5.2 - GRAPHE INTERPRETE TEMPORISE (GIT)

Le comportement d'un circuit feuille est décrit par l'intermédiaire d'un graphe interprété et temporisé ou **GIT**, dont la sémantique est proche du GRAFCET ou des Réseaux de Pétri [MOA 81].

Dans une première partie, nous allons présenter la composition d'un GIT puis nous introduirons les règles d'évolution définies pour ce modèle.

Un GIT est de manière classique un graphe biparti dont les 2 types de noeuds sont les **places** et les **transitions**, chaque noeud est repéré par un identificateur.
Aux places seront associées les actions ou opérations de modification des variables définies dans la RGF et aux transitions seront associées des réceptivités dont le franchissement permettra l'évolution du comportement en cours de simulation.

Remarque

il existe deux modes de représentation équivalents des GIT : un mode graphique et un mode textuel ; pour des raisons de lisibilité, nous utiliserons en général le modèle graphique.

Un squelette de GIT (non interprété, ni temporisé) est donné figure 11a dans sa représentation graphique et figure 11b dans sa représentation textuelle.

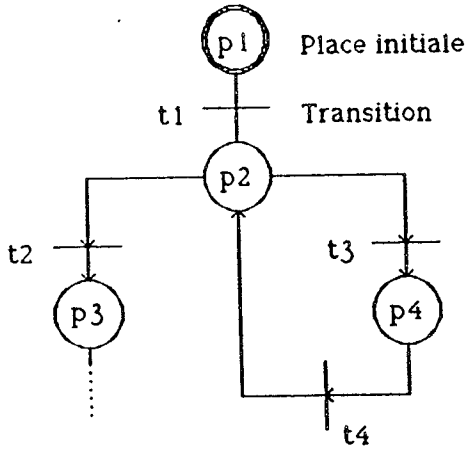


Figure 11a
squelette d'un GIT
(représentation graphique)

```
ACTION
    p1 : ...
    p2 : ...
    ...
GRAPHE
    t1 : p1 - p2 : ...
    t2 : p2 - p3 : ...
    ...
INIT
    p1
```

Figure 11b
squelette d'un GIT
(représentation textuelle)

i) - Blocs algorithmiques - chronogrammes

Nous allons présenter rapidement ici les mécanismes de construction de nouvelles valeurs à partir des variables et constantes définies dans le langage : nous introduirons d'abord ce qu'est une expression en CADOC.LD, nous parlerons ensuite de la notion de chronogramme, base du langage CADOC.LD. Nous terminerons par l'énumération d'un certain nombre de fonctions prédéfinies dans le langage.

* Expression - partie algorithmique du langage

Une expression définissant une valeur ou un ensemble de valeurs peut être construite en CADOC.LD, soit de manière classique, soit au travers d'un bloc algorithmique retournant une valeur ou une liste de valeurs.

*** Construction classique**

Une expression peut être construite à partir des variables ou des constantes déclarées à l'aide des opérateurs habituels ou de fonctions prédéfinies.

*** Construction algorithmique**

Dans certains cas, certaines manipulations ne sont pas exprimables sous forme d'une expression unique mais doivent être réalisées sous forme d'un algorithme. Il est possible de définir en CADOC.LD des blocs algorithmiques retournant une valeur (ou une liste de valeurs) à partir des instructions suivantes :

- affectation classique
- instruction conditionnelle : si-alors-sinon-fini
- instruction répétitive : tantque-faire-finfaire,
- instruction de choix : choix-dans-autres-finchoix,
- instruction de sortie : écrire, pause.

Ces instructions étant dérivées des instructions des langages impératifs classiques, nous ne développerons pas plus cette partie.

Exemple

Soit à calculer le nombre de bits à 1 d'une variable entière représentant une valeur sur 32 bits : on peut écrire le bloc algorithmique suivant :

```
DEBUT
     i=0 ;        /sera déclaré en ALGO/
     aux:=0 ;     /sera déclaré en ALGO/
     TANTQUE i < 31 FAIRE
          SI ((val div 2**i) mod 2 = 1) ALORS aux := aux + 1 FINSI ;
          i:= i+1 ;
     FINFAIRE
     RETOUR aux ;
FIN
```

Remarques

- Comme nous le verrons plus loin il sera possible d'utiliser ce bloc de la même façon qu'une expression ; on se reportera sur ce sujet au paragraphe sur la définition et l'utilisation des ressources algorithmiques.
- Dans la présentation du langage qui suit on utilisera des exemples ou figurent uniquement des expressions classiques, cela uniquement pour des raisons de clarté.

*** Fonctions prédéfinies**

Afin de permettre certaines manipulations sur le temps, un certain nombre de fonctions ont été prédéfinies dans le langage CADOC.LD.
Nous présenterons ultérieurement le modèle temporel précis du langage, mais en ce qui concerne ce paragraphe nous admettrons les points suivants :

- présence d'une horloge absolue initialisée à la valeur 0 (début de simulation) et incrémentée de 1 en 1 jusqu'à la date de fin de simulation,
- présence d'une variable CST indiquant le temps courant.

Notations

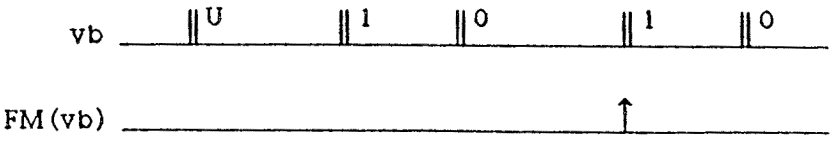
- Si ‹expr› est une expression, on notera val(expr) la valeur de cette expression.
- Si Y est une variable (de classe non ALGO), on notera Y(date) la valeur de Y à la date spécifiée.

*** fonction front montant (FM ou ↑)**

La fonction FM est appelée avec une variable (booléenne, front ou entière) non de classe ALGO comme seul paramètre, elle désigne les instants d'occurrences d'un front montant sur les valeurs de la variable.

* Cas d'une variable booléenne
Etant donnée une variable booléenne VB, il existe un front montant sur VB à l'instant t si et seulement si VB(t-1) = 0 et VB(t) = 1

Exemple



* Cas d'une variable de type front
Si VF est une variable de type front, il existe un front montant sur VF à l'instant t si et seulement si VF(t) = M.

Rappelons que le type front est constitué des 2 valeurs symboliques M (front montant) et D (front descendant).

* Cas d'une variable entière
La notion de front a été étendue aux variables entières de la façon suivante si VE est une variable de type entier (borné ou non), il existe un front montant sur VE à l'instant t si et seulement si :
VE(t) ∈ {X,Z,U}, VE(t-1) ∈ {X,Z,U} et VE(t) ≠ VE(t-1).

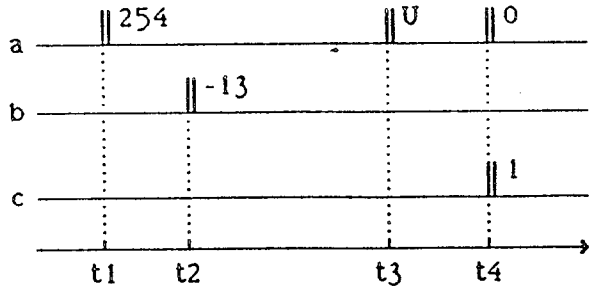
*** fonction front descendant (FD ou ↓)**

Cette fonction est similaire à la fonction FM.

*** fonction change (CHANGE)**

La fonction CHANGE est appelée avec comme paramètre une liste de variables (non de classe ALGO) et désigne les instants de variation des variables de la liste.

Exemple



CHANGE (a,b,c) désignera successivement les instants t1, t2, t3 et t4.

*** Fonction de temporisation (TEMPO)**

La fonction TEMPO est une fonction booléenne appelée avec les paramètres suivants :

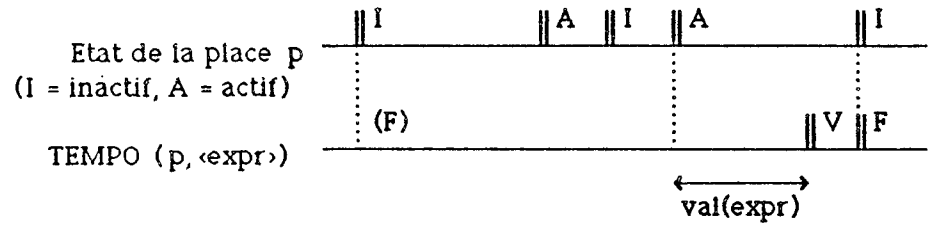
- un identificateur de place
- une expression de type entier

TEMPO (p, ‹expr›) retourne la valeur booléenne V si la place p est active depuis au moins val(expr) unités de temps, sinon la valeur retournée est F.

Remarque

On se reportera au paragraphe sur l'interprétation des GIT pour plus de précision sur l'activation et la désactivation des places.

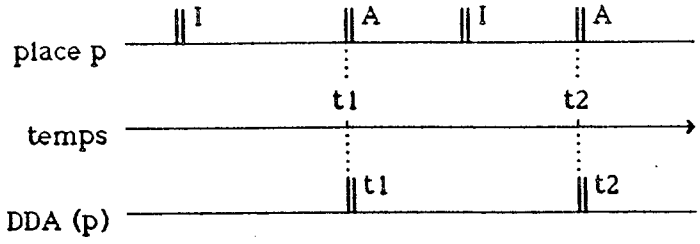
Exemple



*** Fonction d'accès à la date de dernière activation d'une place (DDA)**

La fonction DDA est une fonction entière qui renvoie la dernière date d'activation (date absolue) d'activation de la place passée en paramètre.

Exemple



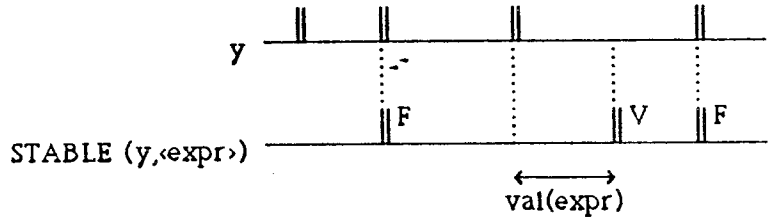
*** Fonction de stabilité d'une variable (STABLE)**

La fonction STABLE est une fonction booléenne appelée avec les paramètres suivants :

- un identificateur de variable (non de classe ALGO),
- une expression entière

STABLE (y, ‹expr›) est vraie si à l'instant courant, la variable y est stable depuis au moins val(expr) unités de temps.

Exemple



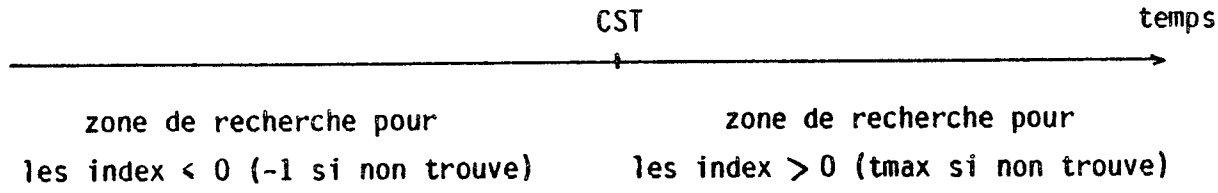
*** Fonction d'accès indexé aux dates de prises de valeurs (DOCX)**

La fonction entière DOCX est appelée avec les paramètres suivants :

- un identificateur de variable (non de classe ALGO)
- une expression dont le type est compatible avec le type de la variable
- une expression entière.

DOCX (y, ‹expr1›, ‹expr2›) désignera la date à laquelle y a été affectée à la valeur val(expr1) pour la val(expr2)ième fois.

En fonction du signe de val(expr2), la date retournée par docx sera inférieure ou supérieure au temps courant CST :



* **Chronogrammes**

Comme nous l'avons dit précédemment, toute variable qui n'est pas de classe ALGO mémorise non une valeur unique mais une suite de couples (valeur, date de prise de la valeur).

Sur l'histoire d'une variable on peut distinguer les 3 zones passé, présent et futur. L'histoire dans le passé est accessible mais non modifiable, l'histoire présente (valeur de la variable au temps courant) est accessible et éventuellement modifiable et l'histoire future est uniquement modifiable.

A la différence des langages de programmation classique qui ne permettent que l'affectation d'une valeur à l'instant courant (défini comme l'instant d'exécution de l'instruction associée) ou de certains CHDLS qui ne permettent que l'affectation d'une valeur à l'instant courant plus un délai, on peut affecter à une variable une suite de valeurs à des dates données.

On représentera cette suite de valeurs par une liste de couples : les figures 12a et 12b donnent deux exemples de chronogrammes et leurs traductions en CADOC.LD.

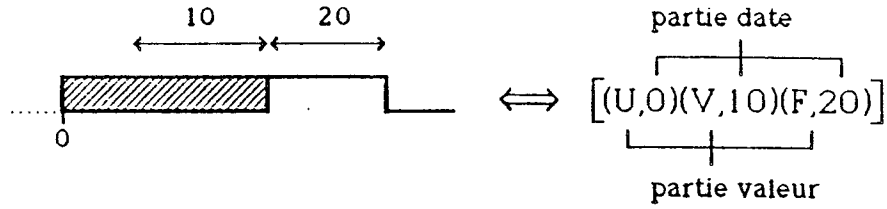


Figure 12a : Chronogramme (exemple 1)

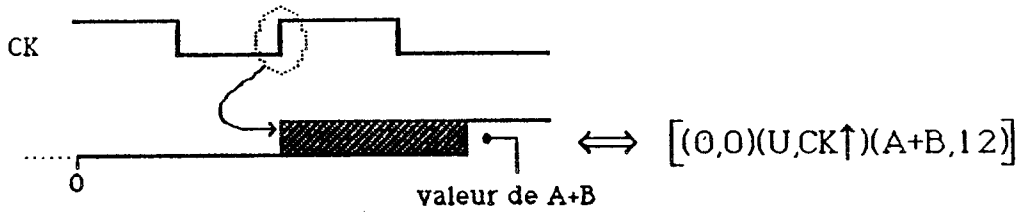


Figure 12b : Chronogramme (exemple 2)

*** Partie valeur d'un élément de chronogramme**

La partie valeur d'un élément de chronogramme est composée d'une expression dont la syntaxe est soit la syntaxe classique faisant éventuellement référence à un certain nombre de fonctions prédéfinies, soit un bloc algorithmique (voir paragraphe précédent) retournant une valeur ou une liste de valeurs. On doit avoir compatibilité entre le type des expressions et le type de la variable correspondante.

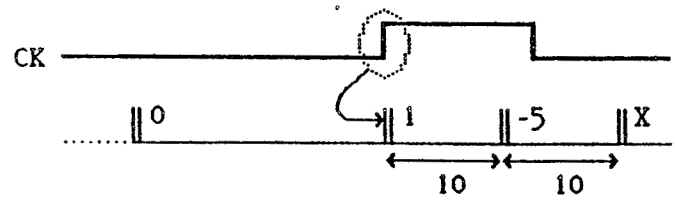
*** Partie date d'un élément de chronogramme**

La partie date d'un élément de chronogramme est soit une expression classique, soit fait référence à des fonctions prédéfinies permettant d'accéder à des dates absolues. Une date étant considérée comme une valeur entière, le type de l'expression doit être entier.

En fonction de l'appel éventuel à des fonctions prédéfinies renvoyant une date absolue, l'expression de date sera soit absolue, soit relative. Dans le cas ou l'expression renvoie une valeur relative, cette valeur est ajoutée à la dernière date absolue définie dans le chronogramme. Si la première expression date du chronogramme est relative, la date absolue correspondante est l'instant d'évaluation du chronogramme (voir paragraphe 7.2).

Exemple

[(0,10) (1,CK↑) (-5,10) (X,20)] définit le chronogramme suivant :



ii) Affectations généralisées

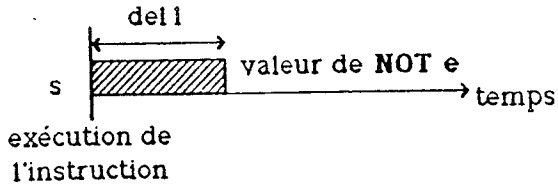
Une affectation généralisée est soit une affectation proprement dite (affectation simple, conditionnelle, à choix multiple) soit une affectation de sortie.

* Affectation simple

L'instruction de base en CADOC.LD est l'affectation d'un chronogramme à une variable ; le type de la variable et le type des expressions des parties valeur du chronogramme devant être compatibles.

Exemple 1

Le résultat de l'exécution de s := [(U,0) (NOT e, dell)] sera :



Exemple 2

```
nb_un :=
   DEBUT
     i:=0 ;
     .
     . /algorithme donné précédemment/
     .
     RETOUR aux ;
    FIN ;
```

Lors de l'exécution de cette instruction, la valeur de "aux" sera affectée à la variable nb_un.

*** Affectation conditionnelle**

Une affectation conditionnelle est de la forme :

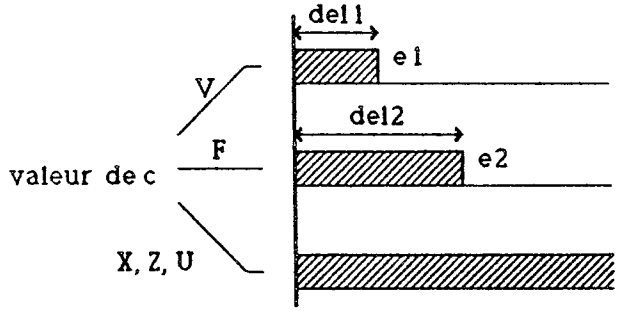
y := SI ‹expr_booléenne› alors ‹chronogramme1› [SINON ‹chronogramme2›] FINSI

La sélection du chronogramme affecté à y se faisant d'après la valeur de l'expression booléenne :

valeur du sélecteur	valeur affectée à Y
vrai	‹chronogramme1›
faux	si la clause SINON existe alors ‹chronogramme2› sinon pas d'affectation réalisée
autres (X,Z,U)	U

Exemple

y := SI c ALORS [(U,0) (e1, del1)] SINON [(U,0) (e2, del2)] FINSI sera interprété de la façon suivante.

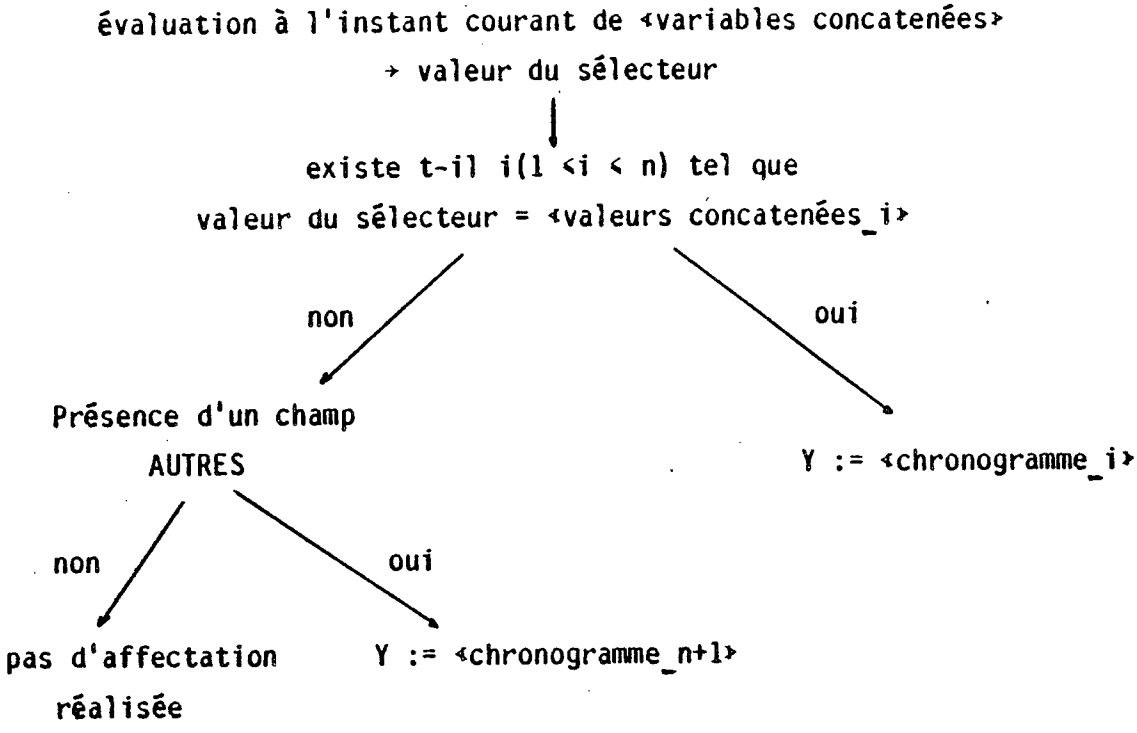


* Affectation à choix multiple

L'affectation à choix multiple est une généralisation de l'affectation conditionnelle ; sa syntaxe est de la forme :

```
Y := MUX ‹variables concaténées› DANS
       ‹valeurs_concaténées_1› : ‹chronogramme_1› ;
           ...
       ‹valeurs_concaténées_n› : ‹chronogramme_n› ;
       [AUTRES : ‹chronogramme_n+1› ;]
     FINMUX ;
```

La sélection de la valeur affectée à Y se fera de la manière suivante :



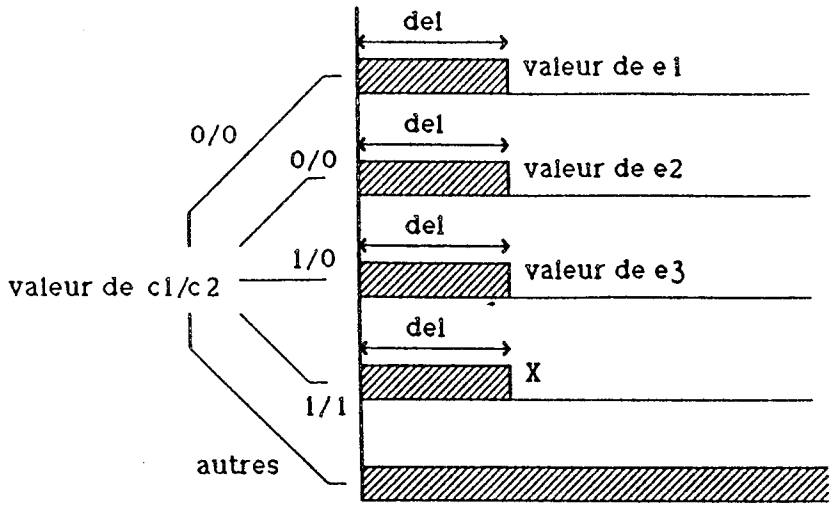
Remarque

A la compilation, aucune vérification n'est faite sur le caractère exclusif des choix possibles : le chronogramme correspondant au **premier** sélecteur satisfaisant la valeur courante des variables concaténées sera choisi.

Exemple

```
y := MUX c1/c2 DANS
        0/0 : [(U,0)(e1,del)] ;
        0/1 : [(U,0)(e2,del)] ;
        1/0 : [(U,0)(e3,del)] ;
        1/1 : [(U,0)(X,del)] ;
        AUTRES : U ;
     FMUX ;
```

sera interprété de la manière suivante :



Remarque
L'affectation du type multiple choix peut être utilisée pour définir des opérateurs dont la sémantique est fonction de la technologie utilisée.

*** Affectation de sortie**

Nous désignerons par ce qualificatif ce que l'on entend habituellement par édition de message, une édition d'un message pouvant être considérée comme l'affectation d'une chaine de caractère à une variable d'un type particulier symbolisant le fichier de sortie.
La syntaxe d'une instruction de sortie est la syntaxe classique.

Exemple

```
ECRIRE ("valeur de A = ",A," au temps ",CST) ;
        /CST = temps courant de simulation/
```

iii) Places

A une place est associée une liste éventuellement vide d'affectations généralisées qui seront exécutées en parallèles (figure 11).

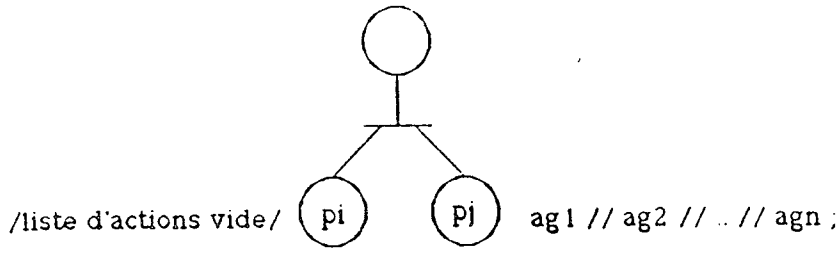


Figure 11 : places

Les actions associées à une place seront exécutées à l'instant d'activation de la place (voir paragraphe VII.2).

iv) Transitions

Le deuxième type de noeud apparaissant dans un GIT est la transition ; une transition est définie par :

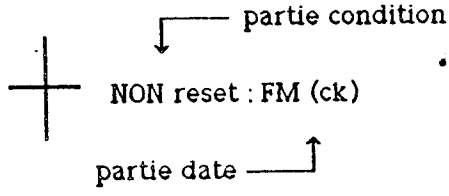
- l'ensemble de ses places amont,
- l'ensemble de ses places aval,
- une partie condition qui est une expression booléenne classique (les blocs algorithmiques n'étant pas autorisés),
- une partie date (ou partie évènement) qui est une expression renvoyant une date absolue construite à partir des fonctions prédéfinies de manipulation du temps (FM, FD ou CHANGE).

Remarques

Ces deux dernières parties sont optionnelles : si la condition est omise, la valeur VRAI est assumée, si la partie date est omise, le temps courant est assumé.

Une transition peut être sans place amont ou sans place aval.

Exemple



Remarque : Modélisation de signaux du type RESET (Transitions globales)

Tout circuit possède en général un certain nombre de signaux du type asynchrone qui lorsqu'ils sont actifs, forcent le circuit considéré dans un état donné, quels que soit son état courant.

Il est parfaitement possible de modéliser l'effet de ce type de signal en utilisant le modèle défini précédemment (GIT) mais cette modélisation se fait au détriment de la lisibilité de la description finale (figure 13c).

En effet on retrouve mélangé la description du circuit, son comportement "normal" (figure 13a) du circuit ainsi que son comportement d'exception (figure 13b). Pour éviter cette confusion, on peut introduire une transition globale (figure 13d) dont les informations seront :

- la liste des places ou cette assertion sera valable (le mot réservé TOUTE indiquant que toutes les places doivent être considérées),
- la condition et l'évènement déclenchant l'activation de cette assertion,
- la liste des places à activer lors de l'exécution de l'assertion.

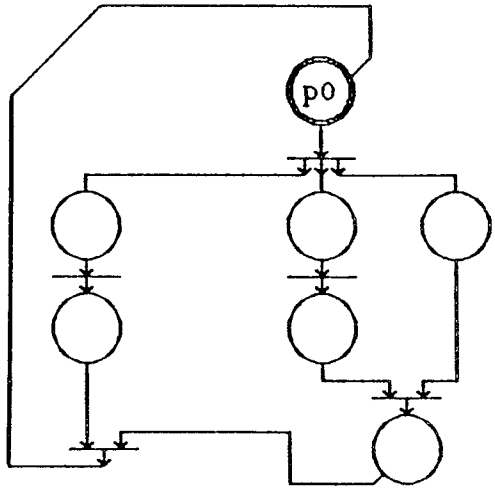


Figure 13a
squelette du GIT modélisant
le comportement "normal" du circuit

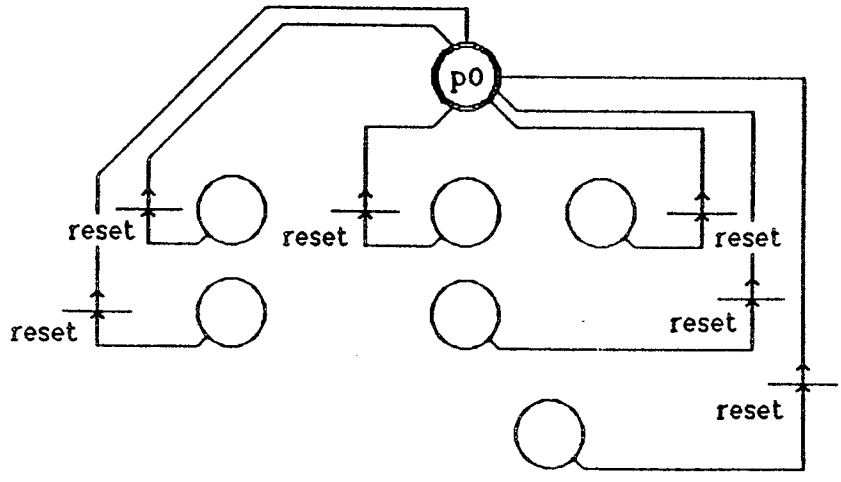


Figure 13b
squelette du GIT
modélisant l'effet d'un
RESET sur le circuit

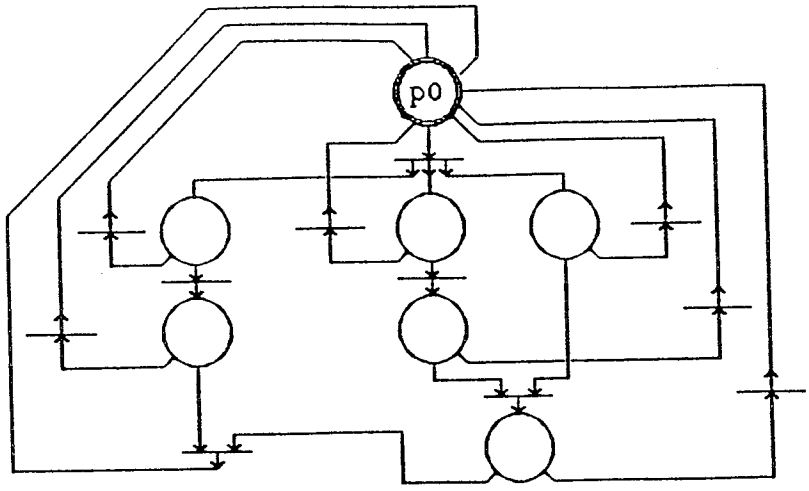
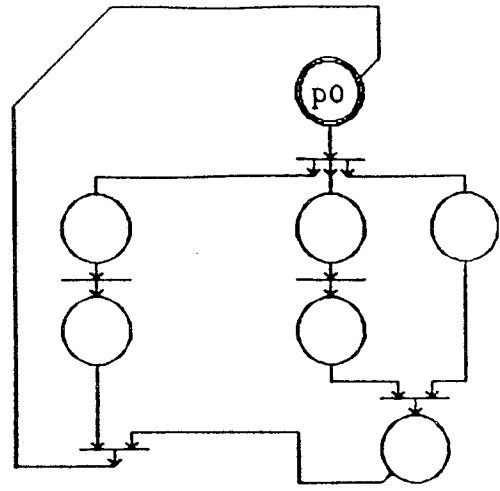


Figure 13c
Squelette du GIT avec superposition du comportement normal et du comportement d'exception



t_glo : TOUTE - p0 : reset

Figure 13d
définition d'une transition globale

Remarque
Si le GIT est saisi de manière textuelle, une transition globale et simplement définie en utilisant le mot clef TOUTE, dans le cas d'une saisie graphique en traitement spécial est à effectuer.

II.5.3 - ASSERTIONS DYNAMIQUES

Les assertions dynamiques ont été introduites en CADOC.LD pour permettre la spécification d'invariants à vérifier en cours de simulation.
Un invariant dynamique est défini par une propriété et des instants de vérification de cette propriété.
Les instants de vérification d'une propriété sont spécifiés par une liste de places ou par une liste d'évènements ; dans le cas ou la propriété donnée sous forme d'une expression booléenne est vérifiée, le message correspondant est édité. Dans le cas général, une assertion dynamique peut être assimilée à une réceptivité dont l'interprétation sera donnée ultérieurement.

Exemple

Une des applications des invariants dynamiques est la vérification des délais entre évènements : si l'on considère par exemple un registre à décalage avec les 2 commandes shift et load, un fonctionnement correct peut passer par le respect d'un intervalle minimum entre les occurences de ces signaux.

Les intervalles minimaux sont donnés dans la figure 14a ; les 4 assertions correspondantes sont définies dans la figure 14b.

	shift	load
shift	ΔS	ΔSL
load	ΔLS	ΔL

Figure 14a : intervalles minimums entre signaux

```
ASDYN
  ad1 : TOUTE : FM(shift) : DOCX(shift,0) - DOCX(shift,-1) > ΔS
      : ECRIRE ("ADI non respectée au temps ", CST) ;
  ad2 : TOUTE : FM(shift) : DOCX(shift,0) - DOCX(load,0) > ΔSL :
      : ECRIRE ("AD2 non respectée au temps ", CST) ;
  ad3 : TOUTE : FM(load) : DOCX(load,0) - DOCX(load,-1) > ΔL :
      : ECRIRE ("AD3 non respectée au temps ", CST) ;
  ad4 : TOUTE : FM(load) : DOCX(load,0) - DOCX(shift,0) > ΔLS :
      : ECRIRE ("AD4 non respectée au temps ", CST) ;
```

Figure 14b : Assertions dynamiques associées

II.6 - Ressources algorithmiques

Comme nous l'avons vu au paragraphe II.5.2, il est possible d'exprimer certains calculs complexes au moyen d'un bloc algorithmique construit à partir d'instructions classiques de langages de programmation et retournant un certain nombre de valeurs. Lorsqu'un même bloc algorithmique est utilisé plusieurs fois il peut être intéressant de l'utiliser comme une fonction au sens classique du terme.

II.6.1 - CONSTITUTION D'UNE RESSOURCE ALGORITHMIQUE

Une description de RGA (le terme de générique étant impropre car on ne peut pas effectivement parler d'une occurrence d'une ressource algorithmique) comporte les trois parties suivantes :

- un entête,
- une partie déclarative,
- un corps algorithmique,
- une liste d'expressions retournées.

Les deux premières parties sont identiques à celles des RGF à la différence qu'il ne peut y avoir de déclaration de RGF internes, ni de connectiques, ni d'assertions.
Le corps algorithmique est composé d'une suite d'instructions (II.5.2).

Remarque
Une RGA peut utiliser d'autres RGA, qui doivent être déclarées (voir paragraphe suivant), la récursivité n'étant pas admise.

Exemple

La description sous forme de ressource algorithmique du bloc algorithmique donné en II.5.2 et calculant le nombre de bits à 1 d'une variable codée sur 32 bits sera :

```
RGA nb_1 (val : [0..2**32-1] ; RETOUR nb : ENTIER) ;
  VAR i, aux : ENTIER ;
  DEBUT
     i := 0 ;
     aux := 0 ;
     TANTQUE i < 31 FAIRE
          SI ((val DIV 2**i) MOD 2 = 1) ALORS
              aux := aux+1 FINSI ;
          i := i+1
     FINFAIRE ;
     RETOUR aux ;
  FIN
FIN nb_1
```

II.6.2 - DECLARATION D'UNE RESSOURCE ALGORITHMIQUE

Une déclaration de RGA figure

- soit dans la partie déclarative d'une RGF ;
- soit dans la partie déclarative d'une RGA ;

Lors d'une déclaration on doit fournir

- le nom de la RGA
- la liste des paramètres formels d'appels ainsi que leurs types
- la liste des paramètres formels de retour ainsi que leurs types.

Exemple

La déclaration dans une rubrique RGA de la ressource algorithmique précédente sera :

```
nb_1 (val : [0..2**32-1] ; retour nb : entier) ;
```

II.6.3 - UTILISATION D'UNE RESSOURCE ALGORITHMIQUE

Une ressource algorithme s'utilise de la même manière que les fonctions dans les langages de programmation classique, à la différence qu'un appel à une ressource algorithmique peut retourner plus d'une valeur.

Exemple
Si la fonction précédente est étendue pour calculer à la fois le nombre de 1 et un bit de parité à partir de la variable passée en paramètre, on pourra écrire :

```
(nb1, parité) := nb_1_parité (val) ;
```

Remarque
La notation

```
(a,b) := ...
```

ne doit pas être confondue avec la notation

```
a,b := ... :
```

dans le premier cas, les variables a et b vont être affectées à des valeurs éventuellement égales retournées par la partie droite de l'affectation (deux valeurs retournées) ; dans le deuxième cas, les variables a et b vont être affectés à la même valeur retounée par la partie droite (une seule valeur retournée).

II.7 - Règles d'évolution

II.6.1 - MODELE TEMPOREL

Dans cette partie nous allons distinguer les deux points de vue possibles : le premier sera celui d'un utilisateur du système et ne fera pas référence aux mécanismes internes de gestion du temps qui seront présentés dans le deuxième paragraphe.

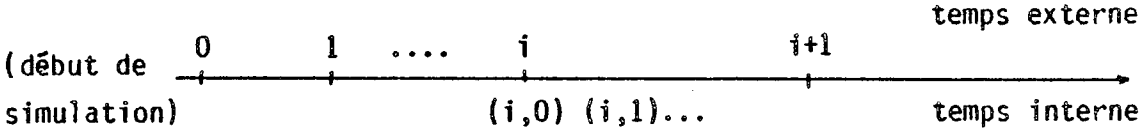
i) Modèle temporel externe (côté utilisateur)

Le temps est une suite discrête de valeurs entières strictement croissantes et d'incrément 1. Cette suite commencera à la valeur (date, instant) 0 qui indique le début de la simulation et se terminera à la valeur donnée par le temps maximal de simulation. On ne peut pas manipuler directement le temps absolu, les manipulations se font par l'application de fonctions prédéfinies sur des variables ; ces variables étant définies par des chronogrammes qui sont relatifs à un instant particulier appelé temps courant de simulation (CST).
Dans le temps externe, il n'existe pas d'instants accessibles entre 2 dates absolues consécutives.

ii) Modèle temporel interne

De manière interne, l'intervalle entre 2 instants consécutifs absolus est subdivisé en un nombre à priori illimité de micro-instants (figure 13).

Une date interne n'est donc pas un singleton mais un couple (date externe, microinstant).

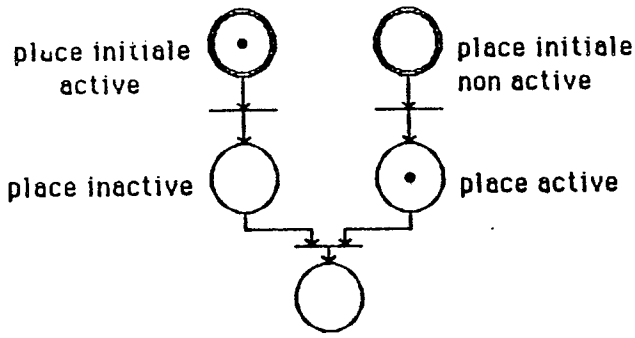


Les micro-instants seront utilisés pour prendre en compte les évolutions instables (ou instantanées) des GIT.

II.7.2 - REGLES D'EVOLUTION

A un instant donné une place peut être **active** ou **inactive**, l'ensemble des places actives définissant l'état du GIT correspondant.
A l'instant initial, l'ensemble des places actives est spécifié par la liste de places donnée dans la rubrique INIT de la partie fonction du GIT.

Une place active est indiquée par un point (jeton), une place initiale est indiquée par deux cercles concentriques :



L'évolution d'un GIT (évolution de son état) se fait par franchissement des transitions, pour être franchissable une transition doit d'abord être validée.

i) - Transition validée

Une transition est validée lorsque l'ensemble de ces places amont sont actives (figures 15a et 15b).

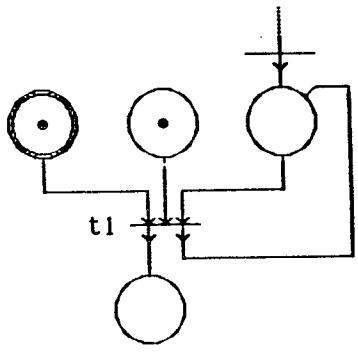


Figure 15a : T1 non validée

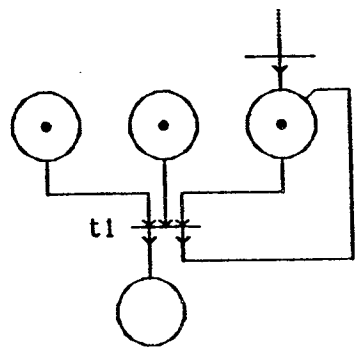


Figure 15b : T1 validée

ii) **Transition franchissable**

Une transition sera franchissable à l'instant absolu Θ si et seulement si :
- elle est validée à Θ,
- la condition booléenne de sa réceptivité est vraie à l'instant Θ,
- la partie date de sa réceptivité est égale à Θ.

L'instant ou une transition validée sera franchissable est fixé par la partie date de la réceptivité qui indique les instants d'échantillonage de la condition booléenne.

Nous allons développer maintenant les cas de figure suivants :
- la partie date de la réceptivité fait appel à FM ou FD,
- la partie date de la réceptivité fait appel à CHANGE,
- la partie date de la réceptivité est non spécifiée.

Ces trois cas qui correspondent aux définitions possibles de la partie date d'une réceptivité vont être étudiés sur des exemples isolés ; le mécanisme global d'évolution sera donné ensuite.

* **Appel aux fonctions FM ou FD**

Lorsque la partie date d'une réceptivité est spécifiée par un appel à la fonction FM (respectivement FD), les instants d'échantillonnage de la partie condition de la réceptivité (et donc les instants de franchissement possibles) seront les dates d'occurence des fronts montants (respectivement descendants) de la variable correspondante, à partir de l'instant courant de simulation.

Exemple

Si l'on considère
- le GIT donné figure 16a à l'instant CST,
- les chronogrammes de a et ck donnés figure 16b,

Alors la transition t1 sera franchie à la date absolue $\Theta 2$ (figure 16c).

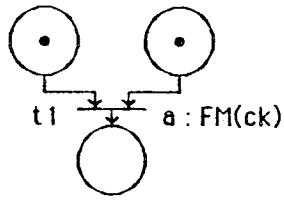


Figure 16a

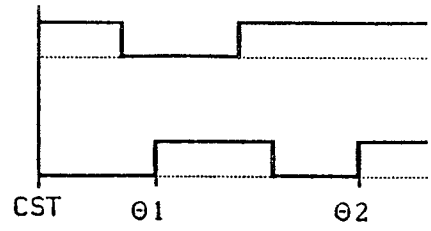


Figure 16b

	t1 validée ?	val(A) = V ?	Occurence de FM(ck) ?
CST	oui	oui	non
Θ1	oui	non	oui
Θ2	oui	oui	oui

Figure 16c

* **Appel à la fonction CHANGE**

Les instants de franchissement possible d'une transition dont la partie date fait appel à la fonction change sont les instants de changement de valeur des variables de la liste des variables associées.

Exemple

Si l'on considère

- le GIT donné figure 17a à l'instant CST,
- les chronogrammes de e, a, b et c donnés figure 17b,

Alors les instants de franchissement possibles seront Θ1, Θ2, Θ3 et la transition t1 sera franchissable à l'instant Θ2.

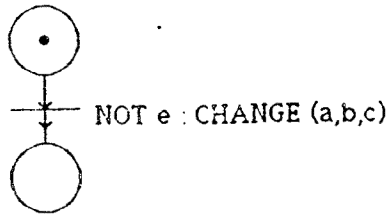


Figure 17a

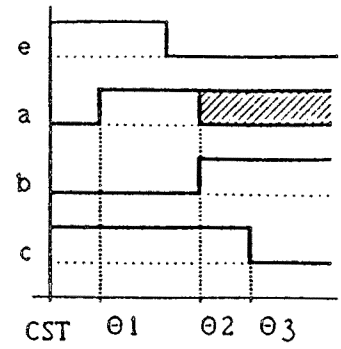


Figure 17b

*** Partie date non spécifiée**

Lorsque la partie date d'une réceptivité est non spécifiée (évènement "toujours" présent), la transition sera franchissable dès que la partie condition de la réceptivité sera vraie.

Exemple

Si l'on considère

- le GIT donné figure 18a à l'instant CST,
- les chronogrammes de a et b donnés figure 18b,

La transition t1 sera franchissable à l'instant Θ1.

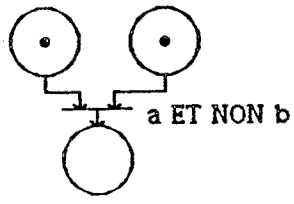


Figure 18a

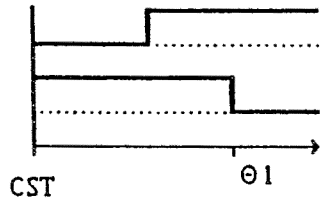


Figure 18b

Remarque

La transition de l'exemple est équivalente à "A et non B : CHANGE(A,B)".

iii) Franchissement des transitions

Etant données les règles d'évolution locales pour les différents types de transition énoncées précédemment, nous allons maintenant définir le comportement global d'un GIT.

* Obtention de la liste des transitions franchissables

A partir d'un GIT dans un état donné (défini par l'ensemble de ses places actives) à l'instant courant de simulation CST on détermine l'ensemble des transitions franchissables à l'instant $\Theta min > CST$.

L'instant Θmin est défini comme le plus petit instant auquel il existe une transition franchissable, cet instant peut être égal à CST dans le cas d'évolution instables ; de plus il peut exister plusieurs transitions franchissables à Θmin.

Exemple 1
Si l'on considère le GIT donné figure 19a et les chronogrammes à l'instant CST donnés figure 19b.
La transition t1 est franchissable à l'instant $\Theta 2$,
La transition t2 est franchissable à l'instant $\Theta 3$.
En conséquence l'instant de franchissement Θmin est $min(\Theta 2, \Theta 3) = \Theta 2$ et l'ensemble des transitions franchissables est {t1}.

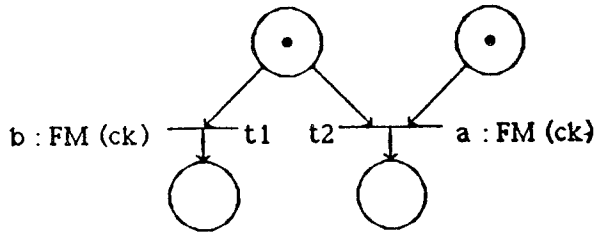


Figure 19a : GIT

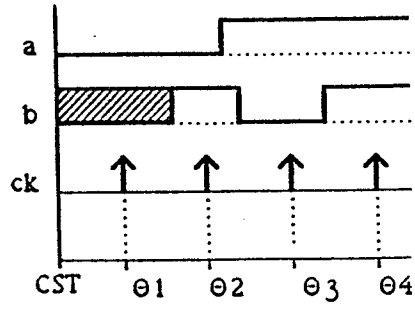


Figure 19b : chronogrammes

Exemple 2

Soit le GIT donné figure 20a et les chronogrammes à l'instant CST donnés figure 20b.

La transition t1 est franchissable à l'instant Θ2,
La transition t2 est franchissable à l'instant Θ2.
En conséquence l'instant de franchissement Θmin est Θ2 et l'ensemble des transitions franchissables est {t1,t2}.

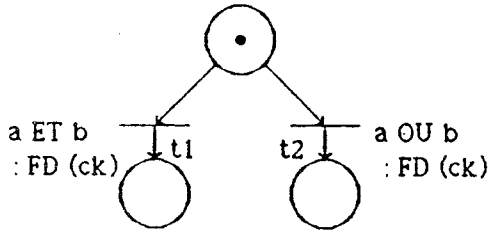


Figure 20a : GIT

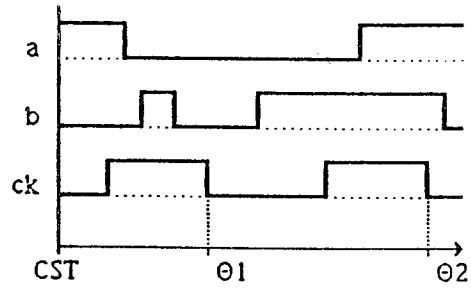


Figure 20b : Chronogrammes

*** Franchissement**

Etant donné un GIT dans un état défini par l'ensemble de ses places actives, après avoir déterminé l'instant minimal de franchissement et la liste des transitions franchissables (LTF) à cet instant on effectue le franchissement proprement dit :

- désactivation de toutes les places en aval des transitions de LTF,
- modification de CST qui devient Θmin,
- activation de toutes les places en amont des transitions de LTF et exécution en parallèle des actions associées.

Exemple

Soit le GIT donné figure 21a et les chronogrammes à l'instant CST donnés figure 21b ; l'instant minimal de franchissement est Θ2, la liste des transitions franchissables à Θ2 est LTF = {t1,t2}. Le franchissement de

{t1,t2} conduira à l'état du GIT donné figure 21c, l'instant courant sera Θ2 et on exécutera en parallèle les actions ag11, ..., ag1n, ag21, ...ag2m.

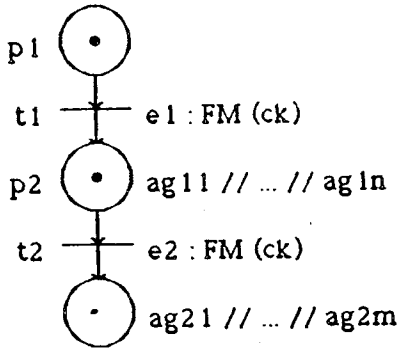


Figure 21a
GIT avant évolution

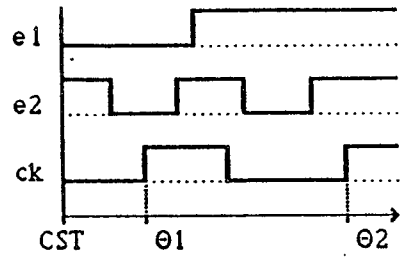


Figure 21b
Chronogrammes

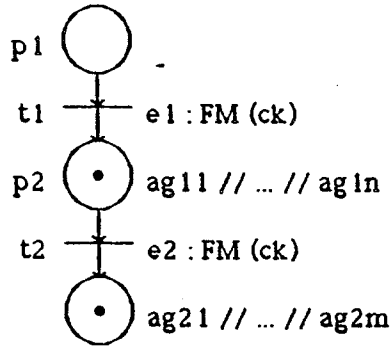


Figure 21c
GIT après franchissement simultané de t1 et t2

*** Remarques**

*** Franchissement parallèle**

Lorsque un GIT se trouve dans un état tel qu'à partir d'une place p il est possible de franchir simultanément plusieurs transitions pour lesquelles p est place amont, ces transitions seront **toutes** franchies simultanément.
Ce point est une des différences principales avec un Réseaux de Pétri pour lequel une seule des transitions, choisie de manière indéterministe serait franchie.

Exemple

Si dans le GIT donné figure 22a, les transitions t1, t2 et t3 sont franchissables alors elles seront franchies simultanément ; l'état du GIT résultant est donné figure 22b.

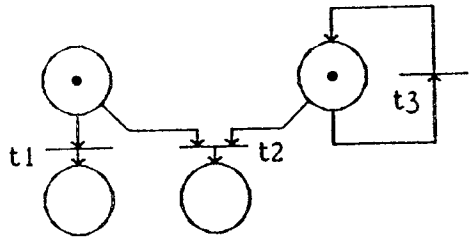


Figure 22a
GIT initial

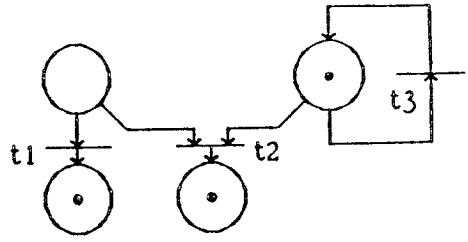


Figure 22b
GIT après franchissement de t1,t2,t3

*** Evolution instables, utilisation du temps interne**

Le temps interne qui est composé de 2 dates : une date externe seule connue de l'utilisateur et une date interne qui est utilisée pour les gestions des évolutions instables et pour permettre la consommation effective des évènements.

Exemple 1 : (consommation effective des évènements)

Si l'on considère le GIT donné figure 23a ainsi que les chronogrammes des variables a, b et ck donnés figure 23b et si l'on se restreint à un système de référence temporelle limitée aux instants externes alors à l'instant $\theta 1$ la transition t1 sera franchie ; p2 devenant active à $\theta 2$, la transition t2 alors validée et franchissable à ce même instant, ce qui n'est pas le résultat escompté (figure 23c).

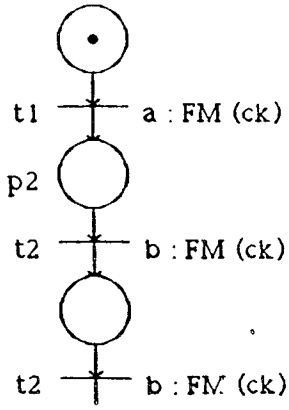


Figure 23a
GIT

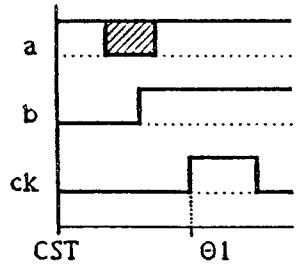


Figure 23b
Chronogrammes

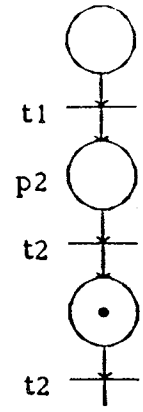


Figure 23c
Evolution incorrecte

Pour remédier à ce problème il faut associer aux transitions un délai non nul mais non mesurable ; cette association peut se faire soit de manière indirecte en gardant la liste des évènements non consommés et en gérant les listes de transitions instables : cette méthode a été appliquée dans une première version du simulateur mais s'avère complexe à mettre en oeuvre et pénalisante sur le plan du temps de calcul. L'autre solution choisie a été d'associer à toute transition un délai de franchissement égale à un microinstant (une unité de temps interne).

Si l'on considère le GIT donné figure 24, le temps courant CST étant Θ1, la transition t étant franchissable à Θ2 (avec éventuellement Θ2=Θ1) avec Θ1 = (ΘE1,ΘI1) et Θ2 = (ΘE2,ΘI2), la place p2 sera activée à l'instant (ΘE2,ΘI2+1).

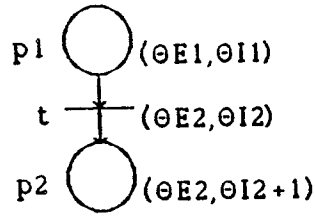


Figure 24 : Microinstants

Exemple 2

Nous allons donner ici un exemple plus complexe, basé sur le GIT dans l'état indiqué figure 25a, à l'instant (θ1,i) et les chronogrammes supposés figés sur la période [θ1..θ5] indiqués figure 25b.

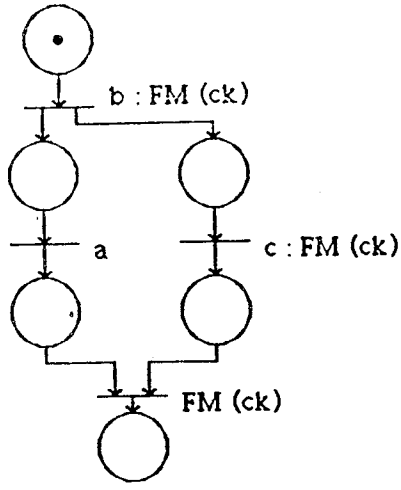


Figure 25a
GIT à la date θ1

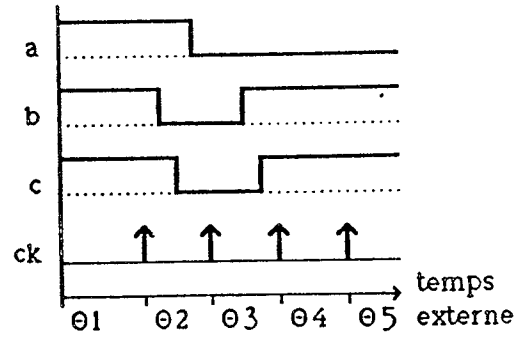


Figure 25b
Chronogrammes

Les dates d'activation des places et de franchissement des transitions sont alors données figure 25c :

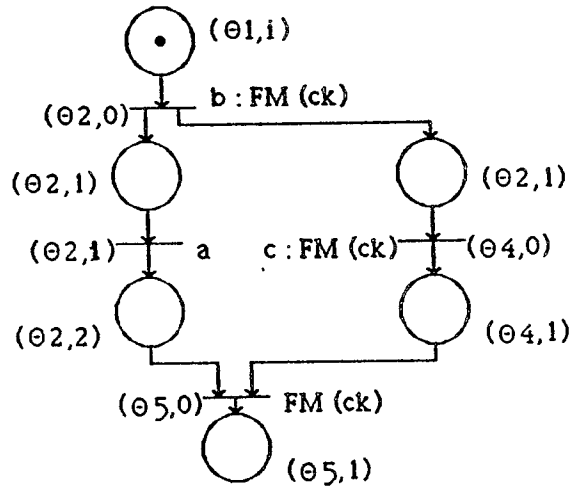


Figure 25c : Instants d'évolutions.

Exemple 3 (enchainement d'évènement)
Si l'on considère le GIT donné figure 26a à l'instant (θ1,i) et si on a un front montant de CK1 à l'instant (θ2,θ) alors les instants d'activation des places et de franchissement des transitions sont ceux donnés figure 26b.

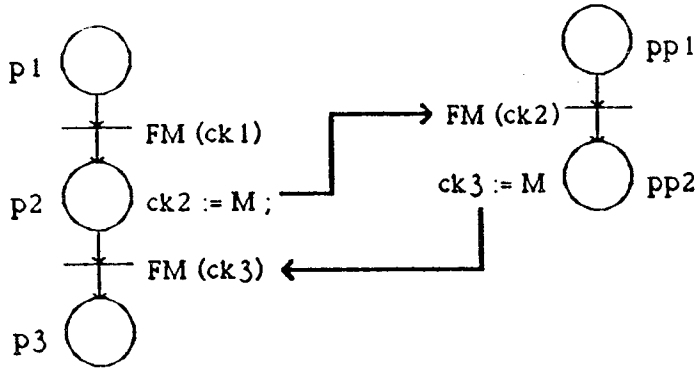


Figure 26a : GIT

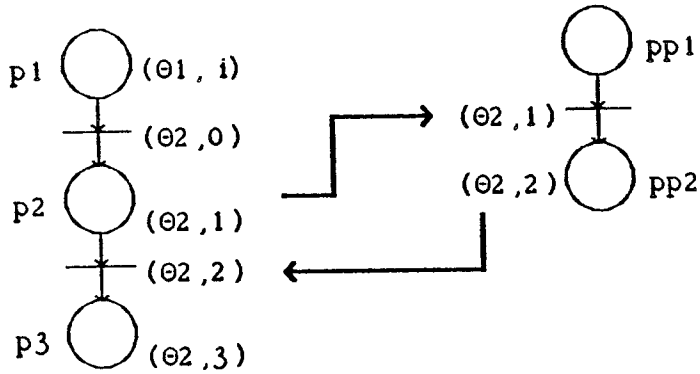


Figure 26b : Instants d'évolution

Remarque
L'exemple 3 illustre le mécanisme de communication entre GITs décrivant le comportement de plusieurs ressources.

II.8 - Exemple

Le circuit qui va nous servir d'exemple est un multiplieur 4 bits par 4 bits travaillant par additions décalages. Nous allons donner de ce circuit 3 modélisations différentes, ces trois modélisations seront données par ordre d'abstraction croissant.

II.8.1 - PREMIERE MODELISATION

Le brochage du circuit est donné figure 27a, sa décomposition en une partie opérative et une partie contrôle est indiquée figure 27b et la structure de la partie opérative est donnée figure 27c. De manière informelle, l'algorithme de calcul utilisé est donné figure 28.

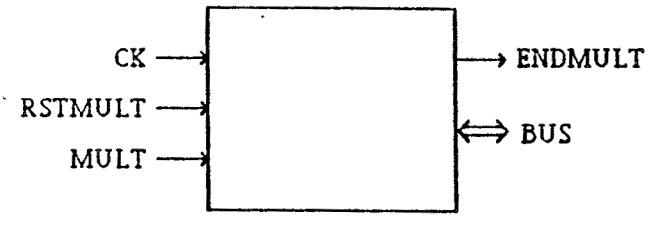


Figure 27a
brochage du multiplieur

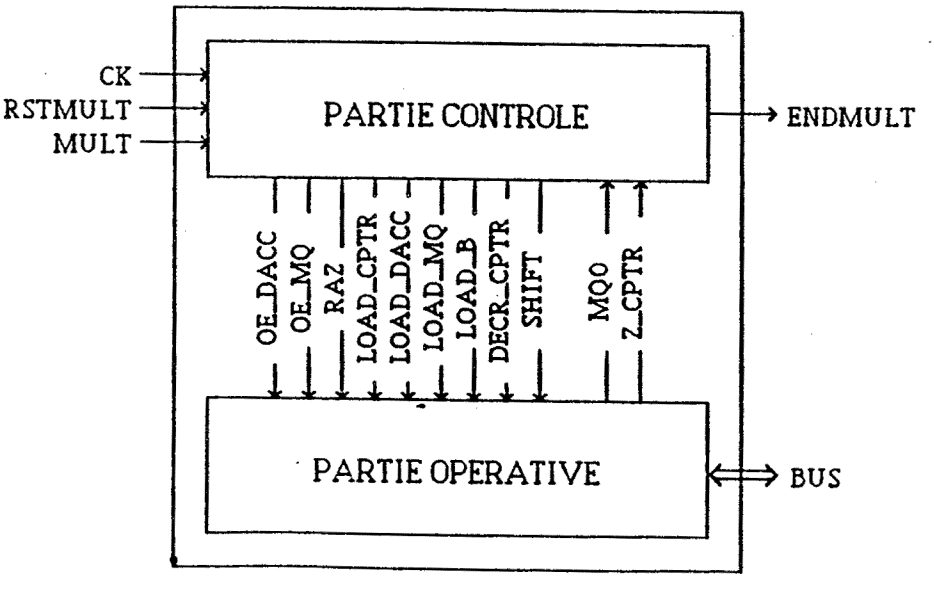


Figure 27b
décomposition PO/PC

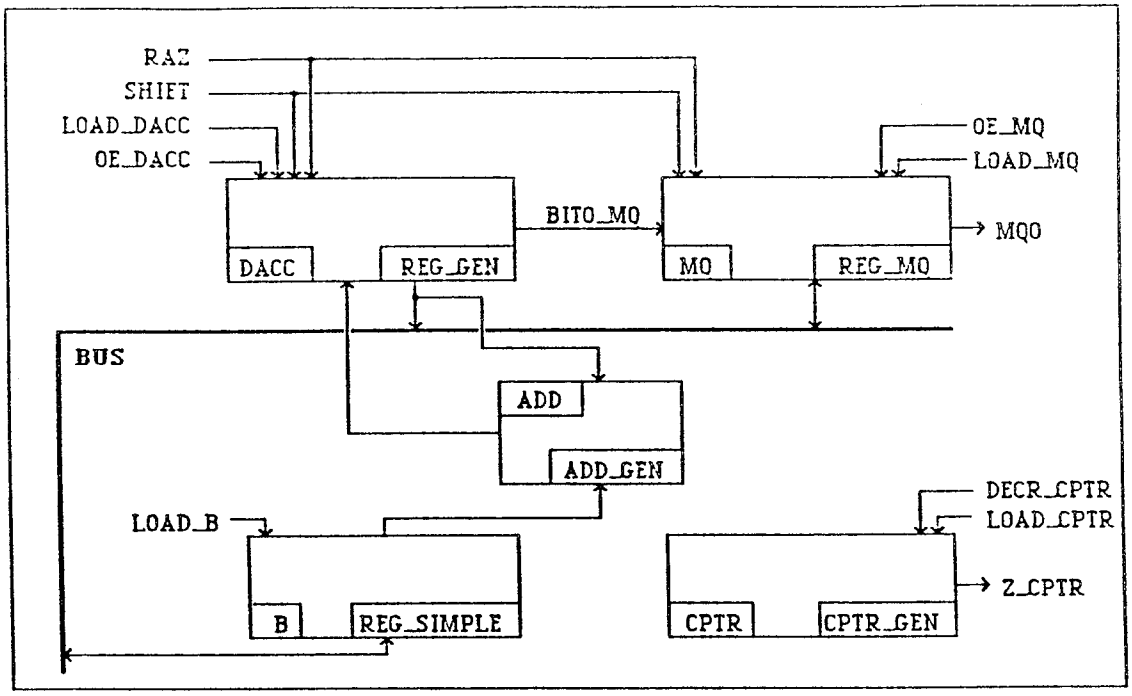


Figure 27c

structure interne de la partie opérative

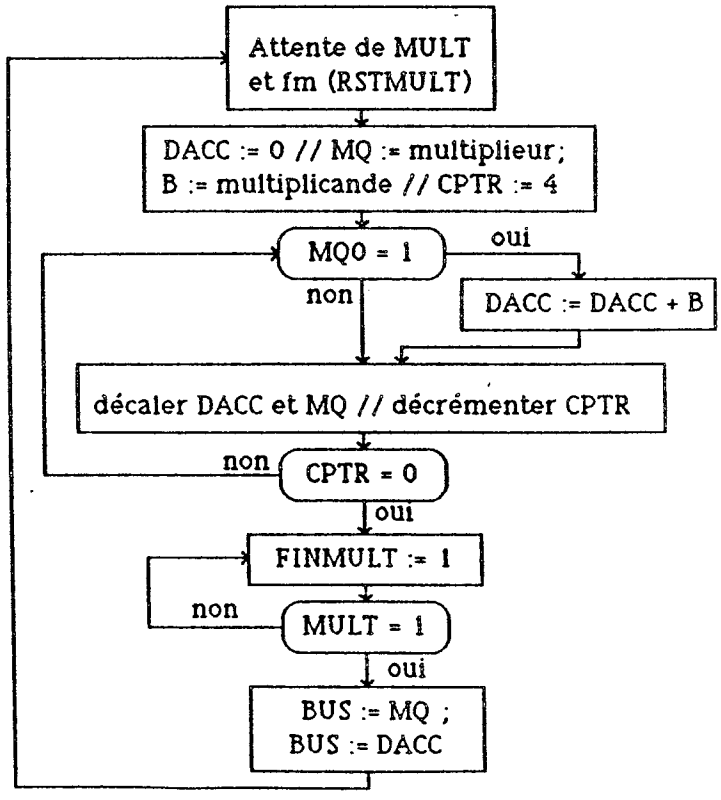


Figure 28

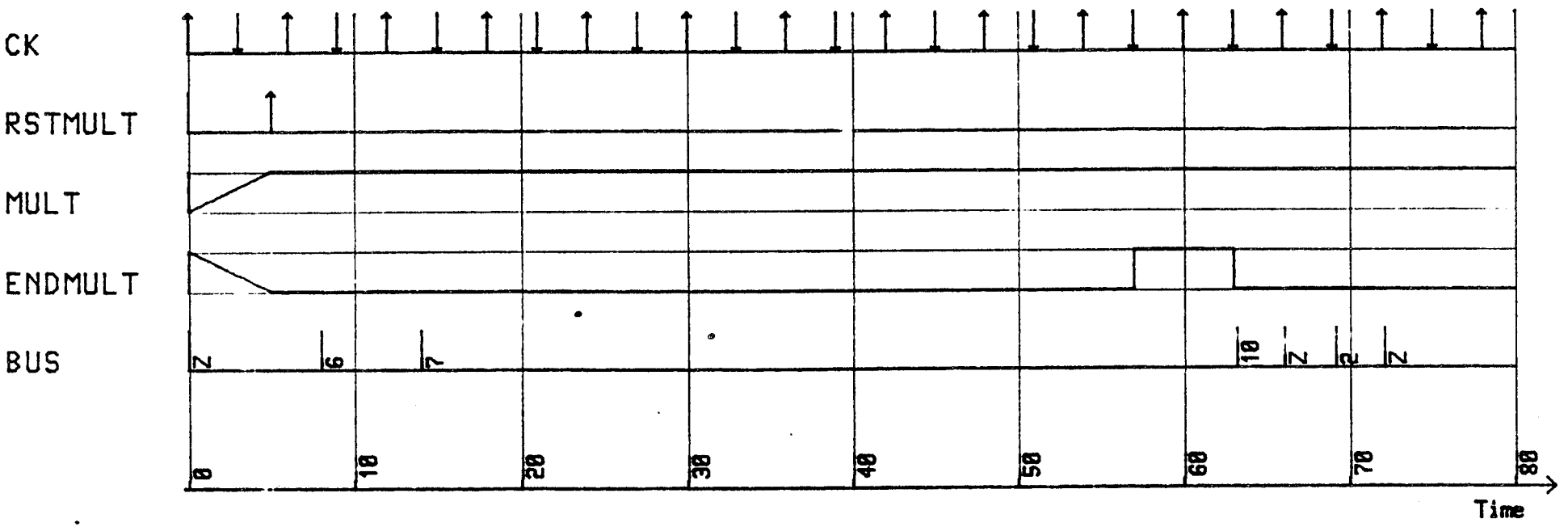
Algorithme de calcul

La première modélisation va suivre cette décomposition : à partir des descriptions des circuits feuilles REG_GEN, REG_MQ, REG_SIMPLE, CPTR_GEN et ADD_GEN (renvoyées en annexe), on construit par assemblage le modèle associé à la partie opérative (RGF MULT_PO : figure 29, page suivante).

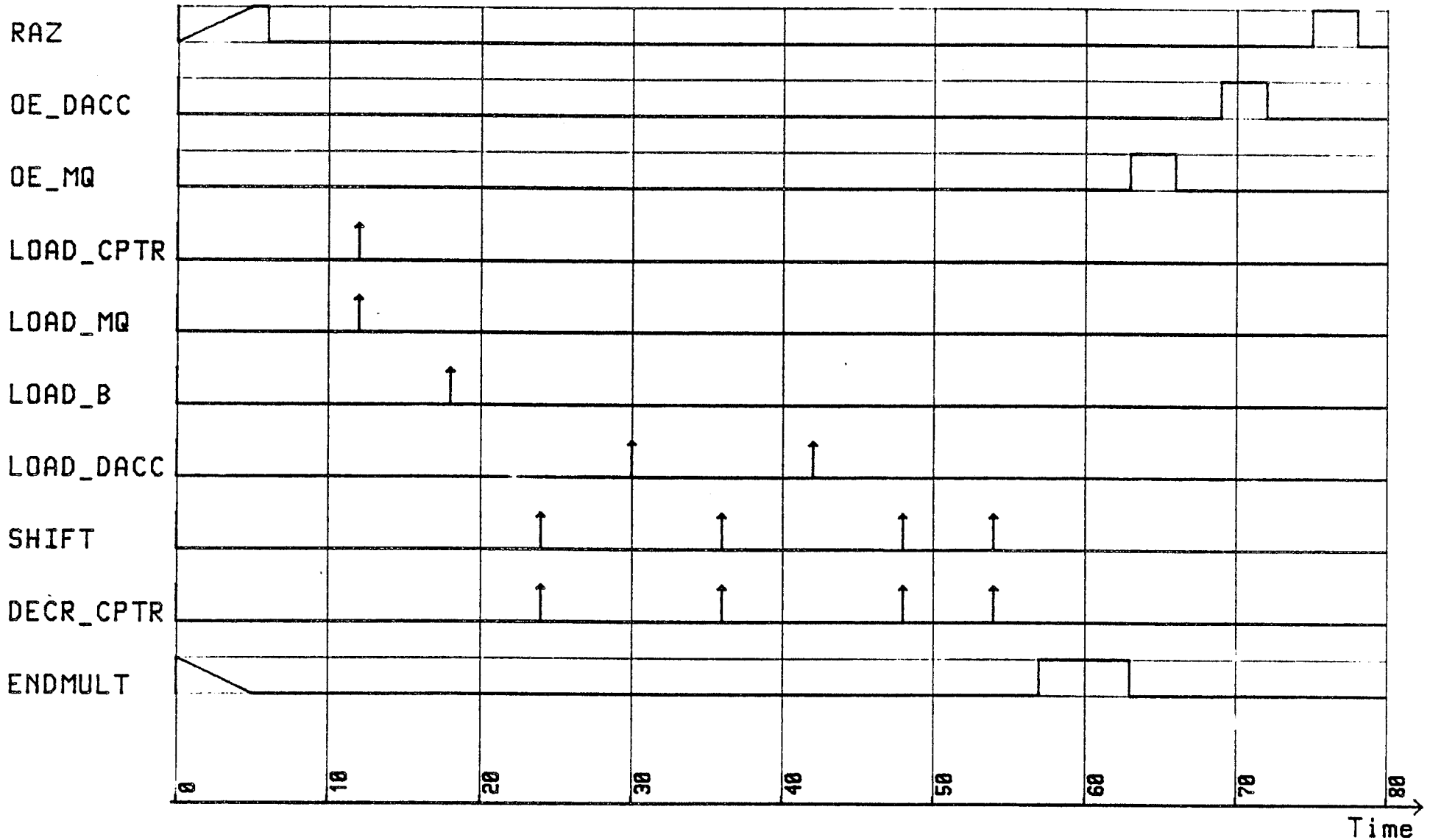
La modélisation de la partie contrôle est obtenue de manière immédiate à partir de l'algorithme indiqué précédemment, la description correspondante (RGF MULT_PC) est renvoyée en annexe.
Par un dernier assemblage entre une occurrence de MULT_PO et une occurrence de MULT_PC, on obtient le modèle du circuit final.
Les résultats de simulation de cette première modélisation sont donnés dans les deux pages suivantes (p. 172-173).

```
RGF MULT_PO : OP ;

 ENTREE
    LOAD_DACC : FRONT ;
    SHIFT : FRONT ;
    OE_DACC : BOOL ;
    RAZ : BOOL ;
    LOAD_MQ : FRONT ;
    OE_MQ : BOOL ;
    LOAD_B : FRONT ;
    DECR_CPTR : FRONT ;
    LOAD_CPTR : FRONT ;

 SORTIE
    MQ0 : [0..1] ;
    Z_CPTR : BOOL ;

 BIDIR
    BUS : [0..15] ;

 VARINT
    BIT0_DACC : [0..1] ;
    S_DACC : [0..15] ;
    S_ADD : [0..31] ;
    S_B : [0..15] ;

 RGF
    REG_GEN (E1, E2 : FRONT ; E3, E4 : BOOL ; E5 : [0..31] ;
             S1, S2 : [0..15] ; S3 : [0..1]) ;
    REG_MQ (E1, E2 : FRONT ; E3, E4 : BOOL ; E5 : [0..1] ;
            S : [0..1] ; BD : [0..15]) ;
    ADD_GEN (E1, E2 : [0..15] ; S : [0..31]) ;
    REG_SIMPLE (E1 : [0..15] ; E2 : FRONT ; S : [0..15]) ;
    CPTR_GEN (E1, E2 : FRONT ; S : BOOL) ;

 RCONST
    DACC : REG_GEN (LOAD_DACC, SHIFT, OE_DACC, RAZ,
                    S_ADD, BUS, S_DACC, BIT0_DACC) ;
    MQ : REG_MQ (LOAD_MQ, SHIFT, OE_MQ, RAZ, BIT0_DACC, MQ0, BUS) ;
    ADD : ADD_GEN (S_DACC, S_B, S_ADD) ;
    B : REG_SIMPLE (BUS, LOAD_B, S_B) ;
    CPTR : CPTR_GEN (LOAD_CPTR, DECR_CPTR, Z_CPTR) ;

FIN MULT_PO
```

Figure 29 : partie opérative



CADOC SYSTEM
Circuit : mults



CADOC SYSTEM
Circuit : mults

II.8.2 - DEUXIEME MODELISATION

Dans cette deuxième modélisation, nous allons inclure les actions réalisées par les éléments de la partie opérative au niveau de la partie contrôle ; on va, par exemple, remplacer l'émission de la commande de chargement du registre CPTR à sa valeur initiale (LOAD_CPTR := M) par l'affectation d'une variable interne représentant ce registre (CPTR := 4).
Cette transformation revient à transférer l'interprétation des commandes émises depuis la partie opérative dans la partie contrôle : "élimination" de la partie opérative.
La deuxième modélisation (RGF MULTB) obtenue est donnée page suivante (figure 30).
Les résultats de simulation de cette description avec les mêmes stimuli de départ que précedemment sont renvoyés en annexe.

Remarque importante

Au niveau des ports d'E/S, les résultats des simulations de cette modélisation (MULTB) sont **identiques** à ceux obtenus lors de la simulation de la première modélisation (MULTS), les temps de calcul (approximations VAX-VMS) étant respectivement de 0,51s pour MULTB et de 3,72s pour MULTS.
Ces résultats sont donc une confirmation de l'efficacité des modèles abstraits et du fait que modèle abstrait n'est pas synonyme de perte de précision.

```
RGF MULTB (BUS,MULT,ENDMULT,CK) : OP ;

 TYPE
   INTEGER_3 = [0..7] ;
   INTEGER_4 = [0..15] ;
   INTEGER_5 = [0..31] ;

 ENTREE
   CK : FRONT ;
   RSTMULT : FRONT ;
   MULT : BOOL ;

 SORTIE
   ENDMULT : BOOL ;

 BIDIR
   BUS : INTEGER_4 ;

 VARINT
   CPTR : INTEGER_3 ;
   B, MQ : INTEGER_4 ;
   D_ACC : INTEGER_5 ;

 FONCTION
 ACTION
   PI : ;
   P0 : BUS := Z // ENDMULT := F ;
   P1 : D_ACC := 0 //
        CPTR := [(4,FM(CK))] //
        MQ := [(BUS,FM(CK))] ;
   P2 : B := [(BUS,FM(CK))] ;
   P3 : D_ACC := [(D_ACC + B,FM(CK))] ;
   P4 : D_ACC := [(D_ACC DIV 2, FM(CK))] //
        MQ := [((D_ACC MOD 2) * 2 ** (N-1) + MQ DIV 2, FM (CK))] //
        CPTR := [(CPTR -1 , FM (CK))] ;
   P5 : ENDMULT := V ;
   P6 : BUS := [(MQ,0)(Z,FM(CK))] // ENDMULT := F ;
   P7 : BUS := [(D_ACC MOD 16, 0)(Z,FM(CK))] ;

 GRAPHE
   T0  : PI - P0 : V : FM (RSTMULT) ;
   T1  : P0 - P1 : MULT : FD (CK) ;
   T2  : P1 - P2 : MULT : FD (CK) ;
   T3  : P2 - P3 : ((MQ MOD 2) = 1 ) : FD (CK) ;
   T31 : P2 - P4 : ((MQ MOD 2) <> 1) : FD (CK) ;
   T4  : P3 - P4 : V : FD (CK) ;
   T5  : P4 - P4 : (CPTR <> 0) ET ((MQ MOD 2) <> 1) : FD (CK) ;
   T6  : P4 - P3 : (CPTR <> 0) ET ((MQ MOD 2) = 1) : FD (CK) ;
   T7  : P4 - P5 : (CPTR = 0) : FD (CK) ;
   T8  : P5 - P6 : MULT : FD (CK) ;
   T9  : P6 - P7 : MULT : FD (CK) ;
   T10 : P7 - P0 : MULT : FD (CK) ;

 INIT PI ;

FIN MULTB
```

Figure 30 : deuxième modélisation

II.8.3 - TROISIEME MODELISATION

La deuxième modélisation a consisté à "éliminer" la partie opérative de la description, la dernière modélisation va consister à éliminer la partie contrôle : pour cela nous allons remplacer les phases du calcul proprement dit gérées par la partie contrôle par un algorithme qui, à partir de la valeur en entrée, va calculer

- la date de positionnement des sorties en fonction de la durée d'un cycle d'horloge et du nombre de bits à 1 du multiplieur,
- la valeur de la sortie du multiplieur.

Cette troisième modélisation est donnée page suivante (figure 31).
Les résultats de simulation sont identiques à ceux obtenus précedemment ; le temps de simulation est de 0,45s.

Remarques

* Le gain au niveau temps de simulation entre la deuxième et la troisième modélisation est peu important : cela est explicable par le fait que les protocoles de chargement des valeurs initiales et d'émission des valeurs calculées sont identiques dans les deux modélisations et que le calcul proprement dit n'est pas trés complexe. La différence serait plus sensible pour des circuits à protocoles plus courts et calculs plus importants.

* Il faut bien être conscient que cette dernière modélisation, même si elle est la plus efficace au niveau de la simulation est aussi la plus difficile à obtenir : elle suppose :

- la connaissance de la structure fine du circuit (ses modélisations moins abstraites),
- l'écriture d'algorithmes permettant une évaluation assez précise des temps de réponse du circuit, ces algorithmes pouvant être difficiles à écrire dans le cas de circuits complexes.

```
RGF MULTB2 (BUS,MULT,ENDMULT,CK) : OP ;

 CONST
   FREQUENCE = 6 ; / DUREE D'UN CYCLE D'HORLOGE /

 TYPE
   INTEGER_3 = [0..7] ;
   INTEGER_4 = [0..15] ;
   INTEGER_5 = [0..31] ;

 ENTREE
   CK : FRONT ;
   RSTMULT : FRONT ;
   MULT : BOOL ;

 SORTIE
   ENDMULT : BOOL ;

 BIDIR
   BUS : INTEGER_4 ;

 VARINT
   CPTR : INTEGER_3 ;
   B, MQ : INTEGER_4 ;
   D_ACC : INTEGER_5 ;

 VAR
   AUX, DUREE, I, J : ENTIER;

 FONCTION
 ACTION
   PI : ;
   P0 : BUS := Z // ENDMULT := F ;
   P1 : MQE:= [(BUS,FM(CK))] //
        DUREE :=            / CALCUL DE LA DUREE EFFECTIVE DU CALCUL /
             DEBUT
               AUX := BUS ;
               I := 0 ;
               J := 1 ;
               TANTQUE J <> 4 FAIRE  / CALCUL DU NOMBRE DE BITS A 1 DE MQ /
                 SI AUX MOD 2 = 1 ALORS I := I + 1 FINSI ;
                 AUX := AUX DIV 2 ;
                 J := J + 1 ;
               FFAIRE ;
               RETOUR (5 + I) * FREQUENCE ; / EN FAIT 1 + 2*I + (4-I) /
             FIN ;
   P2 : B := [(BUS,FM(CK))] ;
   P3 : ENDMULT := [(T,0)(F,6)] //
        BUS := [((MQ*B) MOD 16, 6) (Z, 9) ((MQ*B) DIV 16, 12) (Z, 15)] ;

 GRAPHE
   T0 : PI - P0 : V : FM (RSTMULT) ;
   T1 : P0 - P1 : MULT : FD (CK) ;
   T2 : P1 - P2 : MULT : FD (CK) ;
   T3 : P2 - P3 : MULT ET TEMPO (P2,DUREE) ;
   T4 : P3 - P0 : MULT ET TEMPO (P3, 2 * FREQUENCE) ;

 INIT PI ;
FIN MULTB2
```

Figure 31 : troisième modélisation

III - LE SYSTEME CADOC

Nous allons distinguer trois grandes classes d'outils développés autour du langage CADOC.LD :

- les outils d'exploitation immédiate du langage : compilateur, éditeur de liens, simulateur,
- les outils développés en fonction du langage : aide au test, synthèse de parties contrôle,
- les outils interfaçables avec le système CADOC.

III.1 - Outils d'exploitation du langage CADOC.LD

L'organisation générale du système CADOC est donnée figure 32.

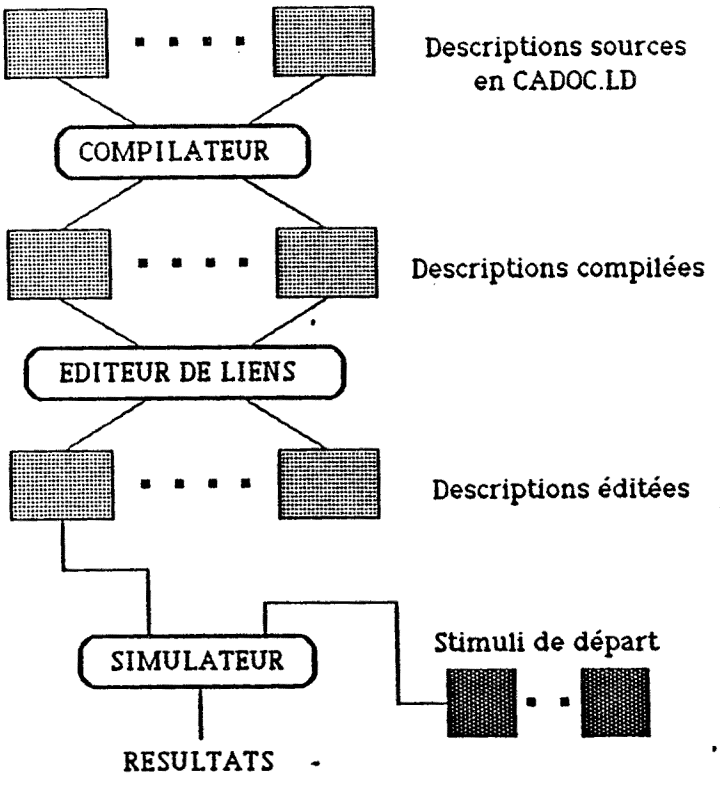


figure 32 : noyau du système CADOC

Dans l'état actuel le système complet représente 30 000 lignes de Pascal, découpées en 26 modules, ce qui correspond sous VAX VMS à un module exécutable de 1500 blocs soit approximativement 750 k octets.

III.1 - Compilateur

A partir d'un fichier source contenant la description du circuit en CADOC.LD, un compilateur génère une structure compilée et effectue les vérifications syntaxiques et sémantiques classiques.
Les modules définissant le compilateur sont au nombre de 13.

III.2 - Editeur de liens

A partir d'un ensemble de structures intermédiaires générées par le compilateur, l'éditeur de liens ([TIA 85]) génère des structures dynamiques directement exploitables par le simulateur. Cette structure étant une description "à plat" des différents modules apparaissant dans la ressource père ; ce mécanisme a été choisi préférentiellement à un mécanisme d'instantiation dynamique des descriptions des sous modules car à la différence des procédures appelées dans un langage classique, les sous modules définissant un circuit sont actifs de manière permanente.

III.3 - Simulateur

A partir
- d'un description éditée d'un circuit,
- du temps maximum de simulation,
- des valeurs des stimuli d'entrée (saisie possible par fichier)

le simulateur effectue dans un premier temps l'allocation de la mémoire nécessaire puis assure l'évolution de la description en fonction du temps.

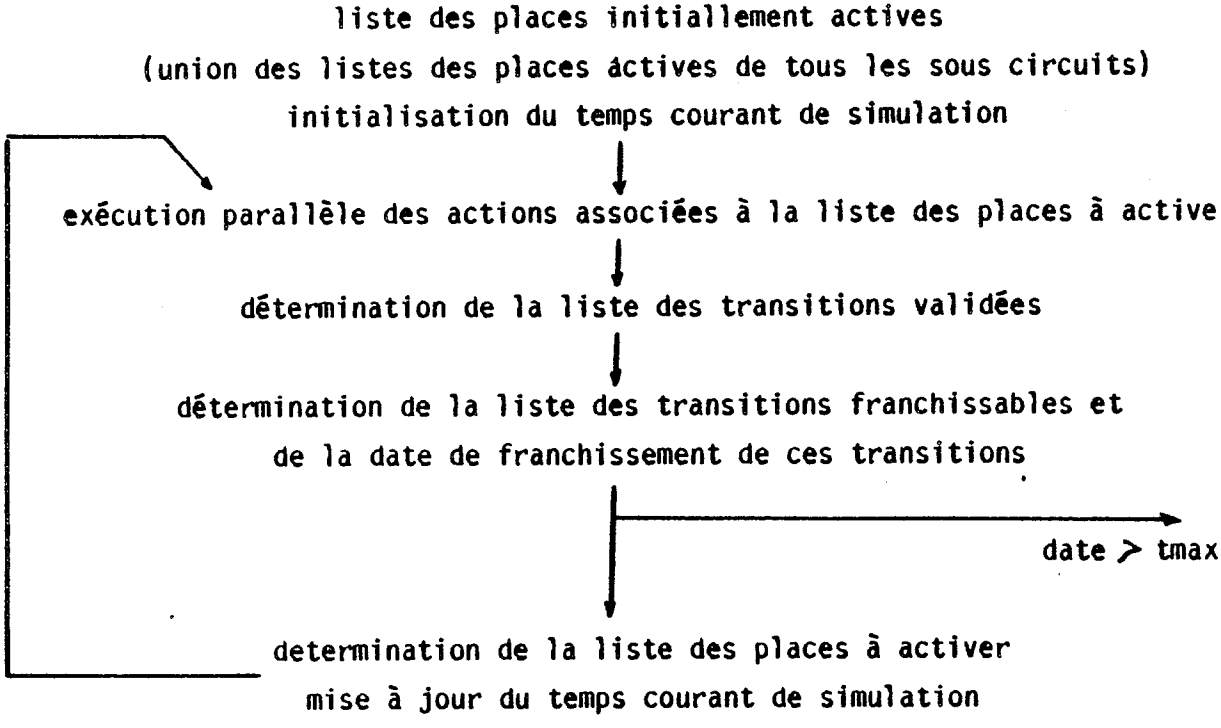
Le simulateur est composé d'un ensemble de 5 modules Pascal ; dans ces modules figure le **module arithmétique** permettant de travailler en entier multiple précision de manière transparente à l'utilisateur.

i) base de la simulation

A partir des descriptions soit éditées, soit compilées des circuits feuilles de la hiérarchie de description de la ressource père, l'éditeur de liens génére une description "à plat" ainsi qu'une liste d'opérations à effectuer en début de simulation : affectation des valeurs effectives des paramètres, résolution des connectiques implicites et explicites. Ces opérations sont réalisées lors d'une demande de simulation pour laquelle l'utilisateur spécifie le nom de la ressource générique à simuler et les valeurs des paramètres éventuels ; on peut noter que les noms des ports effectifs de la ressource père seront ses noms formels.

ii) principe de la simulation

Schématiquement, le mécanisme de simulation est le suivant :



Cet algorithme a permis de valider le langage sur des exemples de différentes complexités mais est une version prototype susceptible d'améliorations. Une des ces améliorations envisagée est l'introduction d'un mécanisme du type guidage de la simulation par événements au niveau de la détermination de la liste des transitions franchissables à partir de la liste des transitions validées ; dans l'état actuel du système, toutes les réceptivités associées aux transitions de la liste des transitions validées sont recalculées à chaque boucle du programme principal. Par incorporation d'un mécanisme événementiel, on espère diminuer le nombre de réceptivités à évaluer ; les algorithmes envisagés ne seront pas développés ici mais notons simplement que les mécanismes utilisés dans les simulateurs logiques classiques sont peu applicables dans le cas de la simulation fonctionnelle, une ressource fonctionnelle ayant un comportement dans le temps qui lui est propre ([LEI 81]) et les instants de variation de ses ports de sortie n'étant pas calculables de manière simple à partir des instants de variations des entrées comme cela est le cas en simulation logique ou logico-fonctionnelle classique.

iii) Résultats de simulation

Actuellement, le simulateur fournit les résultats suivants :

- Une trace chronologique complète de la simulation donnant la liste de tous les événements survenus au cours de la simulation. Cette trace nécessite dans certains cas, l'appel à des procédures de tri sur fichier afin de reclasser les affectations effectuées sous forme de chronogrammes et d'éliminer les événements résultants de ces affectations qui ont été détruits en cours de simulation (doubles affectations),

- Une trace exhaustive de l'histoire des variables non algorithmiques,
- Une trace de l'histoire des places (ensemble des places actives au cours du temps),
- Une trace graphique sur un ensemble de variables défini en fin de simulation.

On trouvera en annexe un exemple de trace illustrant toutes les variations possibles sur une variable booléenne, ainsi que d'autres exemples de résultats de simulation.

III.2 - Outils développés autour du langage

Actuellement, deux types d'application en liaison directe avec le langage CADOC.LD sont développées :

- test fonctionnel,
- génération de parties contrôle.

III.2.1 - TEST FONCTIONNEL

L'étude du test fonctionnel en liaison avec le système CADOC faisant l'objet d'une thèse conjointe à celle-ci ([RAR 85]), nous ne développerons donc pas ce chapitre.

Notons simplement que la méthode proposée est adaptée au problème du test de circuits complexes et utilise
- le langage CADOC.LD comme base de spécification des circuits,
- des mécanismes d'exécution symbolique temporisée,
- des méthodes dérivées de méthodes utilisées en intelligence artificielle pour résoudre les problèmes de propagation et de consistence lors de la génération des vecteurs de test

III.2.1 - GENERATION DE PARTIES CONTROLE

Le point de départ des outils de synthèse automatique de parties contrôles est une description de la partie contrôle considérée, donnée sous forme d'un graphe semblable à ceux définis dans le langage CADOC.LD (avec des restrictions sur les constructions utilisables). A partir de cette description on effectue en parallèle le codage des états du graphe et la minimisation globale des expressions définissant les variables de sortie et les variables internes ; cette minimisation prenant en compte des contraintes liées au type de la réalisation (portes disponibles, contraintes de sortances). Ces équations minimisées servent alors de base aux programmes de génération des schémas des masques correspondants à la réalisation choisie : PLA ou logique aléatoire (en vue d'une implantation sur réseaux prédiffusés).

III.3 - Outils interfaçables avec le système CADOC

Deux extensions sont proposées : la première en direction du système CATA, système d'analyse de testablité de circuits ou de cartes, la deuxième en direction du sytème GAPT de génération automatique de programmes de test pour microprocesseurs.
Il est hors de question de présenter en détail ces deux systèmes ; nous nous bornerons à indiquer leurs principales caractéristiques et les objectifs ou résultats attendus de leur interfaçage avec le système CADOC.

III.3.1 INTERFACAGE CATA/CADOC

i) Le système CATA

Le système CATA (Computer Aided Test Analysis, [ROB 83a], [ROB 84]) est un système d'analyse de testabilité de systèmes informatiques de tailles variables (cartes logiques, sous ensembles fonctionnels, systèmes entiers) ; ces systèmes étant représentés sous des formes variées (logique, fonctionnelle, structurelle), il est nécessaire dans un premier temps d'en donner une modélisation précise.

Une modélisation CATA est basée sur un graphe biparti orienté dont les deux types de noeuds sont :
- les places avec les trois classes suivantes : les places sources (générateurs d'information : ports d'entrée du système par exemple), les places puits (récepteurs d'information : ports de sortie du système par exemple) et les places intermédiaires (représentant les entités matérielles réalisant des fonctions élémentaires),
- les transitions qui modélisent la distribution de l'information au travers du système (on distingue quatre classes de transitions : transitions ET / OU en entrée ou en sortie des places).

Les arcs représentent les voies de transfert de l'information dans le système.

A partir d'une telle modélisation, le système CATA calcule alors les écoulements (chemins élémentaires du graphe) qui sont les activations élémentaires du système étudié. Ces écoulements permettent ensuite d'étudier la testabilité du système et de proposer soit des statégies de test permettant de détecter puis localiser des erreurs dans le système, soit l'implantation de points de test jugés nécessaires;

ii) Objectifs de l'interfaçage

L'objectif visé est de générer automatiquement à partir d'une description d'un circuit dans le langage CADOC une description exploitable par le système CATA (les modélisations CATA étant actuellement réalisées de façon manuelle). L'ensemble des manipulations permettant cette génération s'appuie sur la construction du graphe de dépendance des variables figurant dans la description CADOC ; le graphe obtenu étant ensuite simplifié en vue de son exploitation par le système CATA.

Exemple

Considérons le circuit (registre à décalage sur 8 bits, entrée série) dont le brochage externe est donné figure 33a et le schéma fonctionnel figure 33b ; une description approximative (à but d'illustration uniquement) est celle donnée figure 33c. A partir de cette description on génére le graphe de dépendance (figure 33d), aprés simplification (suppression des variables internes et de l'horloge), on obtient la description compatible avec le système CATA donnée figure 33e.

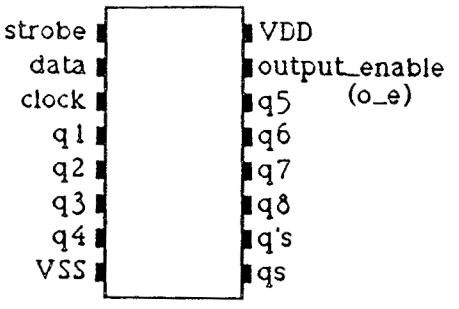


figure 33a
brochage extérieur du HCC/HCF 4094B (GGS)

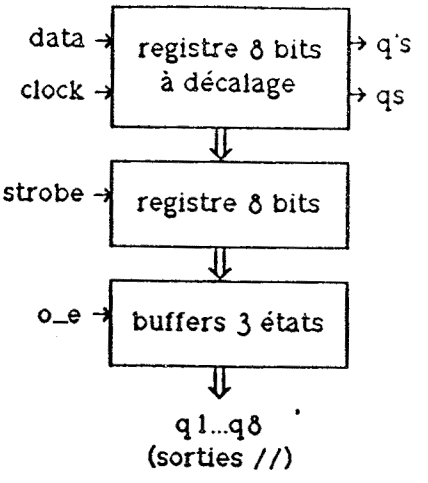


figure 33b
schéma fonctionnel

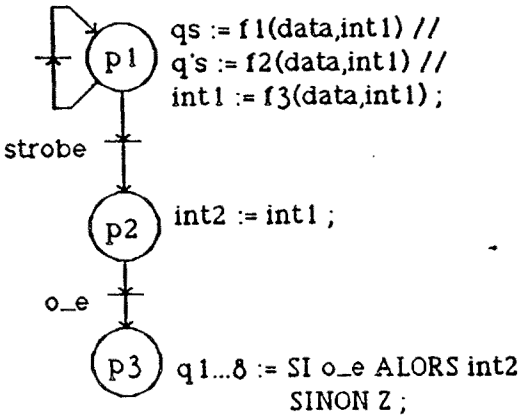


Figure 33c
description CADOC

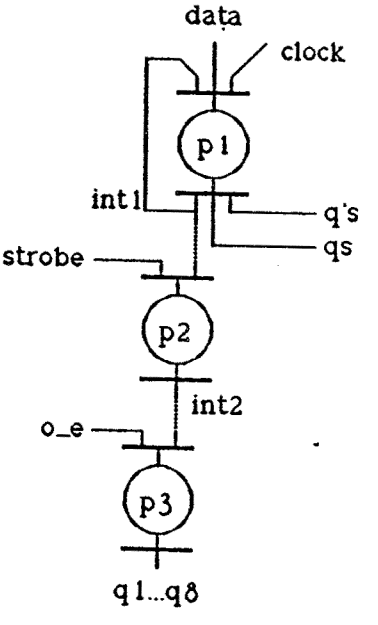


Figure 33d
graphe de dépendance

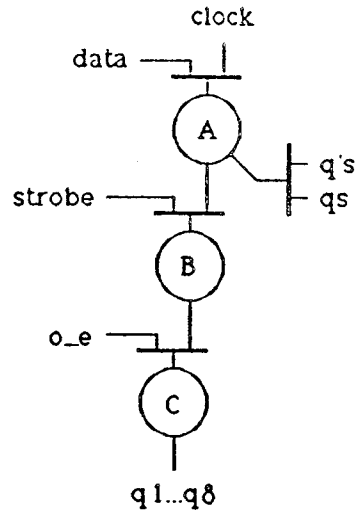


Figure 33e
description CATA

III.3.2 - INTERFACAGE GAPT/CADOC

i) Le système GAPT

Le système GAPT ([BEL 82], [BEL 83]) s'intéresse au test comportemental de microprocesseurs, le terme comportemental signifiant ici que le test sera généré non à partir de la structure interne du microprocesseur mais à partir des fonctions qu'il réalise ; la description de ces fonctions étant obtenue à partir d'un manuel utilisateur fourni par le constructeur.
Le système complet (figure 34) est composé d'un logiciel de génération automatique de programmes de test (logiciel GAPT) et d'un testeur adapté à ce type de test (testeur TEMAC).

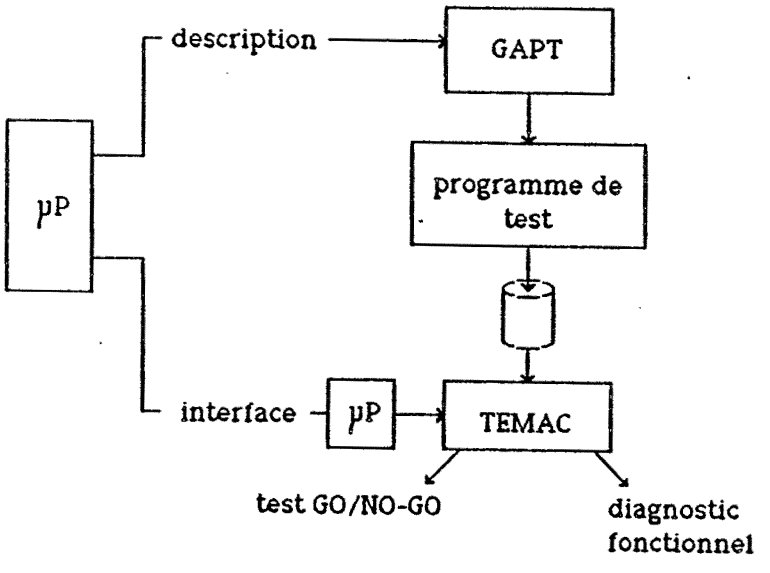


Figure 34 : chaine de test GAPT

Le logiciel GAPT génère automatiquement le programme de test d'un microprocesseur (en langage assembleur) à partir de sa description "haut niveau" en langage GAPT ; ce langage est un langage de description fonctionnelle normalisée de microprocesseurs et permet de décrire le jeu d'instructions d'un microprocesseur de manière compacte.
Le code assembleur généré comprend le code correspondant à l'activation des familles d'instructions décrites ainsi que le code nécessaire pour assurer la cohérence des instructions (initialisations) et l'observation des résultats.

ii) Objectifs de l'interfaçage

Dans l'état actuel du système CADOC, il est parfaitement possible de définir des modèles de microprocesseurs mais l'interfaçage avec le système GAPT nous semble important pour les raisons suivantes :
- la premiere raison est que cet interfaçage nous permettra de profiter du caractère systématique d'une description de microprocesseur en GAPT.
- la deuxième raison est qu'il sera ensuite possible de simuler les programmes de test générés par GAPT, ce qui permettra d'assurer une comparaison entre les sorties du microprocesseur sous test et des résultats de simulation CADOC (alors qu'actuellement, les résultats du test sont comparés avec ceux d'un microprocesseur supposé bon).

Les points importants de cette étude vont être les suivants :
- introduction d'une sémantique d'exécution dans les descriptions GAPT (définition d'une sémantique standard ?),
- introduction de spécifications temporelles (nombre de cycles pris pour exécuter une instruction, ...),
- définition d'un mécanisme de traduction entre le formalisme de GAPT et celui de CADOC.

IV - CONCLUSION DE LA DEUXIEME PARTIE

Nous avons vu que le système CADOC peut être utilisé soit de façon autonome, soit en liaison avec des sytèmes déja existants. Il nous faut remarquer ici que tous les systèmes étudiés font référence à l'aspect fonctionnel ou structurel des circuits ; l'aspect topologique (caractéristiques spatiales des circuits) étant systématiquement ignoré.

La collaboration avec d'autres systèmes (génération de plan de masse, routage,...) nécessite l'introduction d'informations topologiques ; ces informations ne devant pas faire partie des descriptions fonctionnelles des circuits mais plutôt faire partie de l'ensemble des informations associées à un circuit donné ; ces informations pouvant être contenues dans une structure du type base de données spécialisée CAO permettant de faire des vérification de cohérence entre données associées à un même circuit.

Les informations associées à un circuit, que l'on pourrait stocker dans une telle base de données seraient par exemple :

- des descriptions fonctionnelles (classées par niveaux d'abstraction),
- des informations structurelles (hiérarchie,...),
- des informations topologiques (surfaces, dimensions, ...),
- des descriptions destinées à d'autres outils (CATA, ...).

TROISIEME PARTIE

MODELISATION ET SIMULATION DE DISPOSITIFS DE BAS NIVEAU
CIRCUITERIE MOS

IV - SIMULATION FONCTIONNELLE

IV.1 - CAP/DSDL

IV.2 - CADOC.LD : PROPOSITIONS ET EXTENSIONS

IV.2.1 - Approche multiforce

IV.2.2 - Approche séquentielle

IV.2.3 - Extensions du langage : application à la description des circuits précaractérisés et prédiffusés

V - AUTRE APPROCHE

VI - CONCLUSION DE LA TROISIEME PARTIE

I - INTRODUCTION

Etant données les définitions des domaines fonctionnel et structurel, on peut dans un premier temps considérer que l'on a une correspondance entre ces deux domaines pour un même niveau d'abstraction. En d'autres termes, que les objets du domaine structurel sont regroupables de manière disjointe suivant les niveaux : au niveau circuit, on traitera des transistors et des capacités, au niveau logique, les éléments de base seront des portes (non vues comme assemblage de transistors).
Dans un deuxième temps, on s'aperçoit que ces regroupements par niveaux ne sont pas disjoints et qu'il existe quelques dispositifs matériels communs à tous les niveaux : cela est en particulier le cas des transistors qui au niveau circuit jouent le rôle de dispositifs analogiques et qui, au niveau logique doivent être décrits comme élément logique (dans le cas où ils sont utilisés comme porte de transfert bidirectionnelle). En ce qui concerne ces dispositifs, il est évident que le modèle abstrait doit être déduit du modèle fin.

Dans le cadre classique, on peut considérer la classification suivante :

niveau	type de descriptions	sous niveaux
électrique continu	équations des dispositifs (différentielles, non linéaires, linéaires)	suivant les méthodes de résolution utilisees (timing,...)
électrique discret	modèles de type interrupteur	
logique (discret)	interconnexion de portes	

Dans une première partie nous allons rapidement présenter les méthodes d'analyse des circuits au niveau électrique continu ; cette analyse s'appuiera sur une technologie prépondérante pour la conception de circuits à très haute densité d'intégration : la technologie MOS (Métal Oxyde Semiconducteur) mais les principes restent valables quel que soient les technologies employées. Nous nous intéresserons ensuite plus particulièrement à la modélisation de type interrupteur pour les structures MOS. Le paragraphe correspondant au niveau logique présentera un certain nombre de propositions permettant la prise en compte des problèmes spécifiques du type description de portes de transfert, de la logique trois_états, ...

Dans une deuxième partie, nous indiquerons comment un langage fonctionnel peut résoudre ces problèmes et nous terminerons avec des propositions d'extensions du langage CADOC afin d'augmenter son champ d'application, en particulier dans le domaine des descriptions de bas niveau.

II - TECHNOLOGIE MOS

La figure 1 donne la structure d'un transistor MOS à canal n (n MOSFET : nMOS Field Effect Transistor).

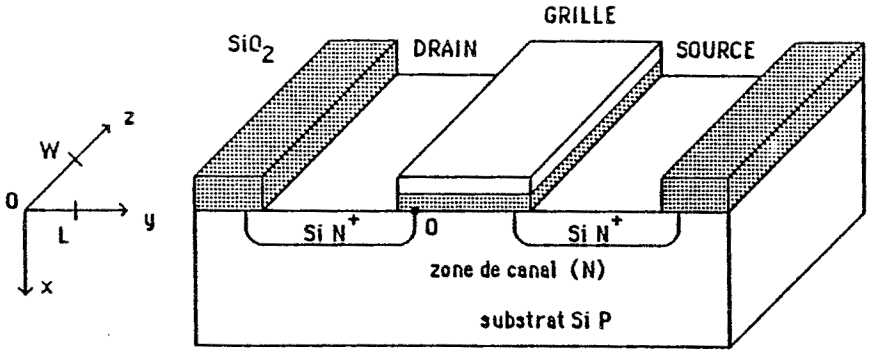


Figure 1 : transistor MOS

Schématiquement un transistor MOS est un dispositif à 3 ports, le potentiel appliqué sur la grille assurant ou non la conduction entre la source et le drain. Avant toute étude plus détaillée, il est important de noter que par nature, un transistor MOS est bidirectionnel entre les deux ports qui ne sont pas la grille. Ce ne sont que des considérations d'habitude sur les polarisations qui permettent l'attribution des termes source et drain à ces deux ports (usuellement on appelle source, le port qui est source des porteurs, ce qui entraîne pour des transistors nMOS que le potentiel de la source est inférieur à celui du drain).

Une étude détaillée du fonctionnement physique fin des dispositifs MOS est hors de propos ici, des informations précises peuvent être trouvée dans [KIT 83], [SAM 82] et [CAST 82].

De manière simplifiée un transistor MOS possède 3 modes de fonctionnement : mode linéaire, mode non saturé et mode saturé, ces trois modes correspondent à une tension de grille supérieure à une tension de seuil VT, pour les tensions VG < VT, le transistor est bloqué.

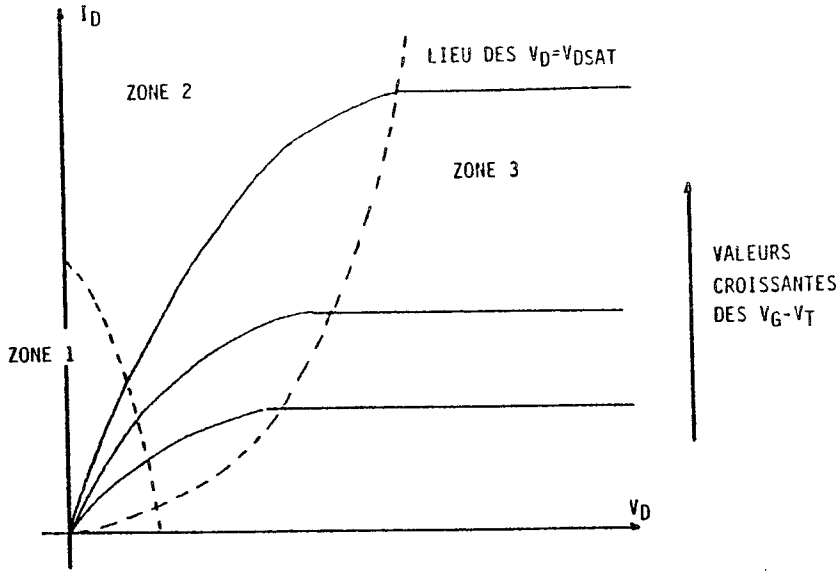
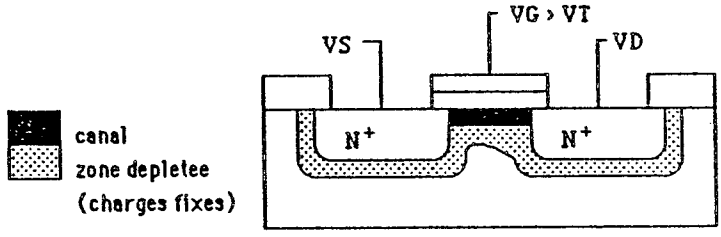
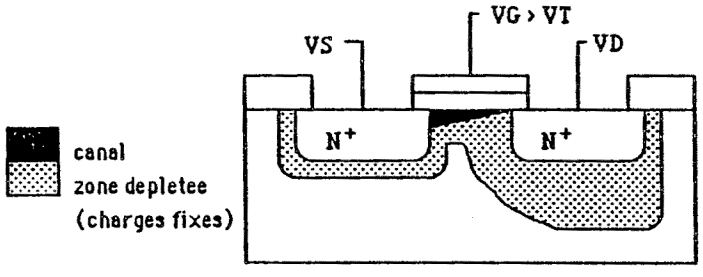


Figure 2 : caractéristique $V_D = f(I_D)$

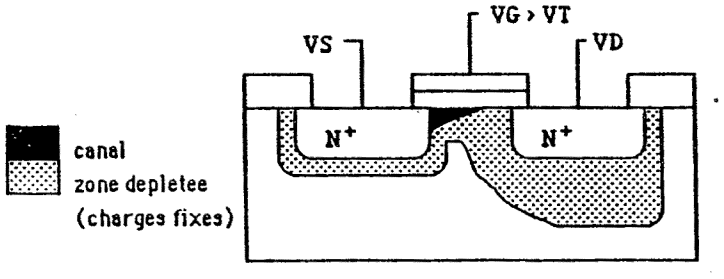
• <u>Mode linéaire</u> : pour les valeurs faibles de V_D, le canal se comporte comme une résistance ohmique dont la valeur dépend de V_G. Ce mode correspond à la zone I de la courbe donnée figure 2.



• Mode non saturé : lorsque V_D augmente jusqu'à une valeur V_{DSAT}, on observe une augmentation de la résistance du canal et une diminution de la pente de I_D. Pour $V_D = V_{DSAT}$, l'épaisseur de la couche d'invertion (canal) en y = L est nulle) : pincement du canal, (zone II).
Dans cette zone, I_D est une fonction complexe de V_D, (fonction quadratique en première approximation).



• Mode saturé : pour V_D plus grand que V_{DSAT}, le pincement du canal a lieu de plus en plus près de la source. I_D devient approximativement indépendant de V_D ; cette zone (zone de saturation) est la zone III de la figure 2.



Rappelons une dernière fois ici, que ces modes de fonctionnement ne sont que des approximations, les équations régissant le fonctionnement étant plus complexes.

Le calcul de I_D se fait à partir de la formule :

$$I_D = \mu_N \frac{W}{L_{eff}} \int_{V=V_S(y=0)}^{V_D(y=L)} - Q_N(y)\, dV(y)$$

$Q_N(y)$ représente la charge contenue dans la portion (y,y+dy) du canal

μ_n est la mobilité des porteurs (n → électrons, p → trous).

L_{eff} représente la longueur du canal.

W représente la largeur du canal.

A partir de cette intégrale on obtient des équations $I_D=f(V_{GS},V_{DS},V_{SB})$ plus ou moins complexes en fonction des hypothèses faites (charge du substrat Q_B fonction de V_{DS}, prise en compte de la longueur effective du canal, problèmes des canaux courts, μ_N non constant, inversion faible...).

Des études sont faites pour proposer des modèles de dispositifs qui soient utilisables par des calculateurs ; des exemples seront trouvés dans [JEN 73], [WHI 80], [LIN 73], [HSP 83] et [ENG 83].

Les points importants à noter sont :

- notion de modèle représentant le transistor de manière plus ou moins précise selon les besoins. A titre de remarque, le circuit électrique équivalent à un TMOS est donné en figure 3 (modèle de Schichman et Hodges) et des exemples d'équations (tirés de [HSP 83]) sont donnés en figure 4.

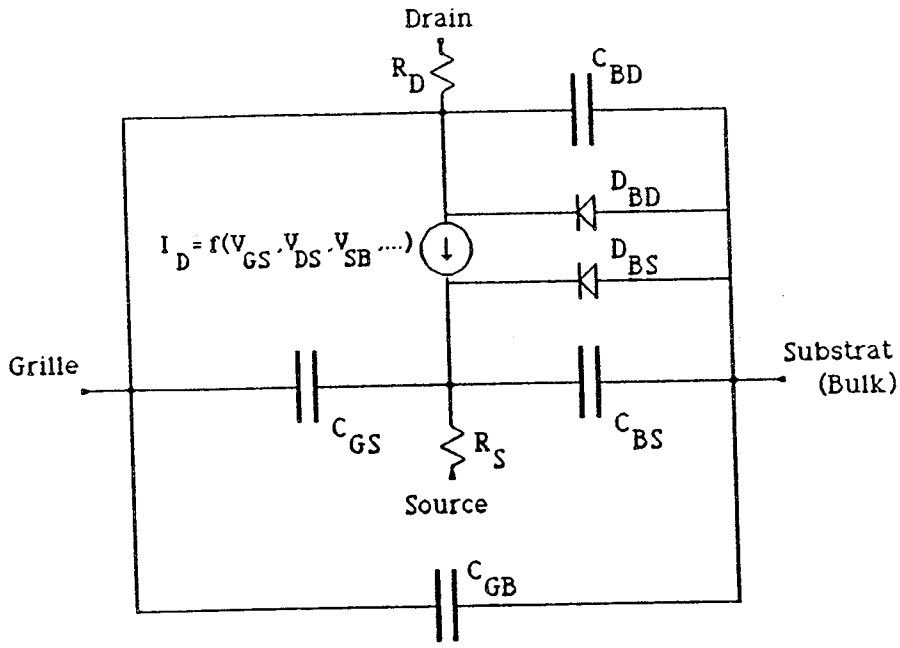


Figure 3 : Modèle de Schichman et Hodges

- $V_{GS} < V_T \quad I_{DS} = 0$
- $V_{GS} > V_T$ et $V_{DS} < V_{DSAT}$

$$I_D = K * (2(V_{GS}-V_T) - V_D(1+Dm))V_D$$

- $V_{GS} < V_T$ et $V_{DS} > V_{DSAT}$

$$I_D = I_{DSAT}\ (1+(V_D-V_{DSAT}) / (VE+V_{DSAT}))$$

K,Dm et VE étant des parametres dépendant de la technologie

Figure 4 : exemples d'équations (HPICE, LEVEL 2)

Remarque :
Les modèles choisis sont déterminés par un ensemble de paramètres qui doivent etre accessibles au concepteur.

- Le deuxième point à noter est que les modèles sont recherchés en fonction de leur facilité de prise en compte par des algorithmes de calcul ; on trouve ici l'opposition entre précision d'un modèle et temps de calcul.

III - SIMULATIONS CLASSIQUES

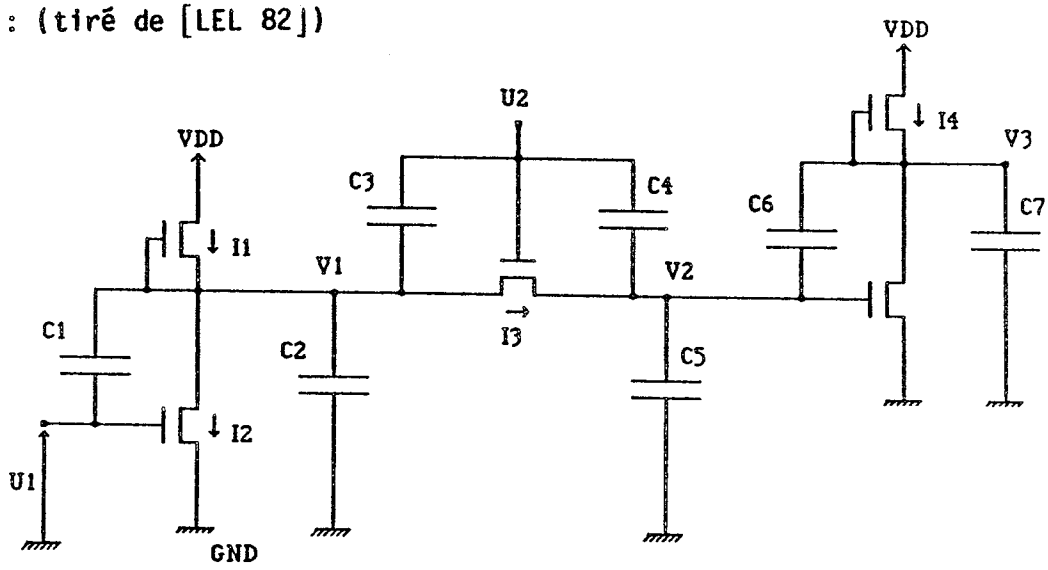
III.1 - Simulation électrique continue

Lorsque l'on considère un circuit complexe, le calcul de son évolution (sa simulation) peut être envisagé en considérant le circuit comme un tout. Etant donné un circuit composé d'éléments de base, on peut écrire l'ensemble des équations définissant son comportement approché en fonction du temps. Pour chaque noeud du circuit (point d'interconnexion) on peut écrire une équation de la forme :

$$C(v(t),u(t)).v'(t) + f(u(t),v(t)) = 0$$

avec v(t) : potentiel inconnu
u(t) : potentiel imposé (noeuds d'entrée)
C : capacité associée au noeud
f : somme des courants chargeant la capacité associée au noeud.

Exemple : (tiré de [LEL 82])



On peut écrire les équations suivantes :

noeud 1 : $(C_1 + C_2 + C_3)v'_1 - i_1(v_1) + i_2(v_1,u_1) + i_3(v_1,u_2,v_2) - C_1u'_1 - C_3u'_2 = 0$

ou : $C_1(v'_1 - u'_1) + C_2(v'_1 - 0) + C_3(v'_1 - u'_2) - i_1(v_1) + i_2(v_1,v_1) + i_3(v_1, u_2, v_2) = 0$

noeud 2 : $(C_4 + C_5 + C_6)v'_2 - C_6v'_3 - i_3(v_1,u_2,v_2) - C_4u'_2 = 0$

noeud 3 : $(C_6+C_7)v'_3 - C_6v'_2 - i_4(v_3) + i_3(v_3,u_2) = 0$

Remarques

- ces équations sont couplées entre elles
- les équations des modèles des transistors apparaissent sous la forme $i=f(v)$

  (exemple : $i_2 = f(v_1,u_1)$ notée $i_2(v_1,u_1)$).

A ces équations il faut ajouter les équations donnant l'état initial :

$$u_1(0) = u_1^0,\ u_2(0) = u_2^0,\ldots$$

De manière schématique on obtient donc un ensemble d'équations couplées de la forme :

$$f_1(u,\ u',\ i,\ t) = 0$$
$$f_2(u,\ i,\ t) = 0 \qquad \text{(système 1)}$$

Remarques

- On s'intéresse à la résolution du système (1) entre les instants t_0 (début de simulation) et t_{max} (fin de simulation).
- Le terme "électrique continu" provient du fait que l'on manipule des valeurs de courant et de potentiel supposés fonctions continues du temps.

Pour résoudre (1) on commence par réécrire les dérivations u'(t) en discrétisant le temps (cette discrétisation n'est pas assimilable à la discrétisation du temps au niveau des modèles logiques par exemple : elle résulte ici d'une méthode mathématique de résolution et non de l'apparitions d'évènements externes).

Les formules générales sont de la forme :

$$u'_n = \alpha_r\left(u_n - \sum_{k=1}^{r} \beta_{rk} u_{n-k}\right)$$

ou α_r et $(\beta_{rk})_{k=1,r}$ sont des coefficients constants

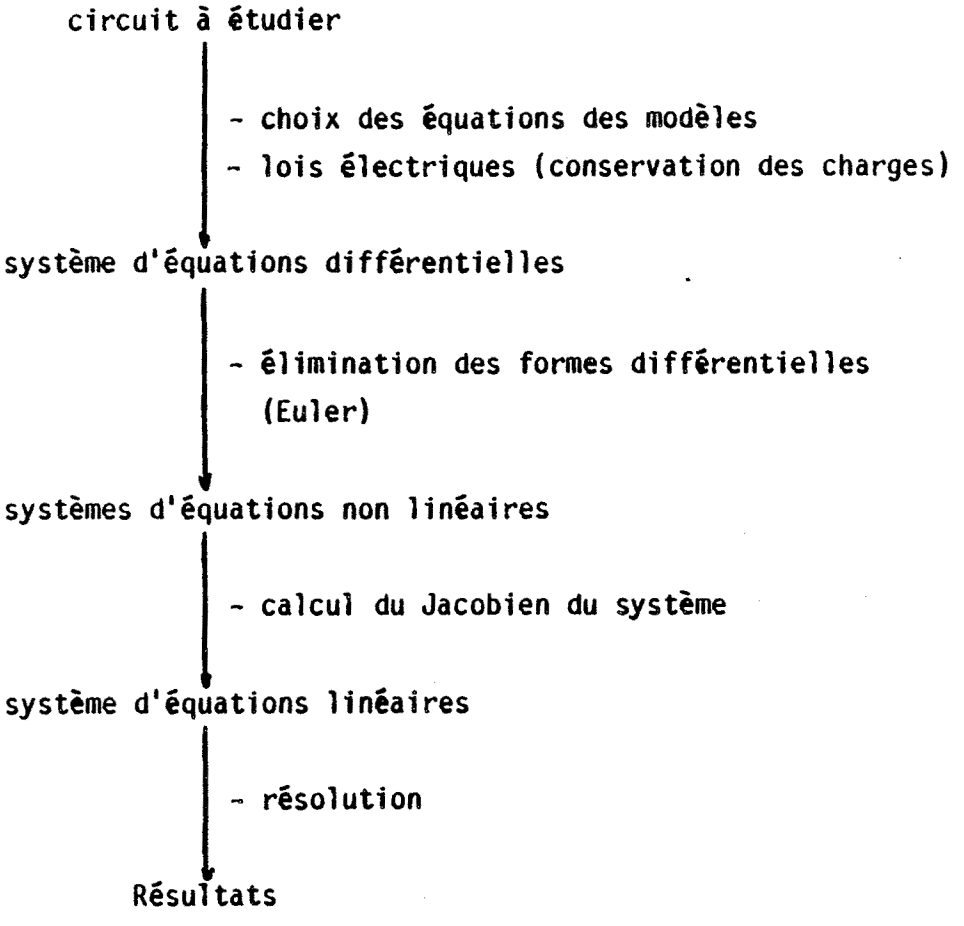
Une méthode généralement utilisée est de prendre la forme d'Euler :

$$u'_n = \frac{u_n - u_{n-1}}{h_n}$$

Cette approximation présente l'avantage d'être simple sur le plan des calculs à effectuer mais présente des problèmes de divergence. Dans le cas des circuits complexes, il peut être préférable d'utiliser des approximations d'ordre supérieur.

Après application de ces méthodes d'intégration, le système (1) devient un système d'équations non linéaires.
Ces équations sont ensuite résolues par application des méthodes du type Newton-Raphson et on retrouve le problème classique de la résolution d'un système d'équations linéaires de la forme Az = B

De manière schématisée on a donc le processus suivant :



Une fois admis ce schéma général on peut décomposer les simulateurs en 2 classes.

- simulateurs classiques : le système est traité en un seul bloc
- simulateurs non classique : décomposition du problème (relachement de certaines contraintes, relaxation).

En ce qui concerne les simulateurs classiques (SPICE2, HSPICE, ASTAP), si la précision qu'ils fournissent est nécessaire dans certains cas (ils servent usuellement de point de comparaison aux autres simulateurs), il est clair que la complexité des calculs qu'ils doivent effectuer les rendent inutilisables pour les circuits de taille importante (quelques centaines de transistors),

le temps de calcul variant quadratiquement avec le nombre d'éléments à traiter.

L'idée des simulateurs non classique résulte de la constatation que dans les circuits seul un faible pourcentage des éléments de base est actif à un instant donné et donc que seule une faible partie de ces éléments nécessite un calcul précis (avec un faible pas d'intégration).

Dans ce qui précède, nous avons vu qu'il est possible de définir 3 niveaux dans l'analyse d'un circuit : niveaux différentiel, non linéaire, linéaire.

La décomposition du système global peut s'effectuer ([HAC 81]) à chacun de ces 3 niveaux.

• Niveau linéaire

A ce niveau (résolution du système d'équations linéaires Az = b), la décomposition peut se ramener à des manipulations de matrices, manipulation permettant d'exploiter l'aspect généralement creux de la matrice A (noeuds faiblements couplés). On peut noter que le terme relaxation (ou méthode indirecte) est parfois employé pour ce niveau.

• Niveau non linéaire

Les simulateurs MOTIS ([CHA 75]) ET SPLICE ([SAL 83]) utilisent des méthodes de relaxation à ce niveau. Nous reviendrons sur ce point dans le paragraphe consacré aux simulateurs timing.

• Niveau différentiel

Dans le cas du simulateur RELAX ([LEL 82]), le système d'équations différentielles est décomposé en sous systèmes, chacun pouvant être résolu indépendemment.

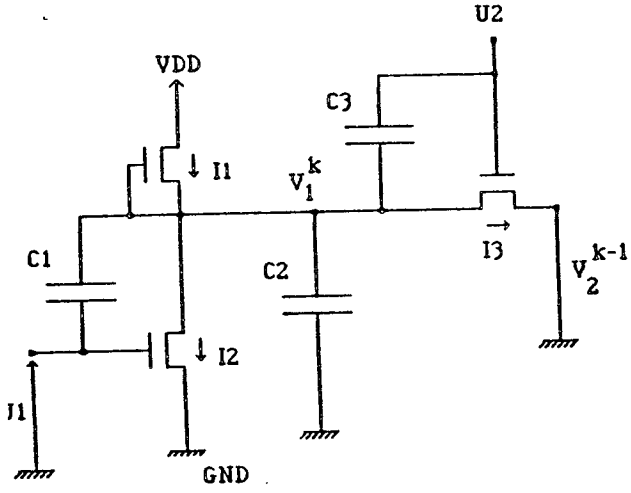
Si l'on considère le circuit précédent et le système d'équations (1), la première équation de ce système peut se réécrire en utilisant l'algorithme de relaxation donné par [LEL 82] :

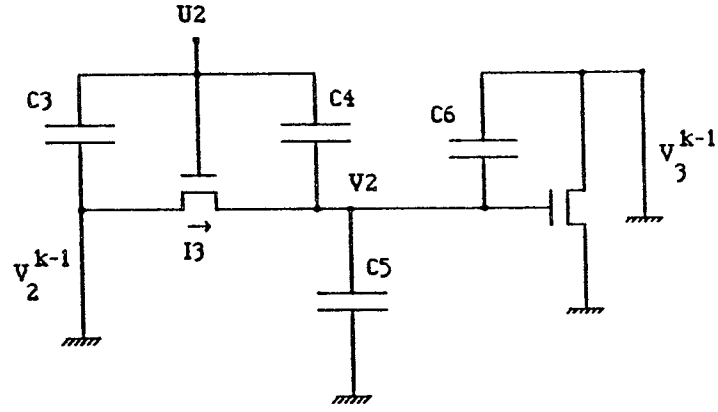
$$(C_1 + C_2 + C_3)v'^{k}_1 - i_1(v^k_1) + i_2(v^k_1, u_1) + i_3(v^k_1, u_2, v_2^{k-1}) - C_1 u'_1 - C_3 u'_2 = 0$$

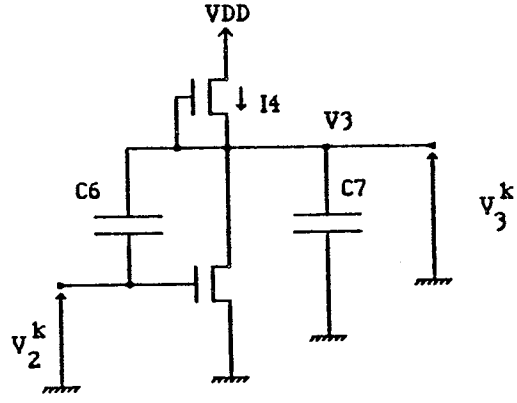
où seul v^k_1 est inconnu.
Ce type d'équation est résolu itérativement jusqu'à ce que l'on ait convergence :

$$\max_{\text{ensemble des noeuds}} \quad \max_{t \in [0, T\max]} |v^k_i(t) - v^{k-1}_i(t)| < \epsilon$$

On peut remarquer que ces équations correspondent aux circuits suivants obtenus par éclatement du circuit initial :







Un des points à noter dans RELAX est que chaque sous-circuit, (ie chaque sous système) est analysé avec son propre pas d'intégration alors qu'en analyse standard, tout le circuit est traité avec un pas unique.

Une critique à ce point étant que ce sont des considérations liées aux méthodes d'intégration et aux problèmes de stabilités qui fixent le pas d'intégration et non pas des considérations sur la précision souhaitée par le concepteur (précision qui aurait pu entrainer le choix d'un pas d'intégration moins fin dans une approche classique) ; d'autres remarques sur cette technique peuvent être trouvées dans [HAC 81] (limitation des types de circuits autorisés, problèmes des capacités flottantes (non reliées à une masse),...)

Le dernier point que nous allons noter est que le circuit étant décomposé en blocs, l'ordre de traitement de ces blocs est important ; l'ordonnancement des différents blocs étant effectué sur des critères de connectiques entre blocs, un traitement particulier étant prévu pour les boucles.
On peut établir ici une analogie entre ce type d'ordonnancement et les méthodes de simulation pilotées par évènement utilisées au niveau logique, l'ordonnancement entre les blocs étant fait suivant le flot de données en cours de simulation.

En ce qui concerne l'efficacité, une comparaison entre RELAX et SPICE 2 (niveau 1 pour les modèles des transistors MOS) semble indiquer un ordre de grandeur de différence au niveau des temps de simulation.

Un autre exemple de simulateur de même type que RELAX est ELDO ([HEN 85]). Le principe de simulation employé dans ELDO est semblable à celui de RELAX, la différence venant de l'ordonnancement des 3 boucles du calcul : boucle sur le temps, boucle sur le nombre de relaxations, boucles sur l'ensemble des noeuds. Dans RELAX ces boucles sont imbriquées dans l'ordre suivant (boucle la plus englobante en tête) : relaxation, noeuds, temps alors que dans ELDO l'ordre est temps, relaxation, noeuds.

Une des conséquences est que ELDO travaille sur un pas de discrétisation constant pour tous les noeuds. RELAX pouvant associer un pas différent à chaque noeud (le temps calcul nécessaire pour assurer la cohésion des différents temps locaux étant à prendre en compte). Un des points de

différence majeur entre RELAX et ELDO étant la mémoire nécessaire pour effectuer une analyse complète (le temps d'analyse est une fonction linéaire de la taille des circuits à traiter).

Cette différence est liée au fait que RELAX travaille sur toutes l'histoire des noeuds avant d'arriver à un résultat correct à la différence de ELDO qui utilise les valeurs de l'instant précédent uniquement, les autres valeurs pouvant être oubliées.

III.2 - Simulation Timing

Comme cela a été dit au paragraphe précédent, la simulation timing fait partie de la simulation continue niveau non linéaire ; elle se caractérise par 3 points ([GRU 83]) :

- le choix du modèle du dispositif : dans le cas de la circuiterie MOS, un modèle du type Schichman et Hodges est généralement utilisé. Les valeurs de $I_D=f(V_D)$ étant soit déterminées à partir d'équations soit obtenues par interpolation à partir de table ([CHA 75], [CHE 84]).
- le partitionnement des équations (ou partitionnement du réseau) avec la contrainte suivante : un réseau doit être suffisamment petit pour pouvoir être évalué rapidement mais suffisamment important pour que les couplages forts soient traités au niveau des sous réseaux.
- les algorithmes de simulation avec les problèmes de choix de l'environnement temporel et d'ordonnancement des sous-réeaux.

Les deux exemples types sont MOTIS et SPLICE ; des comparaisons et critiques de ces systèmes peuvent être trouvées dans [HAC 81]. Le point important à noter pour ces simulateurs est la prise en compte de descriptions mixtes dans lesquelles un circuit est décrit comme assemblage de transistors MOS et de portes logiques. Nous reviendrons sur ce mode de simulation et en particulier sur les problèmes d'interface liés à la différence des données manipulées : variations continues au niveau électrique, variations discrètes au niveau logique.

III.3 - Simulation de niveau interrupteur

Dans les modèles précédents, les grandeurs manipulées sont à domaine de variation continue ; les simulateurs de niveau interrupteur vont manipuler des grandeurs discrètes et mettre en évidence le comportement logique (digital) des dispositifs.

A ce niveau de modélisation vont entrer en concurrence deux types d'approches l'approche niveau interrupteur pure et l'approche logique étendue ; dans la première approche un circuit est décrit comme interconnexion de transistors par l'intermédiaire d'un certains nombres d'autres éléments. Dans la deuxième approche les objets initialement prévus dans le langage sont du niveau porte logique (abstraction d'un ensemble de transistors), cet ensemble d'objets étant étendu pour pouvoir prendre en compte les points liés à la technologie MOS.

Dans une modélisation de type interrupteur, la description d'un circuit est un équivalent immédiat de sa description au niveau électrique ; en particulier, toutes les connexions sont représentées (ce qui n'est pas le cas en particulier pour les alimentations au niveau logique). On peut donc considérer que ce type de simulation est un intermédiaire important entre les descriptions de type abstrait (abstraction au niveau connectique) et les descriptions proches du niveau implantation.

Dans cette partie, nous allons d'abord présenter 2 propositions pour la simulation niveau interrupteurs : MOSSIM ([BRY 81], [BRY 84]) et CSA ([HAY 82], [HAY 84a], [HAY 84b]).

III.3.1 - MOSSIM (MOS SIMULATOR)

Le simulateur MOSSIM s'intéresse uniquement à la circuiterie MOS digitale, en conséquence les circuits basés sur des propriétés analogiques des transistors (amplificateurs différentiels,...) ne sont pas correctement modélisables.

Un circuit est composé d'un ensemble de noeuds ($\{n_1,...n_n\}$) reliés par des transitors ($\{t_1,...,t_n\}$).

Un noeud est caractérisé par son état prenant les valeurs 0,1 et X (au sens de indéterminé soit par non initialisation, soit par suite d'un partage de charges généralement transitoire) avec l'ordonnancement $0 < X$, $1 < X$.

A chaque noeud est associé une taille dans l'ensemble $\{k_1,..., k_{max}\}$ avec $k_1 < k_2 \; .. < k_{max}$, cette taille indiquant la capacité relative du noeud. Un noeud d'entrée de circuit est un noeud de taille infinie.

Remarque :

De manière schématique, on peut considérer que trois tailles k_1,k_2 et k_∞ sont suffisantes pour décrire les réseaux MOS classiques, ce qui permet de retrouver les trois types de noeud d'une version préliminaire de MOSSIM ([BRY 81]) : noeud d'entrée (k_∞), pullup (k_2) et de mémorisation (k_1).

Un transistor est un dispositif à 3 ports (grille, source et drain sans distinction entre drain et source). Un transistor est d'un type donné (n,p ou d) -le type correspondant à ses caractéristiques électroniques- et joue le rôle d'un élément résistif contrôlé par la grille.

En fonction de la valeur de la grille on définit l'état du transistor :

état de la grille \ type du transistor	n	p	d
0	0	1	1
1	1	0	1
X	X	X	1

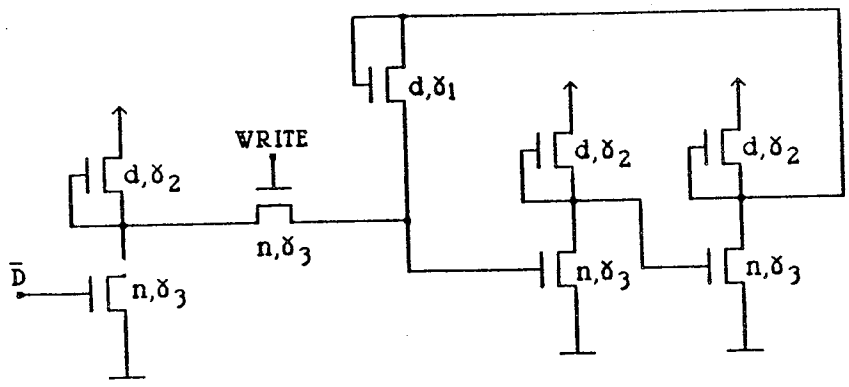
0 : bloqué

1 : conducteur

X : indéterminé

Dans le cas d'un transistor dans l'état 1, la conductance entre le drain et la source est caractérisé par un nombre γ_i avec $(\gamma_1 < \gamma_2 \ .. \ < \gamma_{max})$.

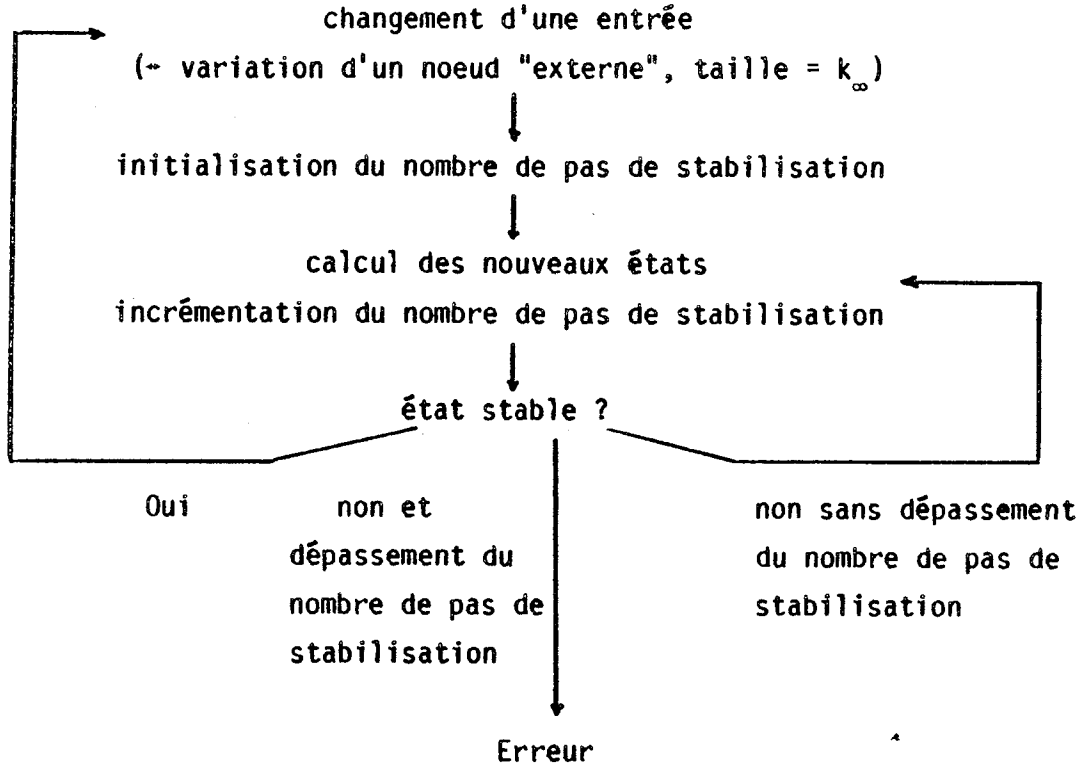
Exemple : Point mémoire statique



Principe de simulation

Le coeur du simulateur MOSSIM est un simulateur à délai unitaire, calculant l'évolution des noeuds d'un circuit en fonctions de l'état passé et des configurations des interconnexions jusqu'à stabilisation dans un état donné (ou dépassement du nombre maximal d'oscillations).

Le mécanisme d'évolution peut se schématiser par :



La supposition qui est faite est que les entrées du circuit (horloge,.) varient suffisamment lentement pour que les noeuds puissent se stabiliser entre deux variations externes.

De manière plus précise l'algorithme de simulation ([BRY 84]) est le suivant (les notations étant de type Pascal) :

- C représente la liste des noeuds subissant une variation

- type_noeud =

```
record
  taille, état, nouvel_état, q, u, d : entier ;
  trouvé, fait : boolean ;
  fan_out : ensemble_de_T
    (* ens. des transistors dont ce noeud est la grille *)
  input_connect : ensemble_de_T
    (* ens. des transistors reliant ce noeud à un noeud d'entrée *)
  memo_connect : ensemble_de_T
    (* ens. des transistors reliant ce noeud à un noeud de
       mémorisation *)
end
```

```
Type_transitor = record
record
  type, force, état : entier
  noeud1, noeud2 : type_noeud ; (* source et drain *)
end
```

Procédure effectuant un pas d'évolution

```
Procédure pas (E)  (* E = liste des noeuds perturbés *)
    début
        C := nil ;
        pour tout n ∈ E tel que non n.fait faire reponse_voisinage (C,n)
        E := nil ;
        pour tout n ∈ C faire
          debut
            n.état := n.nouvel_état ;
            pour tout t ∈ n.fan_out faire
                  si état_transistor (n.état, t.type) ≠ t.état alors
                       debut
                         t.état := état transitor (n.état, t.type) ;
                         si t.noeud1 perturbé le mettre dans C ;
                         si t.noeud2 perturbé le mettre dans C ;
                       fin ;
          fin ;
        retour (E) ;
    fin ;
```

La procédure reponse_voisinage calculant l'état suivant de l'ensemble des noeuds donné, c'est dans cette procédure que sont pris en compte les problèmes liés aux grilles dans l'état X.

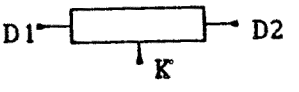
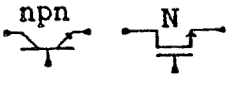
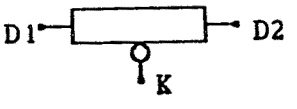
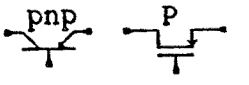
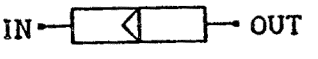
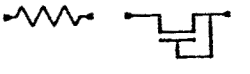
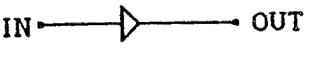
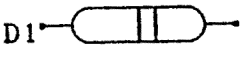
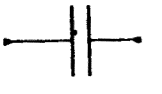
Remarques

- Une des difficultés majeure de la simulation niveau interrupteur réside dans le traitement correct des noeuds grille dans l'état X ; ce problème est lié à la non existence d'un état $\bar{X}$ ([BRE 72]).
- Le simulateur ne donne aucune information sur le comportement du réseau en fonction du temps.
- le type de simulation donné précédemment s'apparente à de la simulation logique par évènements avec délai 0.

III.3.2 - CSA (CONNECTOR SWITCH ATTENUATOR)

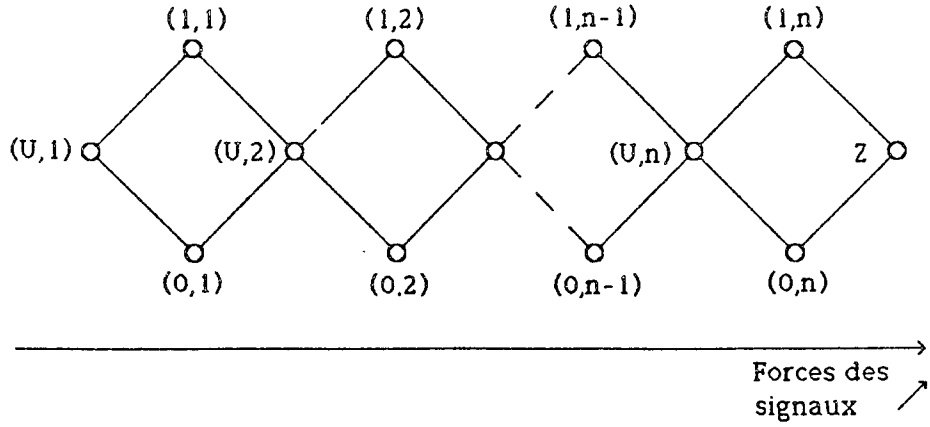
Les buts visés par CSA sont identiques à ceux de MOSSIM : la description de réseaux de transistors considérés comme interrupteurs avec prise en compte des problèmes de type partage de charge, logique, cablée, etc. au niveau logique.

En CSA un circuit est décrit comme un réseau composé des éléments suivants :

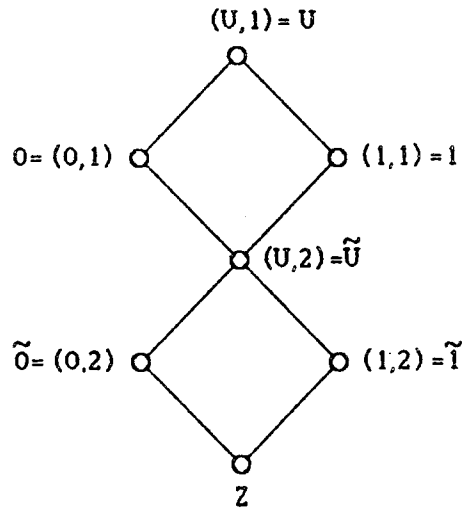
ELEMENT	REPRESENTATION CSA	REPRESENTATION USUELLE
connecteur		
Interrupteur positif		
Interrupteur négatif		
Atténuateur		
Amplificateur		
Capacité (well)		

La valeur d'un noeud est représentée par un couple (VL,F) avec VL $\in$ {0,1,U,Z} représentant la valeur logique du noeud et F sa force ; l'ensemble des forces est à priori non borné mais pour la circuiterie MOS courante, 2 niveaux suffisent.

Dans le cas général l'ordre partie sur les couples (VL,F) est le suivant :



Dans le cas n = 2 on notera $V_4 = \{0,1,\tilde{0}\ \tilde{1},Z,U,\tilde{U}\}$ l'ensemble défini par :



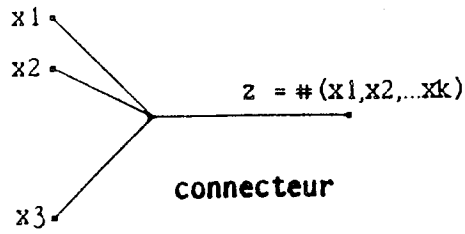
Remarque

- La force d'un noeud est assimilable au courant maximal qui peut le traverser,
- On a (Z,n) = (Z,n-1) = ... = Z, valeur la plus "faible".

i) Connecteurs

Un connecteur est un dispositif orienté qui réalise la fonction $\dot{+}$ sur les entrées, cette fonction étant définie par

$z_c = \dot{+}(x_1,..x_n)$ - z_c est l'élément le plus faible de l'ensemble des valeurs manipulées et $z_c \geq x_i\ \forall\ i = 1,k$



Exemple : k = 2 et n = 1 ; (V_4 = {0,1,Z,U})

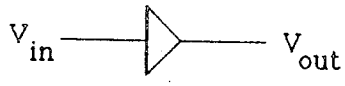
x2 \ x1	0	1	Z	U
0	0	U	0	U
1	U	1	1	U
Z	0	1	Z	U
U	U	U	U	U

ii) Interrupteur

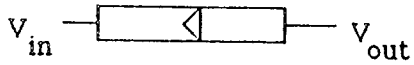
Un interrupteur est considéré comme un connecteur à 2 états : passant ou bloqué. Dans le cas ou la commande de l'interrupteur passe à 1 (cas d'interrupteurs de type positif), les 2 ports D1 et D2 possèdent la valeur # (D'1,D'2), D'1 et D'2 désignant les valeurs de D1 et D2 avant que l'interrupteur ne soit passant ; l'interrupteur se comportant ensuite comme un connecteur.

iii) Attenuateur - Amplificateur

Un atténuateur (resp. un amplificateur) est un dispositif **logique unidirectionnel** dont la fonction est de générer un signal plus faible (resp. plus fort) que le signal d'origine.



V_{in}	V_{out}
$\widetilde{0}$, 0	0
$\widetilde{1}$, 1	1
$\widetilde{U}$, U	U
Z	Z



V_{in}	V_{out}
$\widetilde{0}$, 0	$\widetilde{0}$
$\widetilde{1}$, 1	$\widetilde{1}$
$\widetilde{U}$, U	$\widetilde{U}$
Z	Z

Remarque

les définitions d'atténuateur et d'amplificateur sont généralisables à V_n ; $\forall$ n.

iv) Capacité

Une capacité (well) est un dispositif logique possédant 4 états S0,S1,SU,SZ correspondants respectivement à la mémorisation des valeurs logiques 0,1,U et à l'état déchargé.

Dans V4, le comportement d'une capacité CSA est donné en [HAY 82] par l'intermédiaire du graphe de la figure 5.

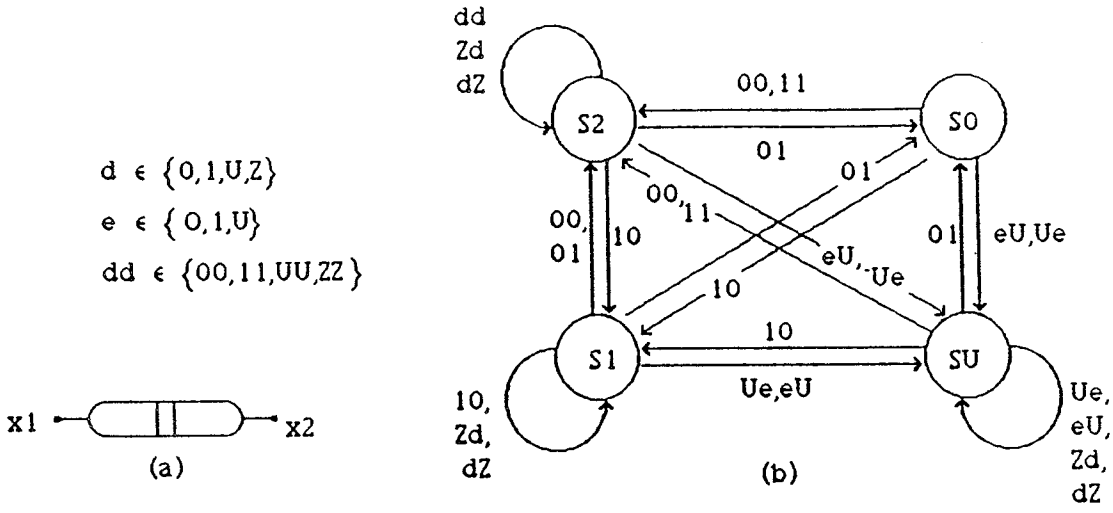


Figure 5a : capacité, 5b : graphe d'état

Remarque

Dans ce modèle de capacité les états S0 et S1 sont parfaitement interchangeables car ce qui est modélisé est la valeur absolue de la charge de la capacité. Le fait d'appeler S0 (resp. S1) l'état de la capacité donné figure 6a et S1 (resp. S0) l'état de la capacité donné figure 6b est arbitraire.

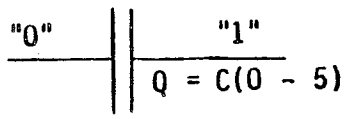


Figure 6a

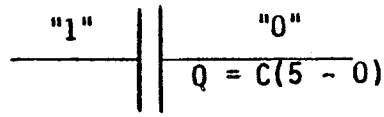


Figure 6b

De plus si l'on considère le dispositif donné figure 7, si la ligne L1 passe de 1 à 0 alors L2 passe à Z mais doit conserver du fait de la capacité C1 la valeur 1 ; le modèle donné ne donne pas d'explications sur la façon dont se passe l'interaction de l'état de la capacité sur les deux ports x1 et x2, ni de possibilité de modélisation des courants de fuite.

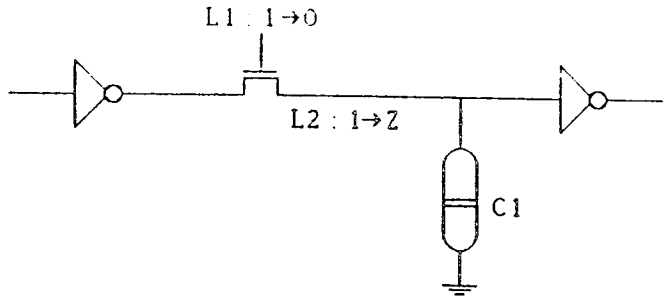
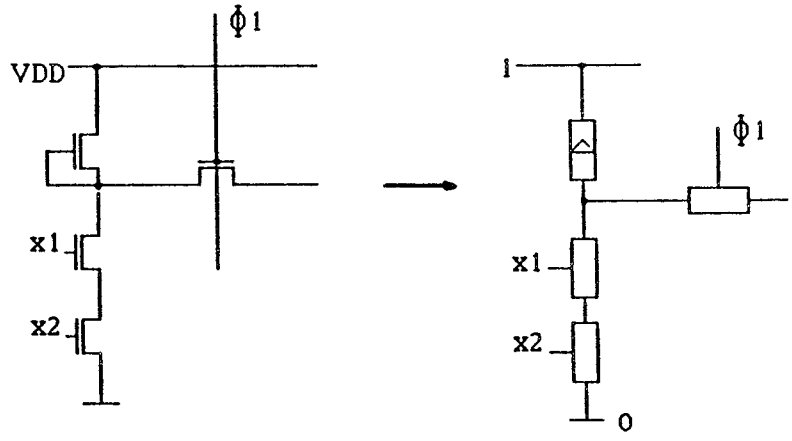


Figure 7

Exemple de modélisation CSA



Remarque

Une approche systèmatique des algèbres multivaluées est donnée dans [HAY 84b] ; on y trouvera aussi une application à la détection d'aléas.

III.3.3 - DIFFERENCES ENTRE MOSSIM ET CSA

- 1) Au niveau des valeurs manipulées, MOSSIM introduit la notion de force au travers des conductances γ_i associées aux interrupteurs et des tailles associées aux noeuds ; dans CSA cette notion est introduite au niveau de l'algèbre de base. Ces deux méthodes sont équivalentes au niveau des possiblités de modélisation, on peut préférer MOSSIM pour des raisons de proximité avec la réalité physique du circuit. Cette raison est renforcée par le fait qu'il existe en CSA des objets (atténuateurs et amplificateurs) qui

ne sont pas, en ce qui concerne la connectique, directement issu du schéma du circuit.

- 2) Au contraire de MOSSIM [BRY 84] qui traite le cas des variables de commande d'un transistor dans l'état U ou Z, CSA élude ce problème en le ramenant à la technologie sous jacente et en affirmant donc qu'un transistor avec sa grille en haute impédance peut être éventuellement considéré comme gardant son état précédent ; en CSA apparait la notion de réseaux à "comportement correct", réseau ou les commandes des interrupteurs prennent leurs valeurs dans {0,1}.

- 3) On peut noter la forte notion d'unidirectionalité qui apparait dans CSA connecteurs, atténuateurs et amplificateurs sont orientés [HAY 82] ; dans [HAY 84a], la bidirectionnalité est traitée comme superposition de 2 signaux de direction opposées.

- 4) Sur le plan formel le modèle CSA étant une manipulation sur l'algèbre des valeurs {0,1,U,Z} x {forces}, il pourrait paraitre plus puissant de décrire ces manipulations non au travers d'opérateurs matériels prédéfinis mais aux travers d'un certain nombre de fonction sur ces valeurs. Les modèles matériels seraient alors construits à partir de ces fonctions de base.

- 5) En ce qui concerne la temporisation des modèles, MOSSIM ne donne aucune possibilité de prendre en compte des caractéristiques temporelles ; CSA par contre introduit la notion de temps au niveau des changements d'état de la capacité CSA avec la remarque suivante : de la même façon que le nombre de valeurs possibles pour les coefficients d'atténuation (alternateurs) dépend du cardinal n de l'ensemble des forces ($n=2$ pour V_7). Le nombre des valeurs possibles pour les temps de charges et de décharges est aussi fonction de n. Si l'on considère V_7 on ne pourra modéliser que les capacités prédominantes dans les réseaux MOS (capacité grille substrat). Notons enfin que ce modèle temporel n'est pas parfaitement défini.

- 6) Les deux systèmes MOSSIM et CSA s'intéressent à la simulation de défauts pour plus d'information on consultera [BRY 85] et [HAY 84a].

III.3.4 - AUTRE MODELE

Une approche semblable à celle de CSA est proposée dans [CER 83] et [RAY 85] où un circuit est décrit comme assemblage de dispositifs (cellules) soit de base (interrupteurs, connexions multiterminal, connexions) soit composés. A partir de cette description (figure 8a), un arbre de décision binaire est automatiquement extrait (figure 8b). L'extraction des arbres de décision binaires peut être très complexe (cas des chaines de Manchester par exemple), dans certains cas l'arbre final ne sera pas généré à la compilation mais dynamiquement lors de la simulation ce qui est fortement pénalisant.
Les arbres associés aux sous circuits sont mémorisés sous forme de programmes de décision et interprérés par un mécanisme de simulation à 2 niveaux : le premier mécanisme de simulation traite les circuits composés à partir de dispositifs de base (simulation du type interrupteur) ; le deuxième mécanisme traitant l'assemblage de ces circuits composés (simulation du type évènementiel).

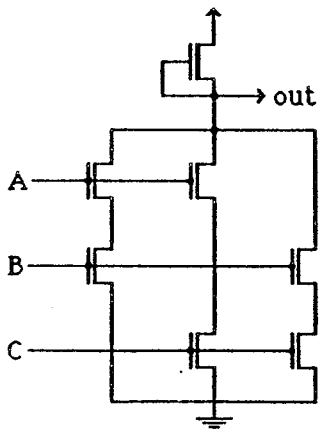


Figure 8a
voteur à la majorité

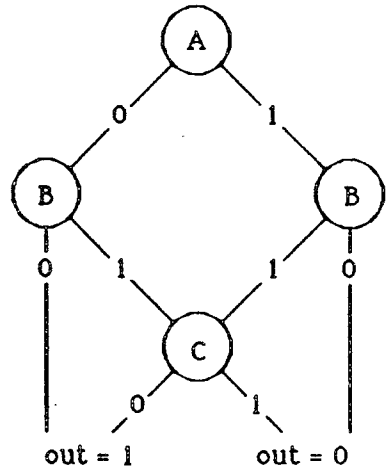


Figure 8b
arbre de décision binaire

III.4 - Simulation logique

III.4.1 - INTRODUCTION

Avant tout développement nous allons préciser ce que l'on entend usuellement sous l'expression "simulation logique".
Les langages de description au niveau logique (nous verrons plus loin que pour une même description il existe plusieurs modes de simulation) permettent de décrire un circuit donné comme assemblage d'objets primitifs à comportement logique (discret) : des portes ; le seul moyen de création d'objet nouveau étant la construction structurelle.

Si l'on se bornait à ne considérer comme langage de niveau logique que les langages pour lesquels l'ensemble des objets prédéfinis est composé de portes logiques simples (INV, NAND,...), il est évident qu'aucun langage actel ne correspondrait à cette description. Nous considérerons donc comme langage de niveau logique les langages possédant un certain nombre d'objets prédéfinis complexes (registres, mémoires,...) mais n'offrant pas la possibilité de définition fonctionnelle de nouveaux objets (on pourra se reporter aux définitions données dans la première partie), une certaine flexibilité des modèles étant obtenue par **paramétrisation.**

Lorsque l'on considère l'expression de "simulation logique", il faut distinguer les 2 notions de niveau et de mode de simulation ou bien encore distinguer les objets manipulés (portes et autres) de la manière dont ils sont manipulés. Les modes de simulation couramment utilisés sont : délais unitaires, délais nominaux, pire cas et simulation de défauts.

Une autre manière de définir la simulation logique est de dire ([LEIN 81]) qu'elle ne manipule que des objets dont la temporisation interne (activité) n'est pas accessible à l'utilisateur ; toutes les évaluations des sorties étant référencées par rapport aux instants de variations des entrées, les différents modes de simulation ne servant qu'à affiner l'instant de prise de valeur de la sortie. Une conséquence de cette remarque étant que la réalisation pratique d'un tel simulateur est du type dirigé par les évènements (le flot des données).

Remarque : Principe des simulateurs dirigés par évènements

Si l'on considère le circuit donné en figure 9, un passage de 0 à 1 de IN1 provoque l'évaluation des fonctions associées aux portes P1,P2,P3 comme seule la sortie de P2 est modifiée il y a création de l'évènement "modification de I2" et évaluation des portes qui lui sont reliées. Cet évènement va entrainer la création de deux nouveaux évènements sur O1 et O2.

Notons encore une fois que l'on peut séparer l'évolution logique et l'évolution temporelle (mode de simulation), les caractéristiques temporelles ne fixant que la date d'apparition de l'évènement.

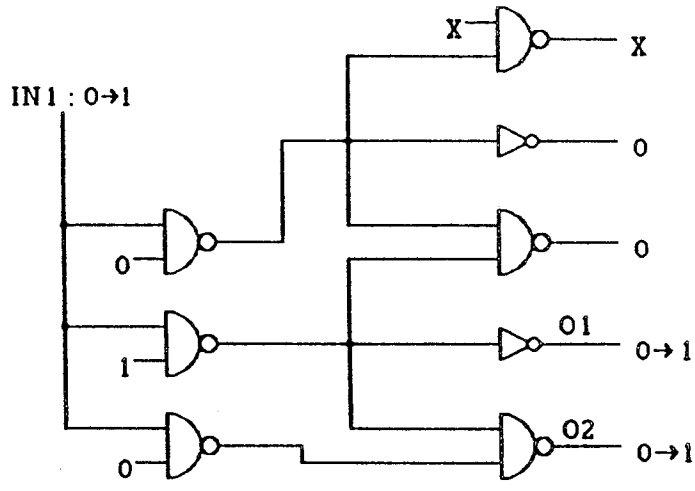


Figure 9

Pour terminer cette introduction on peut remarquer que les langages du niveau logique sont en train d'évoluer de 2 manières :

Evolution vers le bas : possibilité de description d'objets primitifs de types particuliers et liés à une technologie donnée.

Evolution vers le haut : possibilité de descriptions fonctionnelles.

En ce qui concerne l'évolution vers le haut, dans la majorité des cas, l'algorithme de simulation reste dirigé par les évènements : les modèles fonctionnels ne permettant pas d'associer une activité indépendante à un modèle mais plutôt d'effectuer des opérations "instantanées" plus ou moins complexes ; on pourra alors parler de niveau **logico-fonctionnel**.

Nous allons maintenant étudier quelques exemples de simulateurs logiques permettant de prendre en compte les spécificités de la circuiterie MOS, nous donnerons ensuite l'approche développée dans le langage CAP/DSDL pour résoudre ces problèmes et nous terminerons par le langage CADOC.LD.

III.4.2 - PROBLEMES LIES A LA TECHNOLOGIE MOS

Nous regrouperons sous ce titre un certain nombre de problèmes qui ne sont pas forcément exclusifs à la circuiterie MOS mais qui sont traités dans un cadre restreint aux domaines de variations discrets : on ne cherchera donc pas à décrire certains dispositifs à caractère analogique (amplificateurs différentiels par exemple).

Les problèmes que nous étudierons ici sont :

- la modélisation de la bidirectionnalité,
- la modélisation de la logique cablée de la logique 3 états,
- la modélisation du partage des charges et la logique dynamique.

Nous ne nous intéresserons pas particulièrement à la modélisation des délais qui est un problème qui peut être considéré comme "mineur" par rapport aux problèmes précédents. Pour un exemple de modélisation temporellement précises on se reportera à [OKA 83] et à la partie consacrée aux extensions de CADOC.LD.

Deux exemples de circuits dont la modélisation peut poser des problèmes au niveau logique sont donnés en figure 10, et 11 (point mémoire statique).

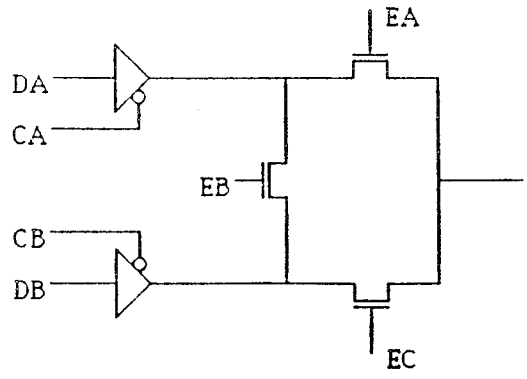


Figure 10

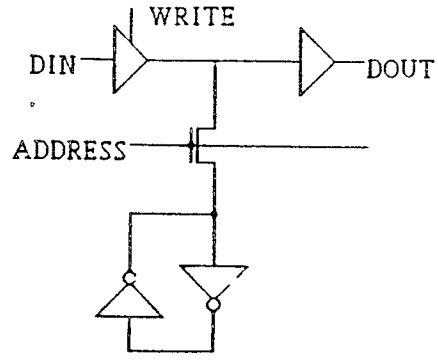


Figure 11

III.4.3 - EXEMPLES DE SIMULATEURS LOGIQUES APPLIQUES A LA CIRCUITERIE MOS

Nous allons présenter un certain nombre de simulateurs logiques prennant en compte les problèmes précédents ; pour chacun nous donnerons l'ensemble des valeurs manipulées, les opérateurs matériels prédéfinis (autres que les portes classiques et nous ferons quelques remarques. Notons ici que le choix de ces simulateurs a été fait plus ou moins arbitrairement dans le nombre important de simulateurs disponibles, certains exemples étant choisis pour illustrer un point particulier.

i) **HILO MARK II** ([FLA 80,FLA 83])

Valeurs manipulées : 6 → 0,1,X,Z,Z0,Z1
avec Z0 = {Z,0} et Z1 = {Z,1}

dispositifs prédéfinis : porte trois-états unidirectionnelle
porte trois-états bidirectionnelle
(porte de transfert)
connexion d'entrée
connexion trois états

Algorithme de simulation : algorithme classique par évènement.

Remarque

Dans [FLA 83] à une variable est associée non une valeur mais un ensemble de valeurs, ce qui permet de prendre en compte les ambiguités liées par exemple à des commandes de portes trois-états possédant la valeur X et de réduire le pessimisme des simulateurs classiques. Cette méthode est semblable à celle proposée dans le langage CAP/DSDL et se fait au prix d'une augmentation non précisée (exponentielle) des calculs à effectuer.

ii) LODGET ([ALM 84])

valeurs manipulées : 8 → 0,1,
E (erreur ou valeur inconnue),
R (front montant)
F (front descendant),
Z,
L (transition à partir de la valeur Z),
H (transition vers la valeur Z).

Dispositifs prédéfinis : porte trois-états unidirectionnelle,
porte trois-états bidirectionnelle,
bus.

algorithme de simulation : classique par évènement.

Remarques

- le système LODGET est écrit en simula
- le comportement des dispositifs primitifs sont donnés à partir de tables (la définition de ces tables par le concepteur n'est pas mentionnée)
- notion de transition illégale (F → L) ou composée (0 → Z)
- les éléments bidirectionnels sont transformés par un prétraitement en un ensemble d'éléments unidirectionnels : (le circuit donné en figure 12 est automatiquement transformé en celui de la figure 13). La complexité et la faisabilité de cette transformation n'est pas évoquée dans l'article (exemple évident).

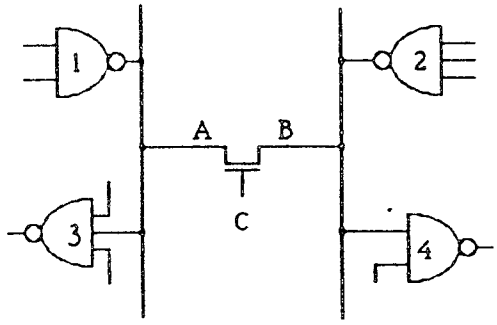


Figure 12 : Circuit initial

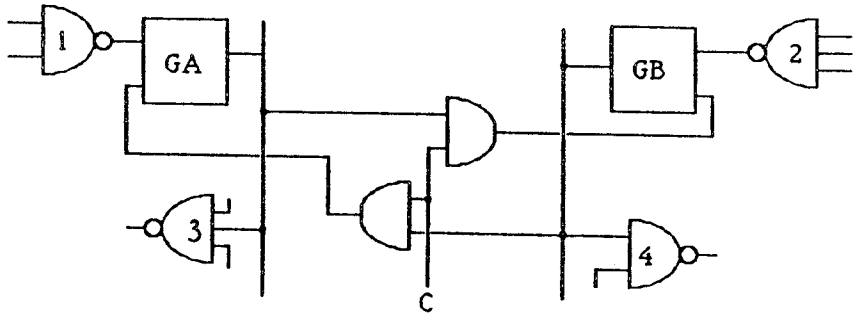


Figure 13 : circuit après traitement

(G1 et GB sont des noeuds de type bus dont l'évolution est donnée par des tables fonctions de la technologie).

iii) MADSIM ([KNU 82]) °

Valeurs manipulées :

9 → 0,1

R (front montant mais très en dessous du seuil),

CR (front montant mais proche du seuil),

T1 (passage du seuil dans le sens 0 → 1),

F (front descendant mais très en dessus du seuil),

T0 (passage du seuil dans le sens 1 → 0),

CF (front descendant mais très proche du seuil),

X (état inconnu).

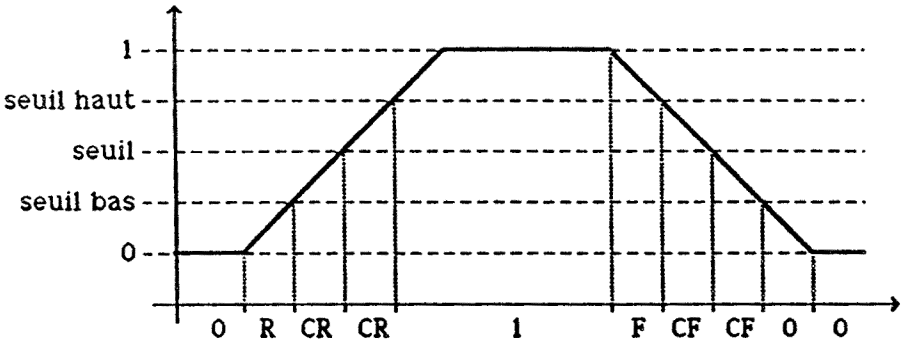


Figure 14 : valeurs MADSIM

Dispositifs prédéfinis : modèle de transistors.

Algorithme de simulation : classique par évènements et évaluation spéciale des réseaux de T.MOS

Remarques

- Cet ensemble d'états se prête bien à l'interfaçage numérique analogique.
- De part les objets manipulés (transitors MOS), ce type de simulateur possède des caractéristiques des simulateurs niveau switch avec de plus des primitives logiques. L'application à la logique trois-états ne semble pas abordée.

- Lors de l'évolution les 9 états des interconnexions sont transformées en 3 états d'un réseau possèdant 2 ports d'entrées-sorties. Ces 3 états indiquant si le réseau est équivalent à une forte résistance (NU), une résistance inconnue (NX) ou une faible résistance (ND).

Un réseau de TMOS est évalué par regroupement suivant le principe donné en figure 15a et 15b et grâce à des tables de celles du type indiqué en figure 16 une table par mode de construction).

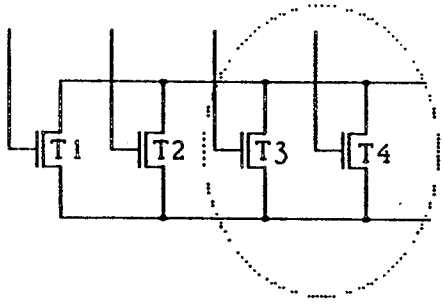


Figure 15a
circuit initial, le premier regroupement crée un noeud dont la valeur est NU,ND ou NX.

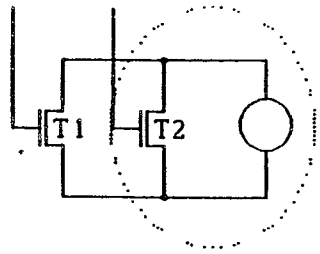


Figure 15b
incorporation du TMOS T2 au noeud puis incorporation de T1

état de l'elément à incorporer	ND	NU	NX	noeud
TO	ND	NU	NX	
T1	ND	ND	ND	
X	ND	NX	NX	
R	ND	NU	NX	
CR	ND	NX	NX	

etc...

Figure 16 : principe d'incorporation d'un élément à un noeud

iv) [WAT 80]

Valeurs manipulées : 3 valeurs statiques 0,1,X.
4 valeurs dynamiques $\tilde{0}$,$\tilde{1}$,Z et E (erreur)

Dispositifs prédéfinis : porte de transfert bidirectionnelle,
connexion,
buffer trois-états,
bus,
logique à précharge.

Remarques

- Les valeurs dynamiques correspondent aux valeurs faibles classiques.
- Le comportement des dispositifs est défini par l'intermédiaire de tables.

Exemple : ET cablé MOS

entrée 2 \ entrée 1	0	1	X	$\tilde{0}$	$\tilde{1}$	Z	E
0	0	E	E	0	0	0	E
1	E	1	E	1	1	1	E
X	E	E	E	X	X	X	E
$\tilde{0}$	0	1	X	$\tilde{0}$	*	$\tilde{0}$	E

etc...

* représentant la dernière valeur pilotant le bus ($\tilde{0}$ ou $\tilde{1}$).

- Toutes les structures MOS sont décrites par l'intermédiaire de dispositifs prédéfinis, il n'existe pas de modèle de transistor isolé. La partie consacrée aux portes de transfert bidirectionnelle est imprécise (utilisation unidirectionnelle ?).
- Ce simulateur est un exemple type de simulateur développé uniquement pour la circuiterie MOS.

v) [McD 82]

Valeurs manipulées : 4 → 0,1,X,Z

Eléments prédéfinis : portes de transfert uni et bidirectionnelles
logique cablée,
logique à précharge.

Remarques

- Le comportement des dispositifs est donné au travers de tables.
- Ce simulateur comme tout les simulateurs ne possédant pas la notion de force (ou un équivalent) ne peut traiter correctement la bidirectionalité. Dans l'exemple proposé la notion de force est contournée par la connaissance a priori du port imposant le sens de transfert de l'information et l'utilisation du modèle orienté traduisant cette connaissance. Le modèle de porte de transfert bidirectionnelle existant étant hautement pessimiste et inutilisable.

vi) BIMOS ([STE 83])

valeurs manipulées :

3 valeurs logiques {0,1,X}

5 forces {E (external), D (driven), R (résistive), L (large high impedance), Z (high impedance)}

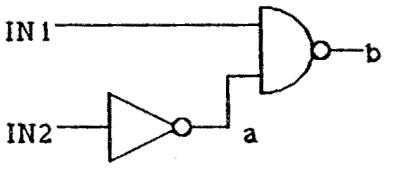
éléments prédéfinis : transistor MOS (tous modes).

Algorithme de simulation : partitionnement du circuit en sous réseaux par coupure des liaisons bidirectionnelles (relaxation). A l'intérieur d'un sous réseau, la propagation des valeurs est faite suivant le flot des données avec superposition des flots opposés, les délais sont calculés après l'évaluation logique des sorties des sous réseaux. Le traitement des sous réseaux est fait de manière classique (algo. par évènements).

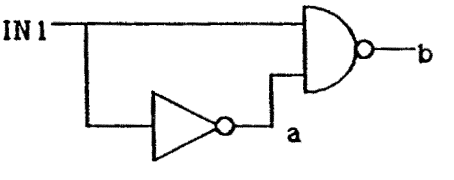
Remarques

- BIMOS fait partie des logiciels associés à HELIX (cf. partie 1, II.1).
- Les temps de montée et de descente sont associés aux éléments prédéfinis et non aux ports de ces éléments ; en contrepartie, il est possible d'associer des délais aux interconnexions (simulation pire cas).
- Définition d'une pseudo valeur $\tilde{X}i$ utilisée en phase d'initialisation uniquement (l'indice i indiquant la source de la valeur X ou $\tilde{X}$) pour essayer de diminuer le nombre de variables initialisées à X en fin d'initialisation.

Exemple



Circuit 1



Circuit 2

Dans le cas du circuit 1, la sortie restera initialisée à X car les entrées du NAND portent les valeurs X_{in1} et $\bar{X}_{in2}$ qui n'ont pas la même origine.
Dans le cas du circuit 2 les entrées du nand valent X_{in1} et $\bar{X}_{in1}$ et b est initialisé à 1.

Cette construction est intéressante sur l'exemple mais son extensibilité à des cas réels (circuits plus complexes, grilles de transistors à X) n'est pas assurée, son extension débouchant sur les techniques de manipulations formelles d'expressions : l'initialisation de b à 1 dans le circuit 2 venant de l'égalité non(a et non a) = 1

III.5 - Simulation Mixte

Le but de la simulation mixte (ou hybride) est de permettre l'utilisation sur un même circuit de plusieurs types de simulateurs : en effet, si pour des raisons d'efficacité il est préférable d'utiliser des simulateurs ne manipulant que des variables à domaines de définition discrets (simulation logique, logico-fonctionnelle, fonctionnelle), pour des raisons de précision il est parfois souhaitable d'utiliser des simulateurs "continus" (simulation timing,...) pour les parties critiques des circuits.

Des exemples de simulateurs mixtes sont donnés dans les deux systèmes organisés autour de MOTIS ([AGR 80], [NHA 80], [CHE 84]) et SPLICE ([NEW 79]) dans lesquels il est possible d'associer un type de simulation à chaque sous-circuit.

III.5.1 - MANIPULATION DU TEMPS

Dans les simulateurs continus, le temps est considéré comme variant continuement (modulo la discrétisation liée aux méthodes de résolution des systèmes) alors que dans les simulateurs du type logique, le temps est discret (modulo les évolutions du type boucle de stabilisation).
La juxtaposition de ces 2 types de temps se fait par un superviseur appelant les differents simulateurs et gérant le temps ; le mécanisme (du type événementiel) utilisé dans MOTIS est donné en figure 17.

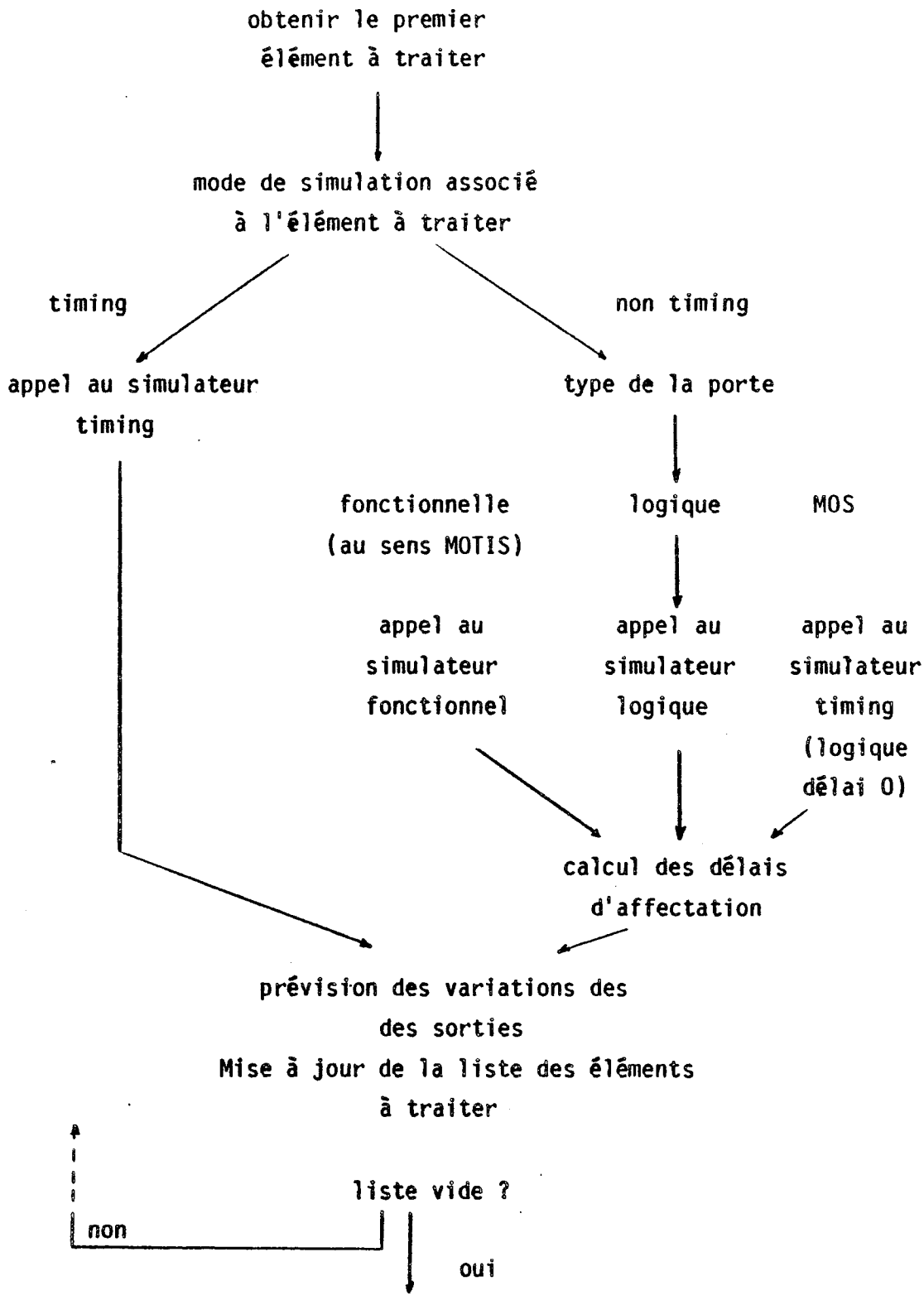

obtenir le premier
élément à traiter
mode de simulation associé
à l'élément à traiter
timing
non timing
appel au simulateur
timing
type de la porte
fonctionnelle
(au sens MOTIS)
logique
MOS
appel au
simulateur
fonctionnel
appel au
simulateur
logique
appel au
simulateur
timing
(logique
délai 0)
calcul des délais
d'affectation
prévision des variations des
des sorties
Mise à jour de la liste des éléments
à traiter
liste vide ?
non
oui

Figure 17

III.4.2 - INTERFACAGE LOGIQUE ANALOGIQUE

L'utilisation simultanée de variables continues et discrètes passe par des mécanismes de traduction (conversion) du type analogique → logique et logique → analogique.

i) Traduction analogique-logique

La réalisation de cette conversion est fonction du nombre de valeurs de référence (seuils) appliquées aux modèles continus, du nombre de valeurs logiques cibles de la conversion et de l'algorithme utilisé.

La figure 18a donne un exemple de conversion avec 2 niveaux d'échantillonnage V0 et V1 sur l'ensemble cible {0, 1, U}.
La figure 18b indique la conversion du même signal avec 3 niveaux d'échantillonnage (V0, V1 et VM), le même ensemble cible et un algorithme différent ; la superposition des deux résultats est donnée figure 18c.

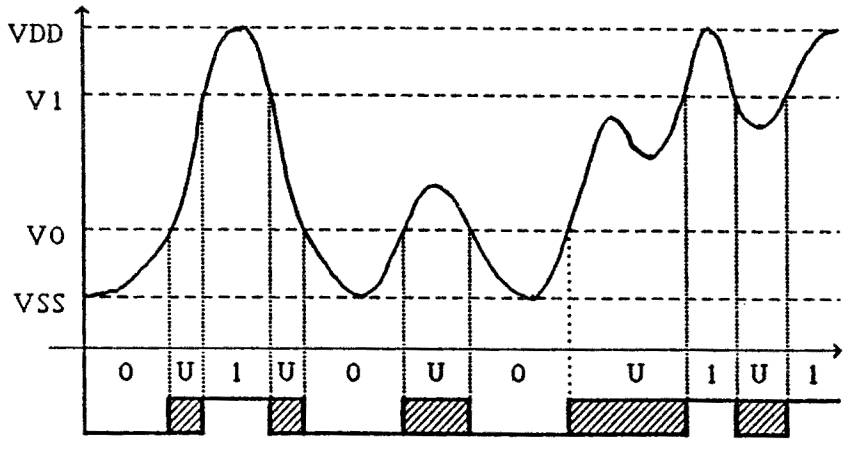


Figure 18a : conversion A → L avec 2 niveaux d'échantillonnage.

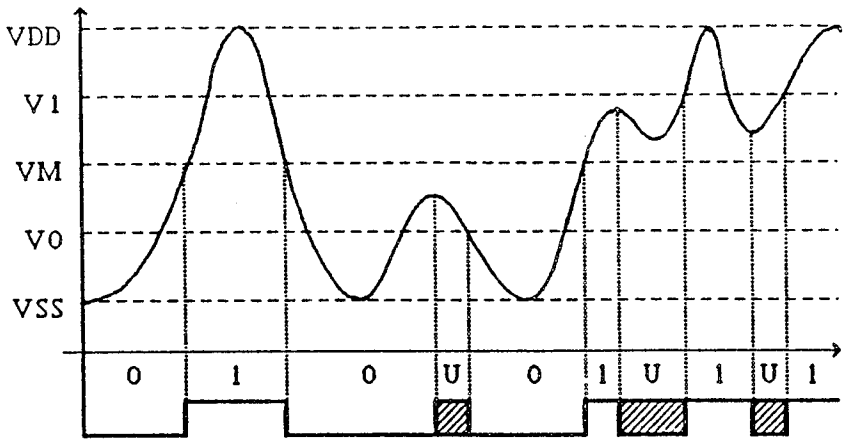


Figure 18b : conversion A → L avec 3 niveaux d'échantillonage.

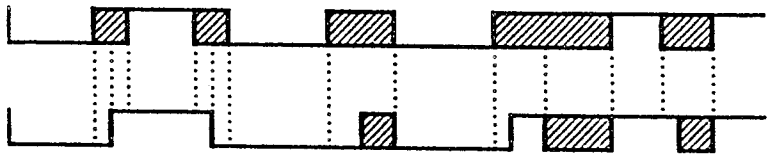


Figure 18c : comparaison des résultats

De manière schématique, parmi tous les algorithmes possibles, on peut distinguer les algorithmes "sans retard" et les algorithmes "avec retard".

Dans les algorithmes "sans retard" (cf. figure 18a), la valeur logique à un instant t est fonction de la valeur analogique au même instant ; le signal logique est une traduction immédiate du signal analogique mais on peut considérer que cette traduction fournit "trop" d'informations pour une simulation logique classique (génération de trop d'évènements du type passage à U).

Une solution possible à cet excés d'évènements pénalisant la simulation logique est de ne générer un U que lorsque le signal analogique semble indiquer effectivement un aléa (en particulier les transitions analogiques du type 0 → U → 1 seront considérées comme correctes et seront traduites par la transition logique 0 → 1).

Dans l'algorithme proposé en figure 18b, l'état U est déterminé par le changement de pente du signal analogique dans la zone critique ⌊V0, V1⌋ ; le terme d'algorithme "avec retard" venant du fait que l'état U ne sera par exemple détecté qu'au temps t2 alors que depuis t1, le signal n'a plus une valeur logiquement interprétable comme "0".
En conclusion, on a donc un compromis à réaliser entre pessimisme et optimisme, précision et excés d'information.

ii) Traduction logique-analogique

La différence fondamentale entre la traduction A → L et la traduction L → A est que cette dernière suppose un enrichissement de l'information.

On va considérer ici une traduction d'un signal à valeurs dans {0, 1, U} vers un signal à valeurs continues (⌊VSS,VDD⌋) et en particulier les transitions du type X → U avec X = 0 ou 1.

Dans un premier cas, le simulateur logique est capable de prédire à l'instant courant t0 la valeur du signal logique jusqu'à la date t0 + Δ, dans ce cas, le traducteur L → A pourra lisser le signal analogique résultant en fonction de divers paramètres (formes de courbes de référence, technologie) fournis par l'utilisateur (figure 19a). Ce type de simulateur peut être qualifié de prédictif, un exemple étant le simulateur de CADOC.LD.

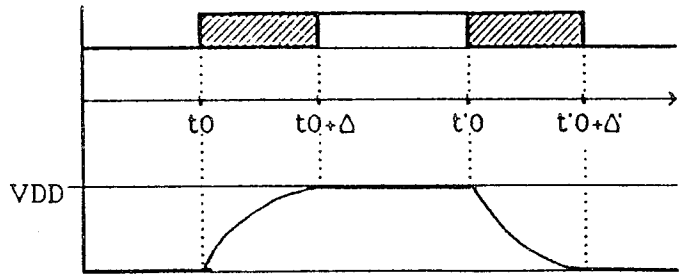


Figure 19a : traduction L → A avec un simulateur logique prédictif.

Dans un deuxième cas, si le simulateur ne fournit que des valeurs logiques à l'instant courant, la seule solution dans le cas d'une transition X → U sera de faire passer la valeur du signal analogique à une valeur "moyenne", en tenant compte éventuellement des problèmes de dérivabilité des signaux (cf. figure 19b).

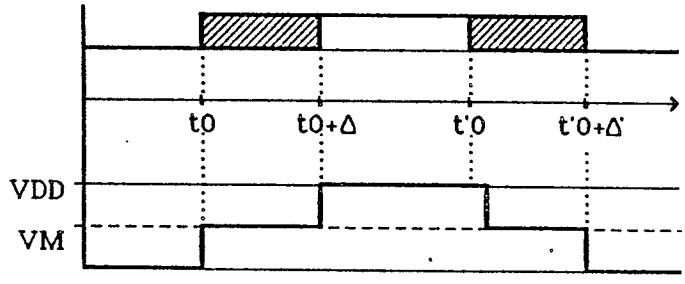


Figure 19b : traduction A → L avec simulateur logique non prédictif.

iii) Prise en compte de la haute impédance

La valeur haute impédance Z (et ses variations haute impédance 0, haute impédance 1 suivant les systèmes) pose un problème lors de la simulation mixte car la signification électrique de l'état logique Z et la génération de la valeur logique Z à partir d'un ensemble de conditions analogiques) ne sont pas immédiates.

L'approche donnée en [NEW 79] consiste à maintenir les valeurs analogiques courantes si le signal logique passe à Z (cas de la traduction L → A) et à affecter la valeur logique Z à tout les noeuds pour lesquels les valeurs du courant et de la conductance qui leur sont associés sont "faibles".

iv)Remarques

- Certains simulateurs présentant un ensemble de base bien adaptés à la traduction L → A (cf. MADSIM, III.4.3).

- Les approximations introduites lors des changements de représentation, et en particulier ceux introduit lors des conversions logique → analogique rendent difficile la justification de la simulation mixte avec fort couplage entre éléments logiques et éléments continus (les simulations continues, utilisées pour leur précision étant initialisées avec des valeurs approximatives).

- Dans le cas d'ensemble de valeurs logique étendu (en particulier avec des valeurs du type X (indifférent)), les mécanismes de traductions sont difficiles à définir.

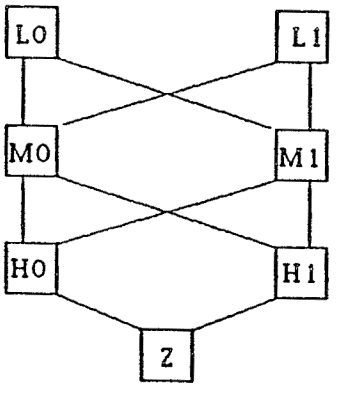
IV - SIMULATION FONCTIONNELLE

IV.1 - CAP/DSDL

Ce langage a été présenté en tant que langage fonctionnel de description de matériel en partie 1, paragraphe III.2, nous allons nous intéresser ici à son application à la description de circuit MOS ([RAM 83b].

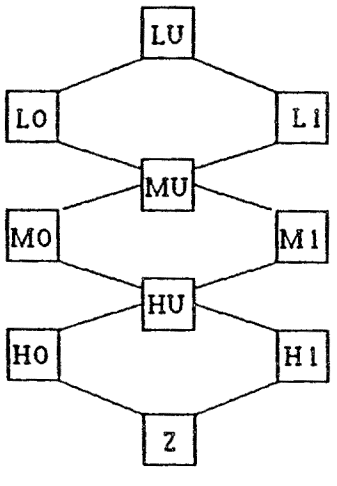
Valeurs manipulées : 7 → LO,L1,MO,M1,HO,H1,Z

Avec l'ordre partiel suivant :



Remarque

Si l'on introduit les valeurs inconnues LU,MU et HU on retrouve le treillis classique (V_{10} de CSA) :



Ces valeurs ne sont pas primitives car on a :

LU = {L1,LO}, MU = {M1,MO} et HU = {H1,HO}

Ces 7 valeurs sont superflues pour la description de la circuiterie MOS, l'ensemble V_7 étant suffisant.

Le point important à noter est qu'à chaque variable (connexion) est associé un ensemble de valeurs ce qui permet de traiter les problèmes liés aux interrupteurs avec des commandes dans l'état inconnu (au prix d'une croissance exponentielle du nombre de configurations à évaluer).

A partir de ces valeurs, l'approche choisie consiste à définir formellement un certain nombre d'opérations (fonctions) sur les variables, puis des modèles utilisant ces fonctions, l'utilisateur n'ayant pas accès aux fonctions primitives.

Cette approche est très semblable à celles de MOSSIM ou CSA avec en plus la notion de temporisation associées aux dispositifs (pas de temporisation en MOSSIM, temporisation fragmentaire au niveau des capacités en CSA).

Objets prédéfinis

- PULLUP : transistor de charge générant un M1
- PULLDOWN : transistor de charge générant un M0 (non utilisé en techno MOS)
- TRANSFER : porte bidirectionnelle
- COLLECT : connecteur avec bidirectionnalité de tous les ports.

Cet ensemble d'objets prédéfinis permet de décrire éventuellement une porte MOS comme assemblage de transistors.

Exemple

la description de la porte donnée figure 20 est la suivante (syntaxe expliquée en III.2, partie 1) :

```
const    nmos = "1" ;
         pmos = "0" ;
var      a,b,c : implicit bit ;
         ap :    implicit bit ;
impdef transfer (pmos,b,"1",ap)
       transfer (pmos,a,ap,c)
       ...
```

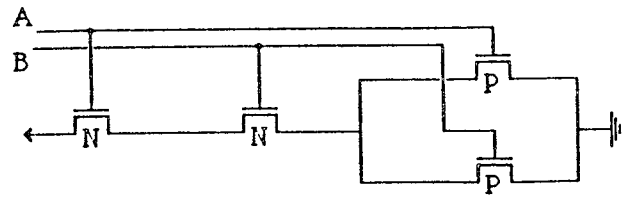


Figure 20

Remarques

- Si l'on considère le principe du langage CAP/DSDL exposé dans la première partie, on peut assimiler ces dispositifs prédéfinis à une extension des opérateurs arithmétiques autorisés en partie définition implicite.
On peut noter que cette extension a été faite au prix d'une modification de l'algèbre de base utilisée au niveau fonctionnel et donc au prix d'une réécriture d'une partie du système (interpréteur en particulier).

- Le langage étant fonctionnel le concepteur peut choisir son mode de description : fonctionnel ou structurel. En ce qui nous concerne, nous pensons que cette possibilité de décrire une porte n'est pas intéressante au niveau de la simulation fonctionnelle : en effet, ce genre de description ne donnera pas plus d'informations temporelles qu'une description fine au niveau fonctionnel (on aura par contre des informations supplémentaires sur la connectique du circuit) avec des temps de simulation moins bons que ceux obtenus en utilisant des simulateurs spécialisés.
Cette remarque étant faite, il est bien entendu nécessaire de pouvoir décrire des objets du type transistor mais lorsqu'ils sont utilisés de manière isolée (en tant que porte de transfert). Sur ce sujet, on se reportera à l'approche développée pour le langage CADOC.LD décrite au paragraphe suivant.

IV.2 - CADOC.LD : Propositions et extensions

Nous avons vu dans les paragraphes précédents que la notion de force associée à un signal permet de prendre en compte les problèmes liés à la circuiterie MOS. Dans une première partie nous allons montrer comment cette approche est possible en CADOC.LD, sans modification du langage ; dans une deuxième partie, nous présenterons une autre possibilité de modélisation des dispositifs MOS : approche séquentielle.

IV.2.1 - APPROCHE MULTIFORCE

L'implémentation de cette approche est immédiate : au lieu d'associer à une variable un type simple, il suffit de lui associer un type composé de deux champs : un champ valeur et un champ force.

Si par exemple on définit le type enregistrement suivant :

```
connecteur_MOS = enreg
                   valeur : bool (* type étendu : 0,1,X,Z,U *) ;
                   force : entier ; (* avec valeur = Z → force = 0 *)
                 fin
```

On pourra ensuite définir des dispositifs travaillant sur ce type.

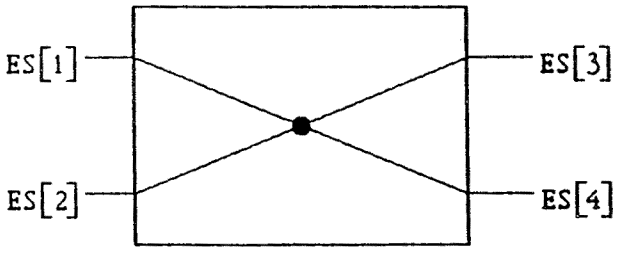


Figure 21 : connecteur

L'opérateur de connexion (logique cablée) avec bidirectionnalité de tous les ports (figure 21) sera défini par la description CADOC.LD donnée pages suivantes.

```
rgf diese : op ;
    type
          connecteur_MOS = enreg
             valeur : bool ;
             force : entier ;
          fin
     bidir
          es : tableau [4] de valeur_MOS ;
                        (* interconnexion de 4 ports *)
    var
          resval, valcour : bool ;
          i, resforce : entier ;
```

partie déclarative de l'opérateur de connexion

```
fonction (* sous forme textuelle *)
    action
     p1 : dummy :=
        debut
           resval := es[1].valeur ;
           resforce := es[1].force ;
                              (* force et valeur du signal résultant *)
           i := 2 ;
           tantque i < 4 faire
               si es[i].force   resforce alors
                  resval := es[i].valeur ;
                  resforce := es[i].force ;
               sinon
                  si es[i].force = resforce alors
                        valcour := es[i].valeur ;
                        resval := choix resval/valcour dans
                           0/1,1/0 : U ;
                                    (* possibilité de message d'erreur *)
                           autres : resval et valcour ;
                        fchoix ;
                   fsi ;
               fsi ;
               i := i + 1 ;
           ffaire ;
             (* propagation de la valeur commune *)
            i := 1 ;
            tantque i < 4 faire
                 es[i].valeur := resval ;
                 es[i].force := resforce ;
                 i := i+1 ;
            ffaire ;
            return 0 ; (* pour des raisons syntaxiques *)
        fin
    graphe
     t1 : p1 - p1 : change (es[1], es[2], es[3], es[4]) ;
    init p1 ;
fin diese
```

partie fonction de l'opérateur de connexion

Exemples

diese ((0,0) (1,0) (U,0) (0,1)) = (0,1)
diese ((0,0) (1,0) (0,0) (1,0)) = (U,0)
etc...

Remarques

- Les explications sur la syntaxe sont données dans le chapitre consacré au langage CADOC.LD.

- Dans cette approche le concepteur définit lui même le comportement de son opérateur de connexion (il peut définir le comportement d'un ET cablé par simple modification de la description algorithmique) ; ces opérateurs pouvant être bien entendu définis en bibliothèque.

- Cet exemple illustre bien l'aspect fonctionnel puissant du langage (pas de définition d'objets prédéfinis nouveaux) ; il est évident que pour des raisons d'efficacité de simulation, un modèle prédéfini (dont l'algorithme est implanté au niveau langage) serait préférable

IV.2.2 - APPROCHE SEQUENTIELLE

Nous allons présenter cette approche dans le cadre particulier des portes de transfert bidirectionnelles.
Le traitement d'un tel dispositif se réduit à la connaissance du port qui va imposer sa valeur à l'autre ou du port qui est **source d'information**. Pris d'une autre manière, cela signifie qu'une porte de transfert est

intrinséquement bidirectionnelle mais qu'en cours d'utilisation, à un instant donné, elle est unidirectionnelle ; le problème se ramène donc à déterminer **l'orientation instantanée** de cette porte.

Si à t donné la porte TP de la figure 22 est orientée de la gauche vers la droite (GD), cela signifie que le réseau gauche est source d'information et le réseau droit destination d'information. Tant que l'orientation reste inchangée, toute variation de G sera répercutée sur D, par contre, toute variation de D non causée par une variation de G pourra être considérée comme un conflit.
Le changement d'orientation de TP ne pourra se faire que sur passage dans l'état Z de la source d'information; une séquence d'initialisation est appliquée en début de simulation et lorsque la commande passe dans l'état haut.

La description en CADOC.LD du modèle associé à cette approche est donné page suivante, des résultats de simulation de ce modèle appliqué au circuit indiqué figure 23 sont indiqués figure 24.

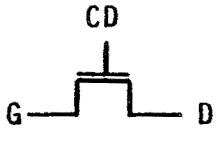


Figure 22 : porte de transfert

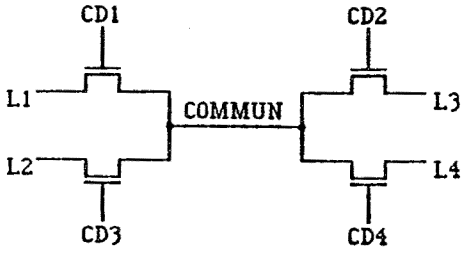


Figure 23

```
RGF PORTE_TRANSFERT (LEFT, RIGHT, CD) : OP ;

 ENTREE
   CD : BOOL ;

 BIDIR
   LEFT, RIGHT : BOOL ;

FONCTION
ACTION
   TR1P0    : ; /PLACE INITIALE /
   TR1P1    : ; /PLACE TELLE QUE CD=F /
   TR1P2    : RIGHT := MUX CD DANS
                         T : LEFT ;
                         U : U ;
                       FMUX ;      / TRANSFERT DE LEFT VERS RIGHT /
   TR1P2_1  : RIGHT := Z ;
   TR1P3    : LEFT := MUX CD DANS
                        T : RIGHT;
                        U : U ;
                      FINMUX ;
   TR1P3_1  : LEFT := Z ;
   TR1P4    : ; /DIRECTION DU TRANSFERT INCONNUE /
   TR1P5    : LEFT, RIGHT := U ;     /CONFLIT DE TRANSFERT /

GRAPHE
   TR1T1   : TR1P0 - TR1P4     : (CD <> F) ET (LEFT = Z) ET(RIGHT = Z) ;
   TR1T2   : TR1P0 - TR1P1     : CD = F ;
   TR1T3   : TR1P0 - TR1P2     : (CD <> F) ET ( LEFT <> Z) ET (RIGHT = Z) ;
   TR1T4   : TR1P0 - TR1P3     : (CD <> F) ET ( LEFT = Z) ET (RIGHT <> Z) ;
   TR1T5   : TR1P0 - TR1P5     : (CD <> F) ET (LEFT <> Z) ET (RIGHT <> Z) ;
   TR1T6   : TR1P1 - TR1P2     : (CD <> F) ET (LEFT <> Z) ET (RIGHT = Z) ;
   TR1T7   : TR1P2 - TR1P2_1   : CD = F ;
   TR1T8   : TR1P2_1 - TR1P1   : V ;
   TR1T9   : TR1P2 - TR1P2     : (CD <> F) : CHANGE (LEFT) ;
   TR1T10  : TR1P2 - TR1P4     :  (CD <> F) ET (RIGHT = Z);
   TR1T11  : TR1P2 - TR1P5     : (RIGHT <> LEFT) : CHANGE (RIGHT) ;
             /DETECTION DE CONFLIT D'ACCES DURANT TRANSFERT LEFT VERS RIGHT/
   TR1T12  : TR1P1 - TR1P3     : (CD <> F) ET  (LEFT = Z) ET (RIGHT <> Z) ;
   TR1T13  : TR1P3 - TR1P3_1   : (CD  = F) ;
   TR1T14  : TR1P3_1 - TR1P1   : V ;
   TR1T15  : TR1P3 - TR1P3     : (CD <> F) : CHANGE (RIGHT) ;
   TR1T16  : TR1P3 - TR1P4     : (CD <> F) ET (LEFT = Z) ;
   TR1T17  : TR1P3 - TR1P5     : (RIGHT <> LEFT) : CHANGE (LEFT) ;
                        /DETECTION CONFLITS RIGHT VERS LEFT/
   TR1T18  : TR1P1 - TR1P4     : (CD <> F) ET (LEFT = Z) ET (RIGHT = Z) ;
   TR1T19  : TR1P4 - TR1P1     : (CD = F) ;
   TR1T20  : TR1P4 - TR1P2     : (CD <> F) ET (LEFT <> Z) ET (RIGHT = Z) ;
   TR1T21  : TR1P4 - TR1P3     : (CD <> F) ET (LEFT = Z) ET (RIGHT <> Z) ;
   TR1T22  : TR1P4 - TR1P5     : (CD <> F) ET (LEFT <>  Z) ET (RIGHT <> Z) ;
   TR1T23  : TR1P1 - TR1P5     : (CD <> F) ET (LEFT <> Z) ET (RIGHT <> Z) ;
   TR1T24  : TR1P5 - TR1P1     : (CD = F);
   TR1T25  : TR1P5 - TR1P4     : (CD <> F) ET (LEFT = Z) ET (RIGHT = Z) ;
   TR1T26  : TR1P5 - TR1P2     : (CD <> F) ET (LEFT <> Z) ET (RIGHT = Z) ;
   TR1T27  : TR1P5 - TR1P3     : (CD <> F) ET (LEFT = Z) ET (RIGHT <> Z) ;

 INIT TR1P0 ;

FIN PORTE_TRANSFERT
```

Figure 23 : porte de transfert

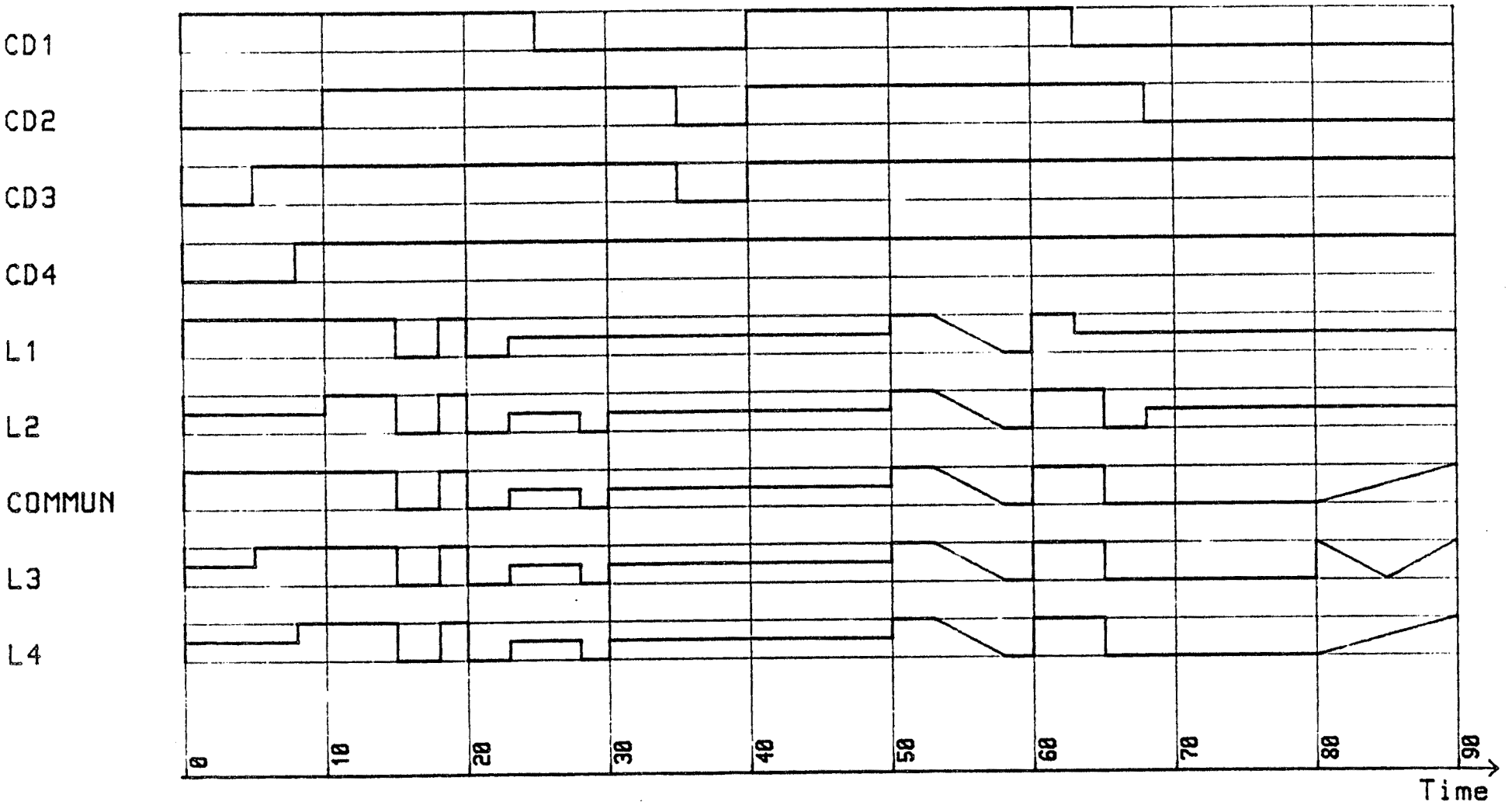


Figure 24

CADOC SYSTEM
Circuit : qua_bool

Remarques

- un développement de cette approche est donné dans [CRA 85]

- dans ce modèle, nous n'avons pas cherché à introduire des caractéristiques temporelles. Des propositions d'extension du langage pour les modélisations fines sont présentées par la suite.

- Il faut noter que ces approches visent uniquement à décrire un transistor MOS isolé, utilisé comme porte de transfert.
Il est hors de question de décrire par ce mécanisme des portes logiques comme assemblage de transistors : les modèles fonctionnels équivalents assurent une précision temporelle au moins équivalente pour une meilleure efficacité.

- Il est évident qu'une porte de transfert unidirectionnelle étant beaucoup plus simple, il est préférable (sur le plan de l'efficacité de simulation) de l'utiliser lorsque la connaissance du circuit global le permet.

Exemple

la porte complexe donnée figure 25 n'a pas à être décrite comme asemblage de 8 transistors MOS mais de la manière fonctionnelle donnée figure 26 où f(A, B, C, D, S) est une fonction éventuellement complexe donnant le temps de réaction de la sortie par rapport à une variation d'une ou plusieurs entrées.

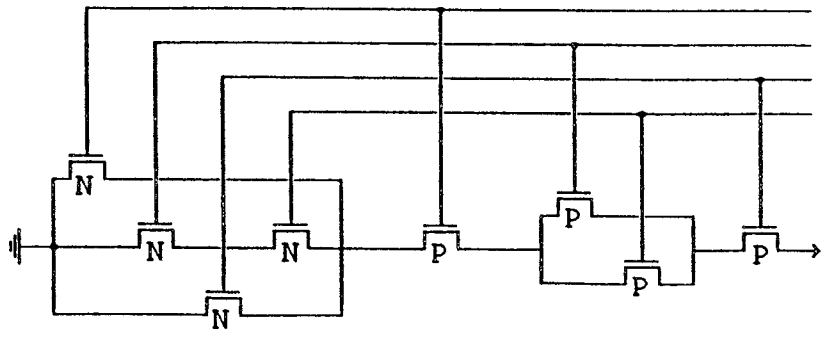


Figure 25 : porte complexe

```
CHANGE (A,B,C,D)
                    S : = [(U,0)
                           (non (A ou B et C ou D), f(A, B, C, D, S))]
```

Figure 26 : partie fonction de la description CADOC.LD

IV.2.3 - EXTENSIONS DU LANGAGE : APPLICATION A LA DESCRIPTION DES CIRCUITS PRECARACTERISES ET PREDIFFUSES.

Dans la deuxième partie, nous avons montré l'adéquation du langage CADOC.LD au problème de la description de circuits à un haut niveau d'abstraction ; dans les paragraphes précédents, nous avons indiqué comment il était possible de prendre en compte les problèmes liés à la technologie MOS.
Nous allons nous intéresser maintenant à la modélisation d'objets de complexité faible du type circuits prédiffusés et précaractérisés.
Le but de cette étude étant de créer, parallèlement aux bibliothèques donnant les descriptions topologiques des circuits, des bibliothèques donnant les modélisations fonctionnelles associées. Un des intérêts étant alors qu'un circuit conçu à partir de ces cellules de bibliothèques pourra être simulé en prenant en comote son environnement, éventuellement complexe et non complétement spécifié ; la modélisation de cet environnement étant faite à un niveau d'abstraction élévé dans le langage CADOC.

i) informations disponibles

Nous allons illustrer ce paragraphe sur un élément simple de bibliothèque : un multiplexeur deux voies vers une (figure 27).

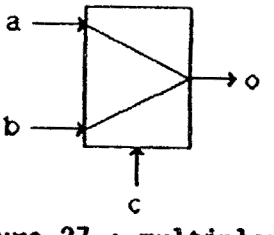


Figure 27 : multiplexeur

Classiquement, la description d'une telle cellule est composée des deux parties suivantes :

*** informations indépendantes de l'environnement**
On y retrouve le comportement du circuit (donné sous forme de table de vérité pour les éléments combinatoires) ainsi que des informations du type temps de traversée intrinsèque de la cellule.

*** informations dépendantes de l'environnement**

Dans cette section de la description de la cellule, on trouve la description de l'influence des ressources connectées sur la ressource décrite ; cette influence étant décrite au niveau des temps de positionnement des sorties en fonction de la capacité totale qu'elles ont à piloter. On y trouve aussi les paramètres indiquant l'influence de la ressource sur les cellules qui la piloteront : valeurs des capacités d'entrée des ports.

Exemple

La description du multiplexeur contiendra les informations suivantes :

CA = CB = 0.4 pF
CC = 0.2 pF

TPHL = 3 ns + 2.5 ns/pF
TPLH = ...

La spécification du temps de propagation depuis le niveau haut jusqu'au niveau bas de la sortie étant constituée
- d'une partie fixe , indépendante de l'environnement (3 ns),
- d'une partie variable (2.5 ns par pF attaqué).

ii) <u>approche proposée</u>

Par l'utilisation du paramétrage et de la notion d'affectation de chronogrammes, on peut aisément prendre en compte les paramètres indépendants de l'environnement ; une description possible du multiplexeur est donnée figure 28.

```
RGF mux_gen (PARAM tp : INTEGER) ;
  ENTREE
    a,b,c : BOOLEAN ;
  SORTIE
    o : BOOLEAN ;
  FONCTION  /sous forme textuelle /
    ACTION
       p0 : o := CHOIX c DANS
                    Z,U : U ;
                    0 : [(U,0)(a,tp)] ;
                    1 : [(U,0)(b,tp)] ;
                FINCHOIX ;
    GRAPHE
       t0 : p0 - p0 : V : CHANGE (a,b,c) ;
                 / partie condition toujours vraie /
    INIT p0 ;
FIN mux_gen
```

Figure 28

le problème qui se pose alrs est celui de la prise en compte de l'environnement extérieur : le paramétrage classique étant incapable de le prendre en compte.
L'extension proposée revient à donner la possiblité au concepteur de définir un certain nombre d'informations associées aux ports d'Entrée/Sortie et d'utiliser des fonctions de manipulation de ces informations.

*** informations supplémentaires**

Ces informations seront données lors de la déclaration des ports de la ressource considérée sous la forme d'un identificateur et d'une valeur. Cet identificateur pouvant être assimilé à un champ supplémentaire définissant la variable.

Remarque

Si l'on considère une variable simple A (sans extension), le seul champs manipulé est le champs implicite donnant la valeur de cette variable :
A := ‹expr› est équivalent à A.valeur := ‹expr›.

Exemple

Sur l'exemple du muliplexeur on peut envisager la partie déclarative suivante :

```
ENTREE
  a,b !capa_e = 4! : boolean ;
  c !capa_e = 2! : boolean ;
SORTIE
  o !sortance = 200! : boolean ;
```

dans laquelle les informations supplémentaires données par le concepteur sont indiquées entre "!" ; ces champs étant manipulables comme des champs classiques des variables de type enregistrement. Il faut noter que **ces champs ne sont pas prédéfinis**, ce qui ne fige pas le langage à un domaine donné et est compatible avec le caractère fonctionnel de CADOC.LD

*** Manipulation des informations**

Les informations supplémentaires associées à un port d'entrée étant définies, il reste à spécifier les mécanismes d'accès à ces informations. Ces mécanismes se ramènent à la connaissance par une ressource des ressources qui lui seront connectées et donc à la connaissance de la liste des ports reliés à un port donné et à des manipulations sur cette liste.

Dans un premier temps, on peut proposer les fonctions suivantes :

- **connect** (‹nom de port›) : cette fonction retournera la liste des ports connectés au port spécifié.
- **sigma** (‹liste de ports›!‹idf de champs›) : effectuera la somme des valeurs des champs de nom ‹idf de champs› des ports spécifiés dans la liste ; la valeur 0 étant la valeur assumée par défaut.

Exemple

On peut envisager des expressions du type

sigma (connect(o) ! capa_e)

qui donnera la capacité totale à piloter par le port o.

Cette expression pourra être utilisée

- dans une assertion du type

```
assertion1 : sigma(connect(o)!capa_e) > o!sortance
             ecrire ("assertion 1 non vérifiée") ;
```

- dans la partie date d'un chronogramme : la partie fonction du multiplexeur donnée page précédente devenant :

```
o := CHOIX c DANS
          Z,U : U ;
          O : [(U,0)(a,sigma(connect(o)!capa_e))]
          ...
```

Il est à noter que les évaluations de telles expressions seront faites une fois et une seule : lorsque une expression de ce type sera rencontrée pour la première fois en cours de simulation, elle sera évaluée et remplacée par sa valeur.

iii) résultats attendus - remarques

Ce type d'extension du langage permet la modélisation de cellules de bibliothèques de complexités faibles ou moyennes tout en gardant une bonne précision temporelle. sur le plan de l'efficacité, même si aucune donnée numérique n'est encore disponible, il est évident que si le circuit global est composé en majorité de cellules de faible complexité (portes par exemple), le simulateur fonctionnel du système CADOC sera moins efficace qu'un simulateur spécialisé. Par contre, dés que ces cellules seront plus complexes (registres, UAL, ...), on peut prévoir une meilleure compétitivité de la simulation fonctionnelle. Dans tous les cas, la simulation fonctionnelle permet, en outre, la modélisation de l'environnement du circuit.
Dans le cadre plus général de la modélisation fonctionnelle, les extensions précédentes restent justifiées, même dans le cas de descriptions d'abstraction élevé lorsque l'on connait la structure fine du circuit : l'abstraction étant alors souhaitable pour des raisons d'efficacité.

Remarque

Si l'on considère un circuit complexe, la notion de capacité d'entrée reste valable : la capacité d'entrée d'un port étant la capacité du premier dispositif de bas niveau rencontré (la première grille de transistor). De même, la sortance d'un port "abstrait" est la sortance de la porte réelle pilotant ce port.

Cette remarque se trouve confirmée par la constatation que deux descriptions d'un même circuit à deux niveaux d'abstraction différents ont la même enveloppe structurelle (même ports d'E/S) : on peut considérer que la méthode précédente consiste à remonter certaines informations contenues dans une description fine au niveau des ports d'une description abstraite obtenue à partir de la description fine.

V - AUTRE APPROCHE

Toutes les approches précédentes peuvent être qualifiées d'orientées simulation dans le sens où elles permettent la modélisation précise d'un certain nombre de types de circuits, ces modélisation étant destinées à être simulées ; en contrepartie de la richesse syntaxique de ces langages, il semble difficile de les utiliser pour faire de la preuve de circuits. L'approche proposée dans [GOR 81] consiste à définir le comportement d'un circuit par l'intermédiaire d'un ensemble d'équations, la syntaxe du langage étant restreinte pour permettre des manipulations mathématiques des modèles.

Nous allons présenter ici quelques exemples de modélisations, un développement ainsi que des références à des approches similaires se trouve dans le chapitre consacré aux langages fonctionnels appliqués à la description du matériel (partie 1, paragraphe II.2).

Un circuit est défini dans [GOR 81] par son comportement, la forme générale d'une définition de comportement étant la suivante :

$$F(x,y,\ldots) = \lambda(i_1,\ldots,i_m).\ \{o_1 = E_1,\ldots,\ o_n = E_n\},\ F(E_x,\ E_y,\ldots)$$

- $F(x,y,\ldots)$: nom du modèle
- $x,y,\ldots$: variables d'état
- $i_1,\ldots,i_m$: noms des lignes d'entrée
- $o_1 = E_1,\ldots,\ o_n = E_n$: noms des lignes de sortie et expressions permettant de les calculer
- $E_x,\ E_y,\ldots$: expression donnant le nouvel état (prochain coup d'horloge)

Remarque

Un modèle de circuit fait référence à la notion d'état et les changements d'états se font sur variations d'une horloge de base implicite.

Exemple

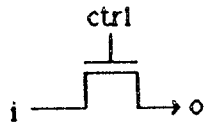
Le comportement d'un multiplexeur (cd : ligne de contrôle, e1 et e2 : entrées, S : sortie) est donné par :

$$MUX = \lambda\ \{cd, e1, e2\}\ .\ \{S = (cd \rightarrow e1,e2)\},\ MUX$$

(la notation cd → e1,e2 se lisant "si cd alors e1 sinon e2")

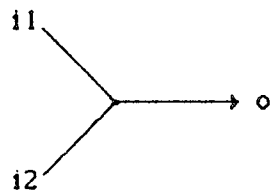
Pour la description de dispositifs MOS, l'auteur définit d'abord l'ensemble des valeurs manipulées {0, 1, Z} et l'extension des opérations classiques sur cet ensemble ; un dispositif MOS est ensuite décrit comme interconnexion des dispositifs primitifs suivants :

porte unidirectionnelle (figure 29a),
connexion multiterminal (figure 29b),
transistor de charge (figure 29c),
masse (figure 29d).



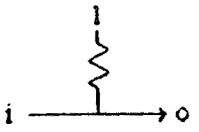
$$P(t) = \lambda\{i,ctrl\}.\{o = (ctrl = 1 \text{ ou } (ctrl = Z \text{ et } t = 1\ ) \rightarrow i,Z)\},\ G(ctrl)$$

Figure 29a : porte unidirectionnelle



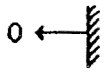
$$CM = \lambda\ \{i1,i2\}\ .\ \{o = i1 \text{ ou } i2\},\ J$$

Figure 29b : connexion multiterminal



PU = L {i}.{o = (i = 0 → 0,1)}, PU

Figure 29c : transistor de charge (Pull-Up)



GND = λ{}.{o = 0}, GND

Figure 29d : masse

Remarques

- Le seul effet de PU est de transformer Z en 1, ce qui n'a pas forcément un sens suivant les technologies.

- De même que le modèle de porte est unidirectionnel, la connexion multiterminal est orientée.

A partir de ces dispositifs de base on peut alors construire par structuration des portes plus complexes et prouver par des mécanismes d'expansion, de remplacement et de réduction l'équivalence entre un modèle "fonctionnel" et une réalisation structurée.

Dans le cas de la porte NOR donné en figure 30, on montre que le comportement (les notations explicitant l'interconnexion des dispositifs)

NOR (t1, t2) = [P(t1)| P(t2) | GND | CM |PU]/11 12 13 14

est équivalent au comportement défini par

NOR (t1, t2) = λ{i1, i2} . {0 = non(i1 ou i2)}, NOR (i1, i2)

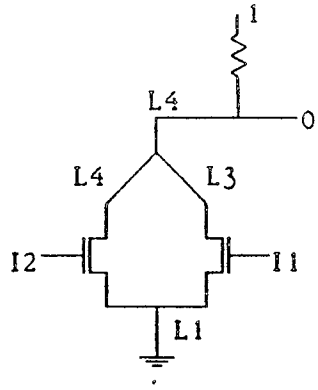


Figure 30 : porte NOR

Remarque

- L'intérêt de telles approches est la possibilité de manipulation mathématique des comportements et donc la preuve d'équivalence entre 2 comportements décrits différemment. Ces approches "orientées preuve" ne sont, en contrepartie, pas adaptées à la simulation ou à la modélisation précise de bas niveau (forte unidirectionnalité, manipulation explicite du temps difficile).

VI - CONCLUSION DE LA TROISIEME PARTIE

Une étude rapide de la technologie MOS nous a permis de dégager un certain nombre de problèmes liés à cette technologie ; pour les prendre en compte, certains langages proposent une extension de l'ensemble des objets primitifs, cette extension justifiée dans le cas des langages structurels classiques est par contre contradictoire avec la philosophie des langages fonctionnels ; un exemple de contradiction de ce type est donnée par le langage CAP/DSDL.

Nous avons montré comment le langage CADOC.LD est capable de traiter ces problèmes sans introduction d'une telle contradiction ; il nous faut signaler ici que les modèles de certains dispositifs de bas niveaux étant complexes, il est nécessaire de ne les utiliser, sous risque de pénaliser les temps de simulation, que lorsque cela est "nécessaire".

Pour terminer, ce chapitre nous permet de fixer la limite inférieure d'utilisation du langage CADOC.LD : elle se situe entre les niveaux logique et interrupteur ; tout dispositif à fonctionnement discret (non analogique) étant à priori modélisable en CADOC.LD.

CONCLUSION

Les orientations des langages de description du matériel sont multiples : on peut définir les orientations simulation, vérification/preuve et synthèse ; les primitives des langages étant fonction de ces orientations qui sont mutuellement incompatibles.

L'étude des langages aussi bien spécifiques que non spécifiques nous a permis de souligner des points importants que nous avons essayé de prendre en compte lors de la définition du langage CADOC.LD ; les objectifs des langages non spécifiques étant différents de ceux de CADOC, certaines caractéristiques importantes n'ont malheureusement pas pu être considérées (en particulier dans le domaine de la preuve du matériel qui nécessite, du moins actuellement, un langage à syntaxe restrictive et peu adaptée à la modélisation en vue de simulation).

Le but de CADOC.LD est double : c'est d'abord un langage de description utilisable pour la modélisation de circuits déja conçus et c'est ensuite un langage de spécification permettant de modéliser un même circuit tout au long de son processus de conception, depuis les niveaux d'abstraction élevée, jusqu'à un niveau proche de sa réalisation matérielle.
Nous avons montré que ce langage est aussi bien adapté aux modélisations abstraites, efficaces lors de simulations, qu'aux modélisations de bas niveaux. La gamme des niveaux couverts permet au langage de s'adapter aussi bien aux descriptions de dispositifs complexes (microprocesseurs) qu'aux dispositifs de faible complexité (cellules de bibliothèques d'éléments prédiffusés ou précaractérisés : portes, registres,...). Rappelons que par définition des langages fonctionnels, les descriptions à tous niveaux se font en utilisant une même syntaxe, ce qui signifie en particulier que l'utilisateur doit faire l'apprentissage d'un seul langage et non d'une série de langages.

La dernière partie de cette étude s'est intéressée aux dispositifs de bas niveaux, dont le comportement est souvent lié à une technologie donnée ; nous avons montré qu'une description fonctionnelle de ces dispositifs était possible sans contradiction avec les caractéristiques de base du langage et que la modélisation de ces dispositifs n'était pas synonyme d'absence de finesse dans le domaine temporel.

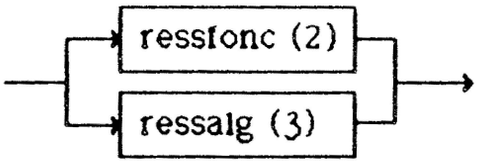
ANNEXES

CARTE SYNTAXIQUE DU LANGAGE CADOC.LD
TABLE DES MOTS RESERVES

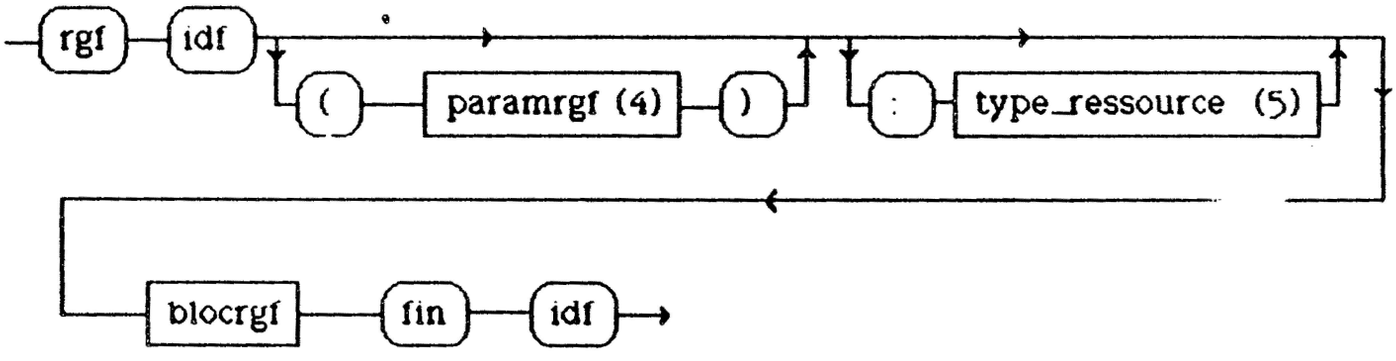
EXEMPLES DE DESCRIPTIONS

RESULTATS DE SIMULATIONS

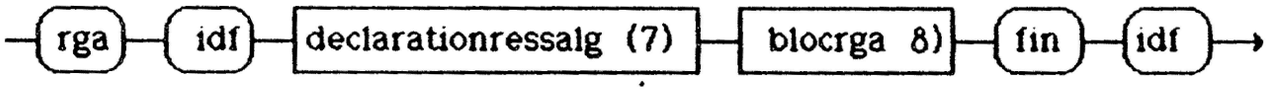
1 : ressource



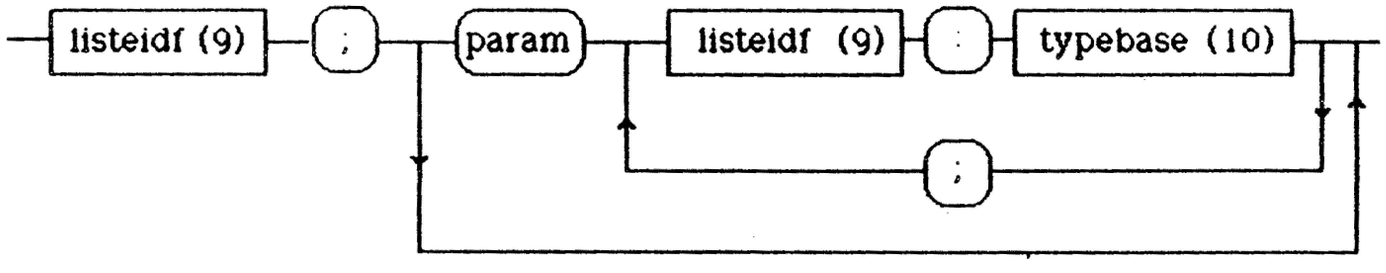
2 : ressfonc



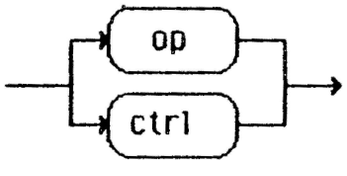
3 : ressalg



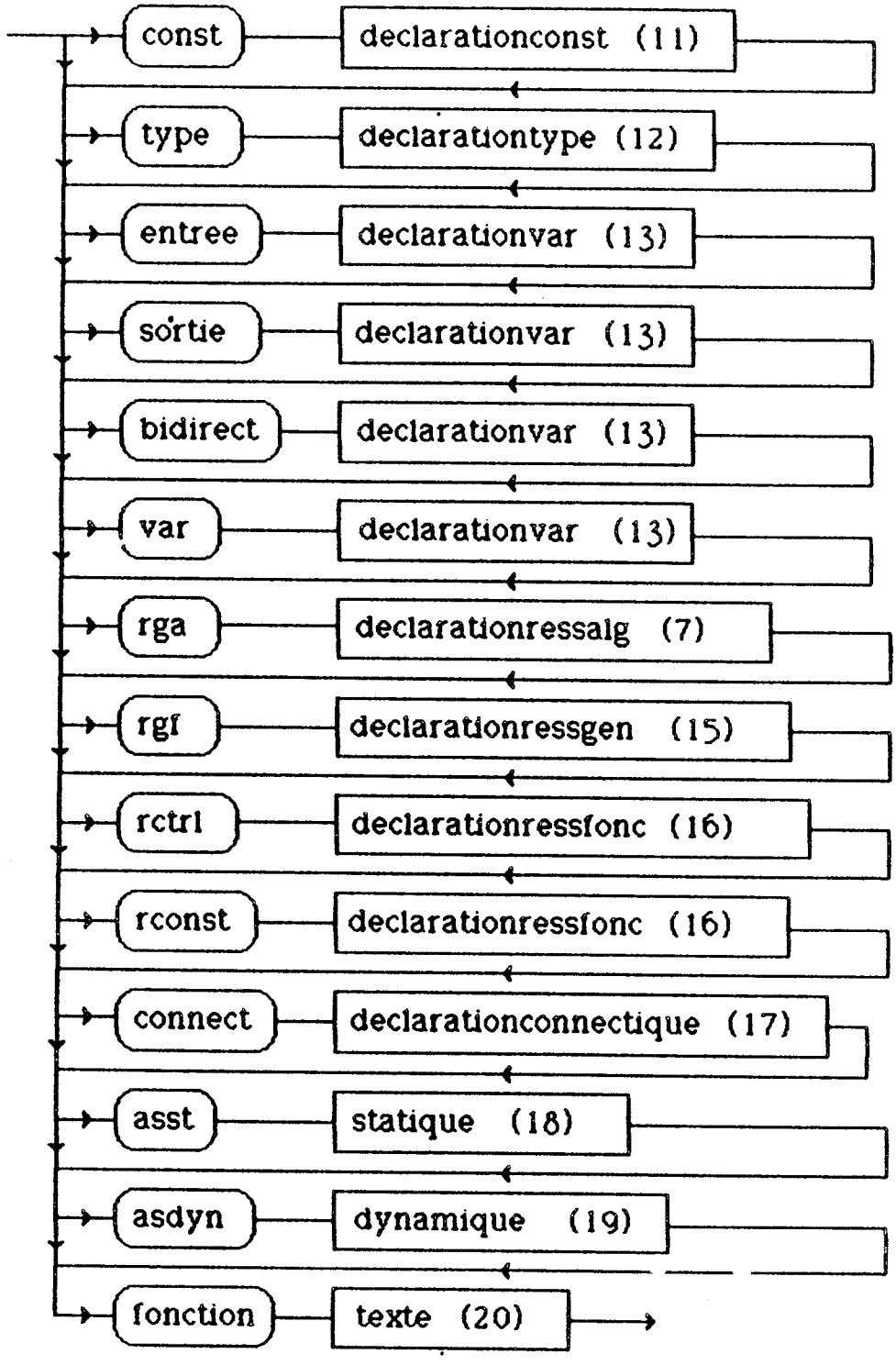
4 : paramrgf



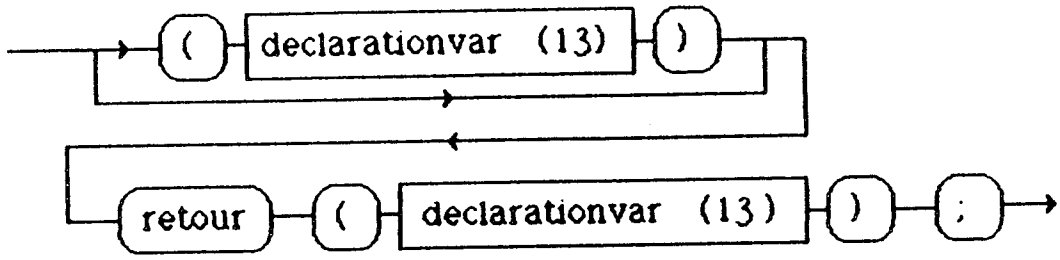
5 : type_ressource



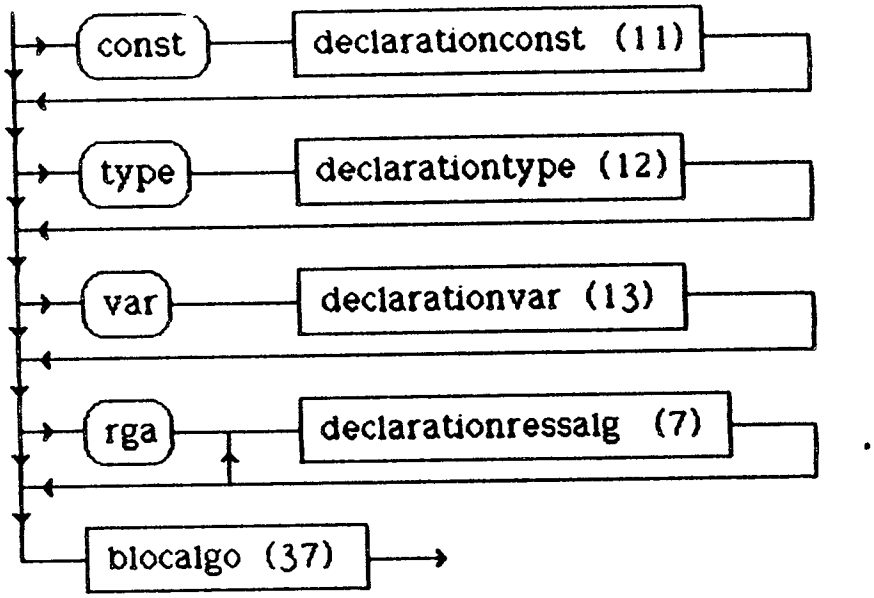
6 : blocrgf


const
declarationconst (11)
type
declarationtype (12)
entree
declarationvar (13)
sortie
declarationvar (13)
bidirect
declarationvar (13)
var
declarationvar (13)
rga
declarationressalg (7)
rgf
declarationressgen (15)
rctrl
declarationressfonc (16)
rconst
declarationressfonc (16)
connect
declarationconnectique (17)
asst
statique (18)
asdyn
dynamique (19)
fonction
texte (20)

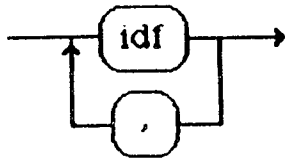
7 : declarationressalg



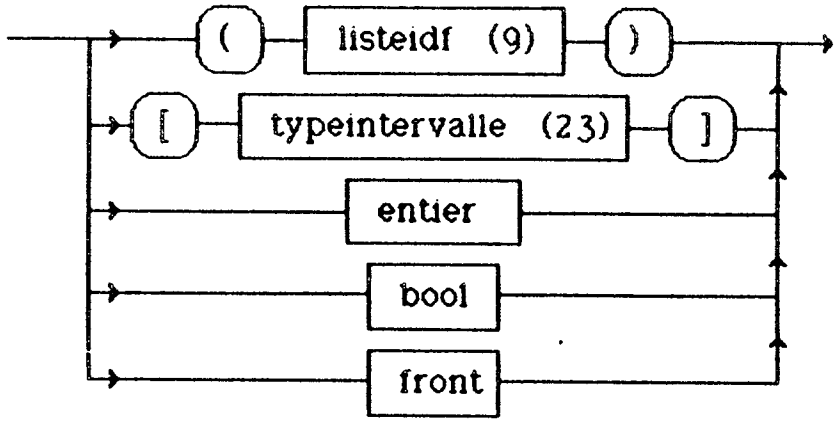
8 : blocrga



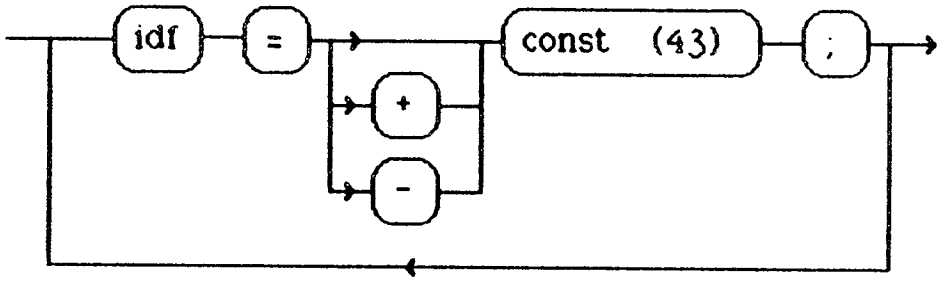
9 : listeidf



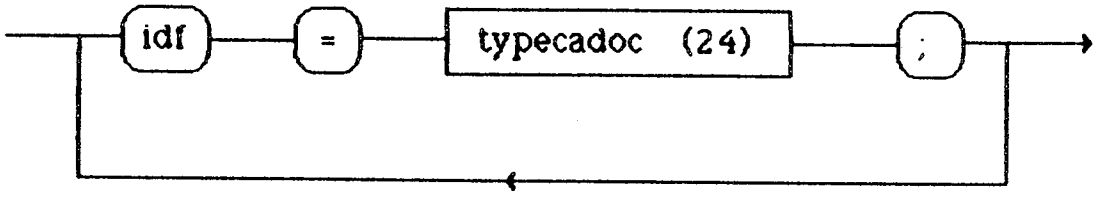
10 · typebase



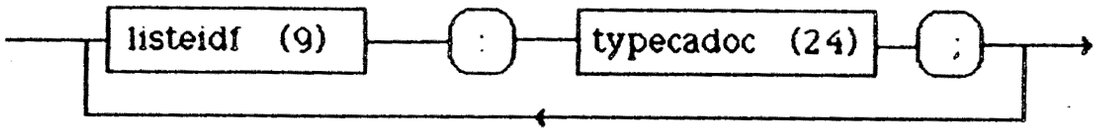
11 : declarationconst



12 : declarationtype

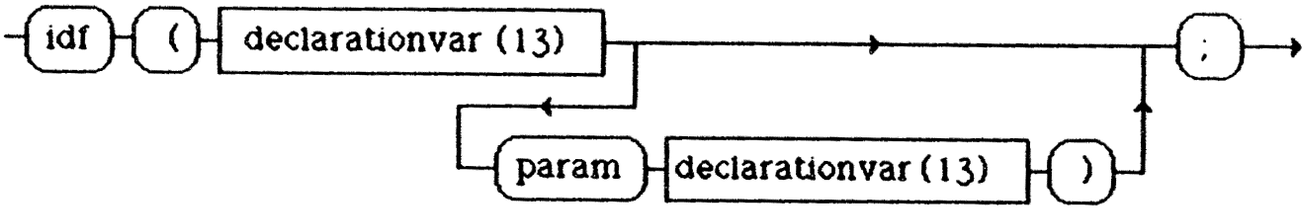


13 : declarationvar

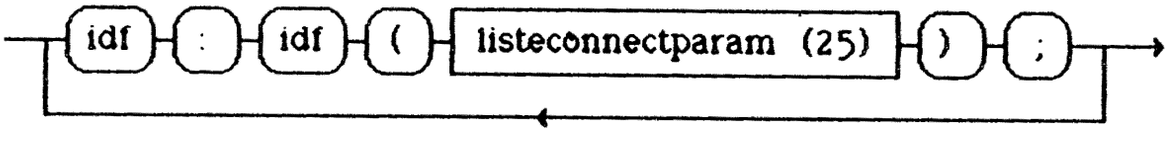


14 : règle non utilisée

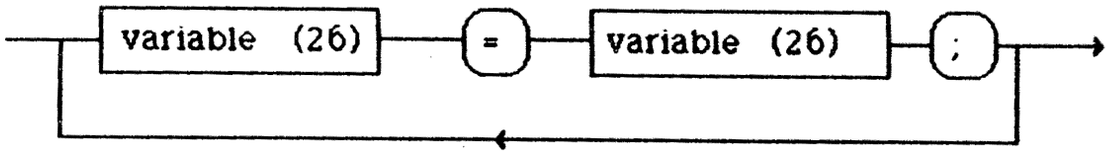
15 : declarationressgen



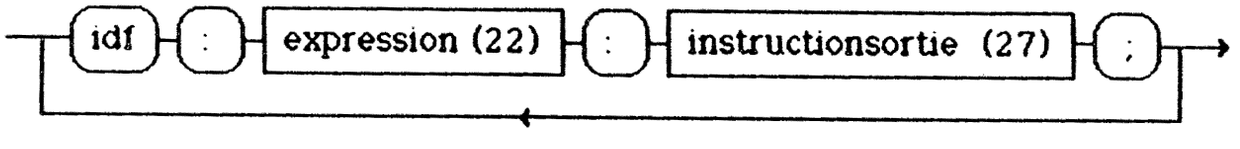
16 : declarationressfonc



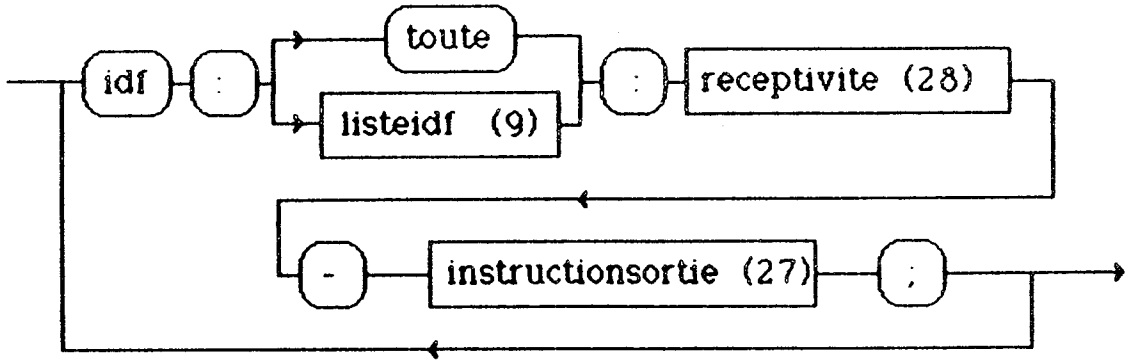
17 : declarationconnectique



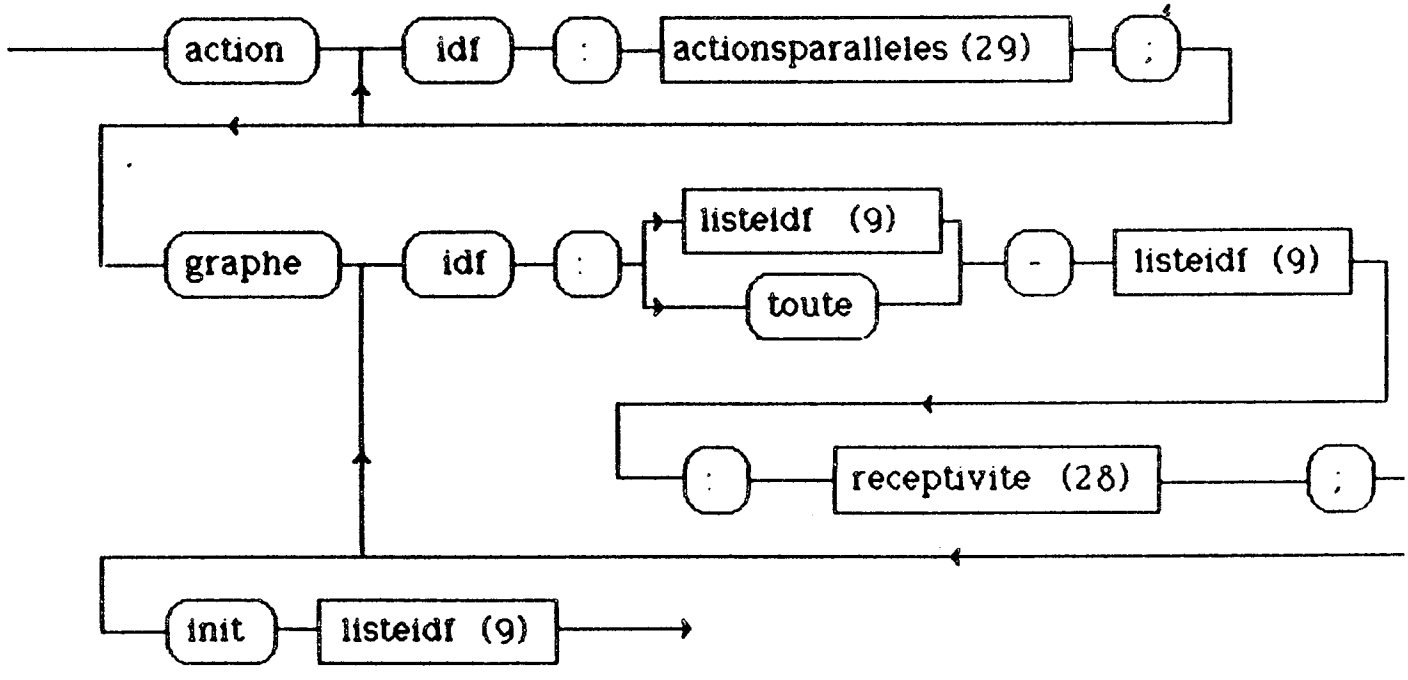
18 : statique



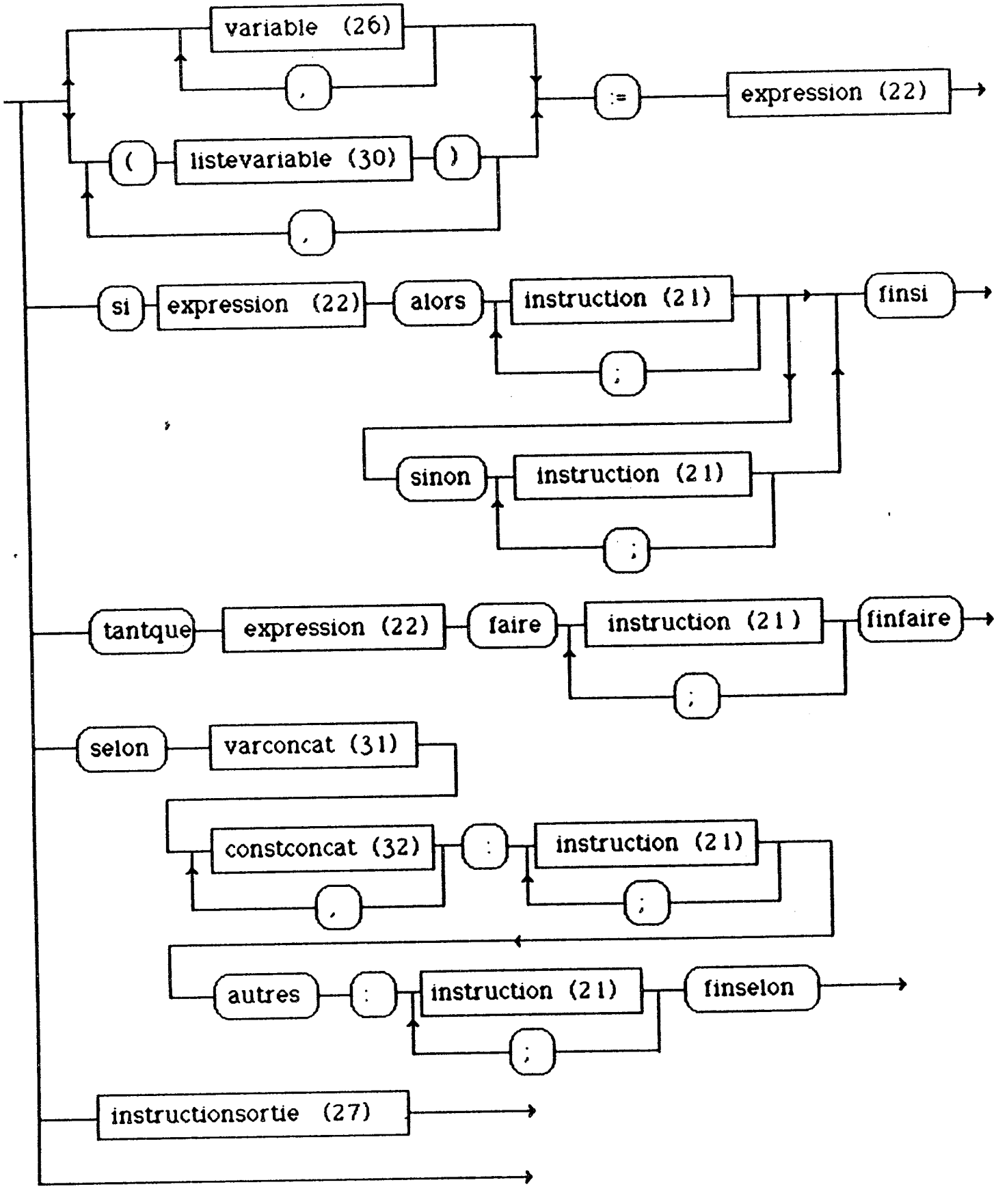
19 : dynamique



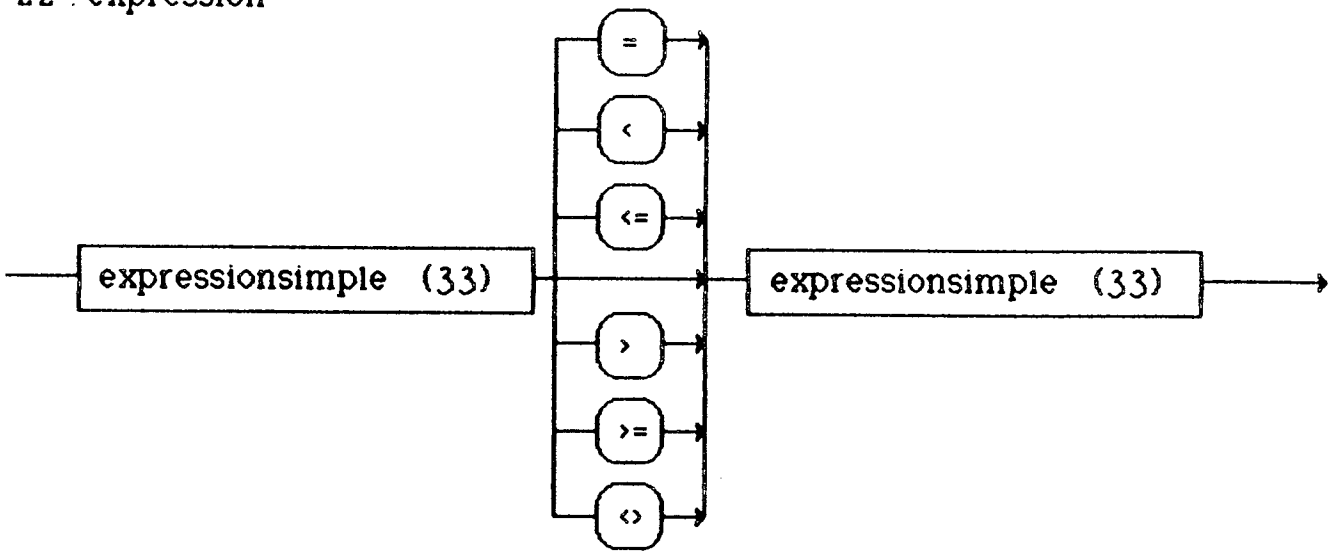
20 : texte



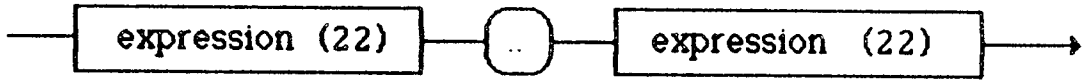
21 : instruction



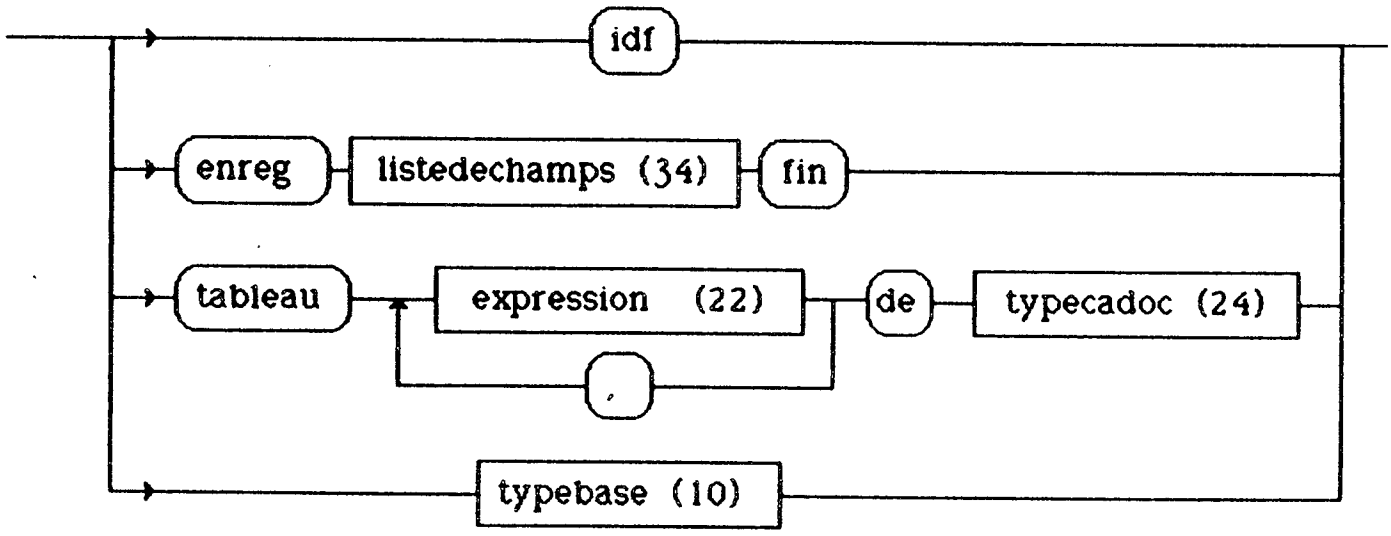
22 : expression



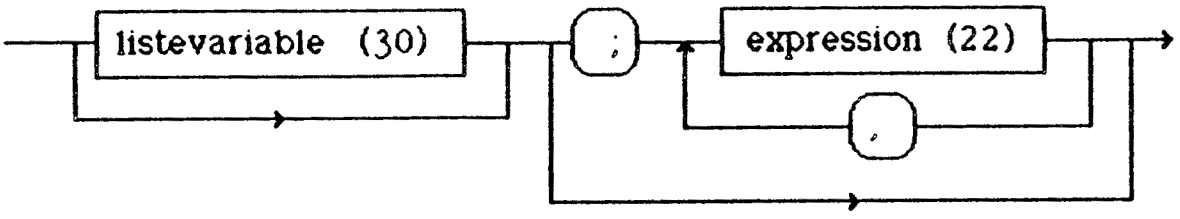
23 : typeintervalle



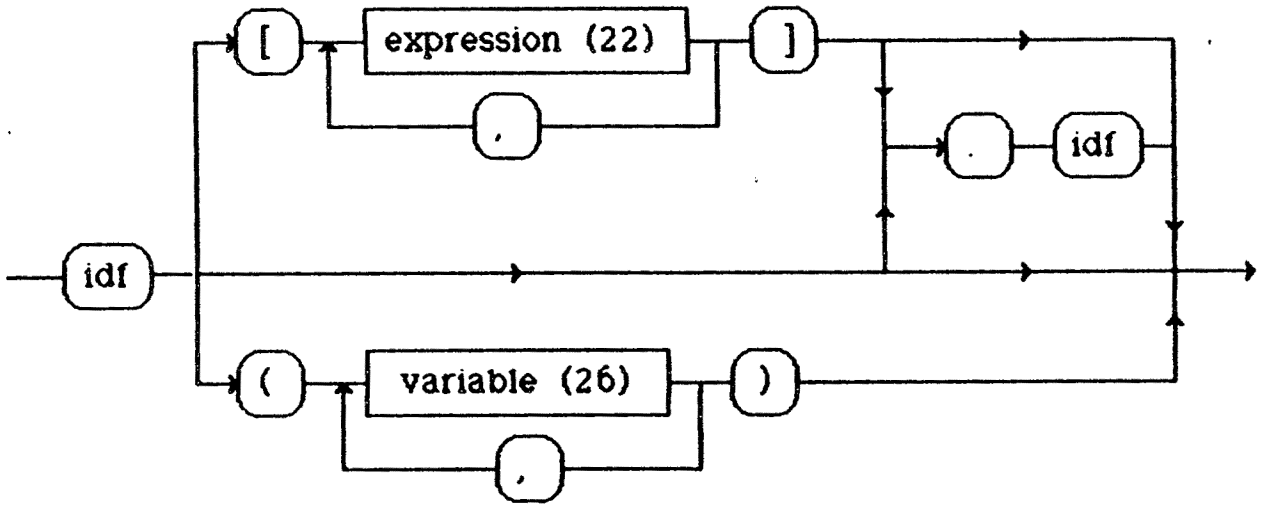
24 : typecadoc



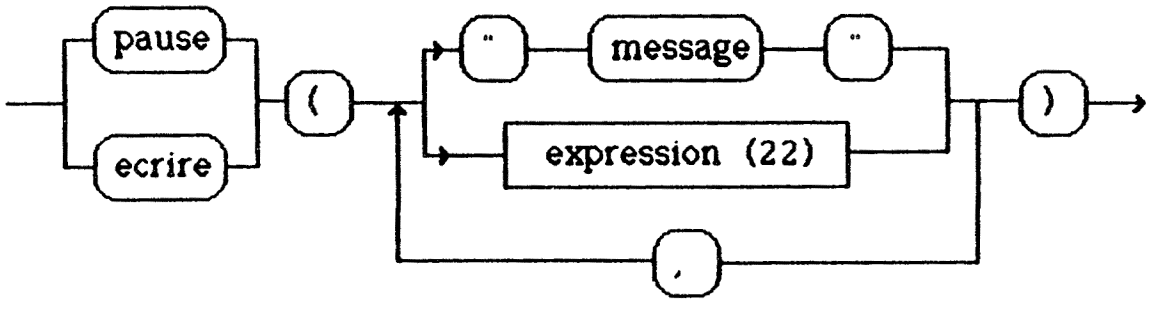
25 : listeconnectparam



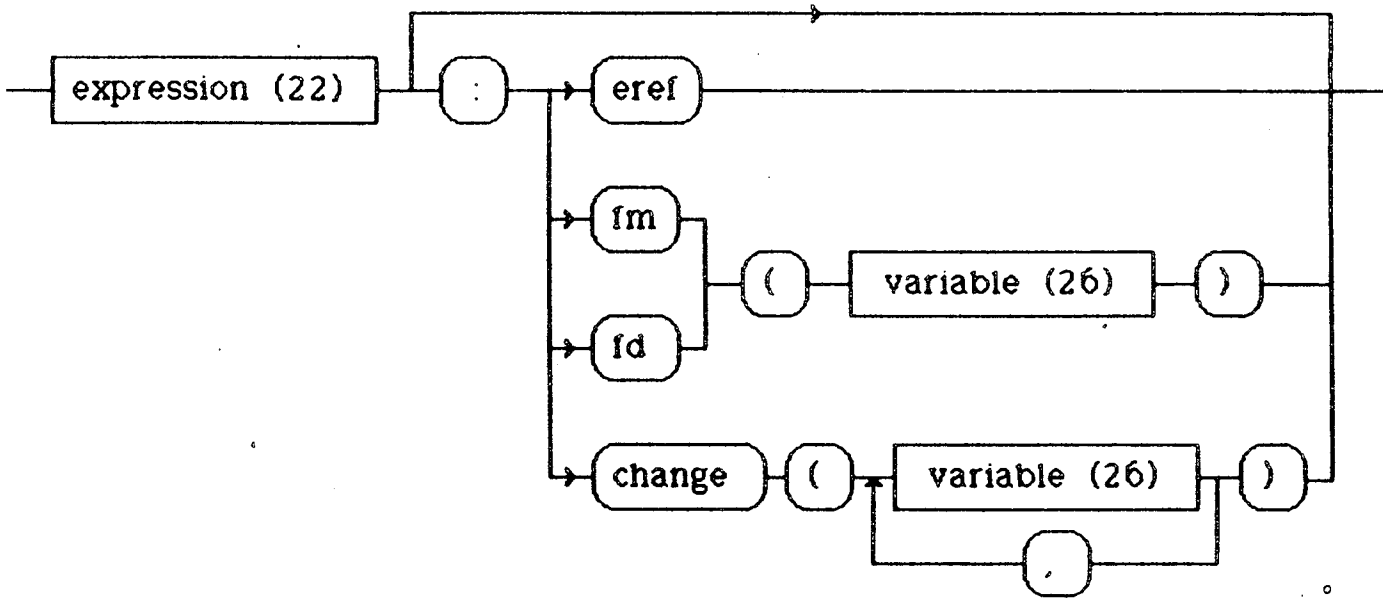
26 : variable



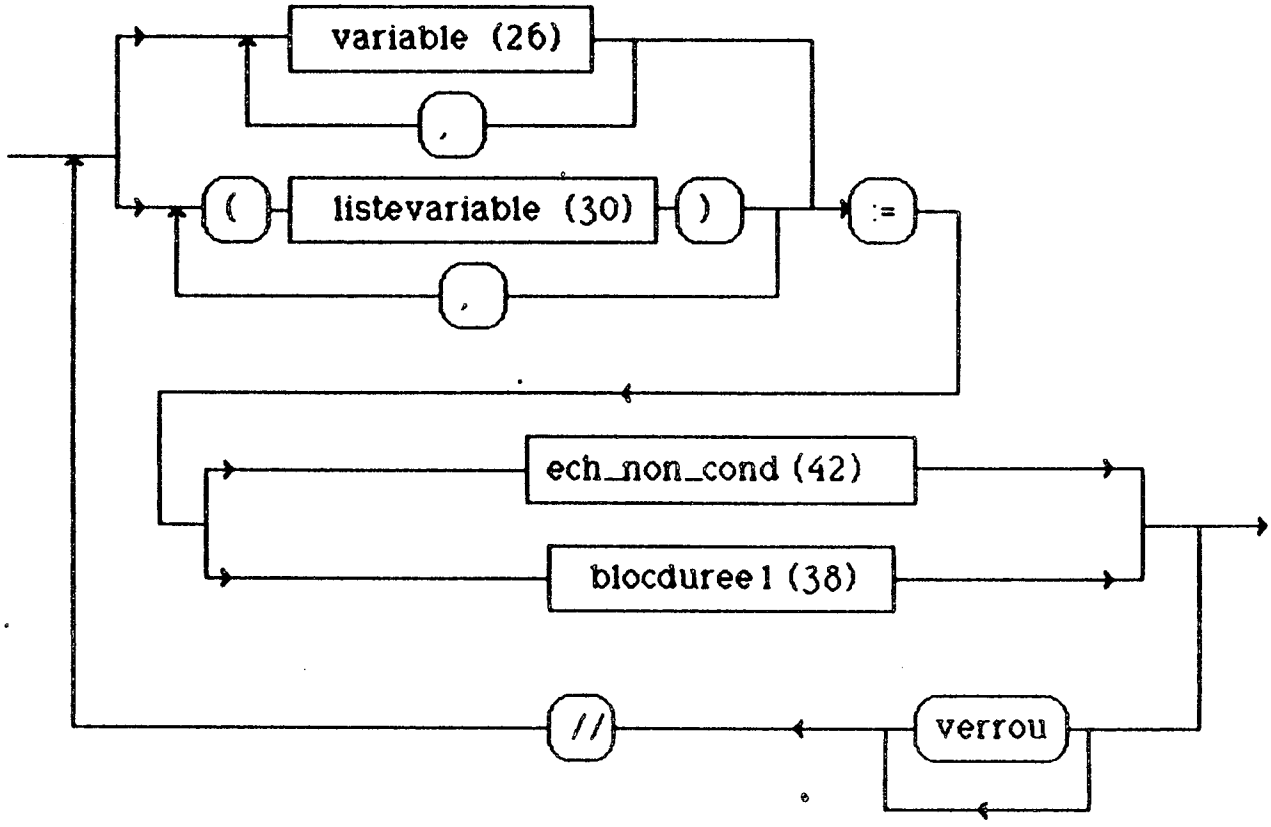
27 : instructionsortie



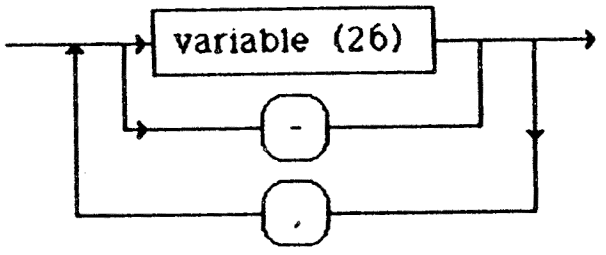
28 : receptivite



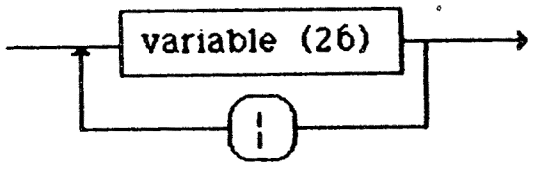
29 : actionsparalleles



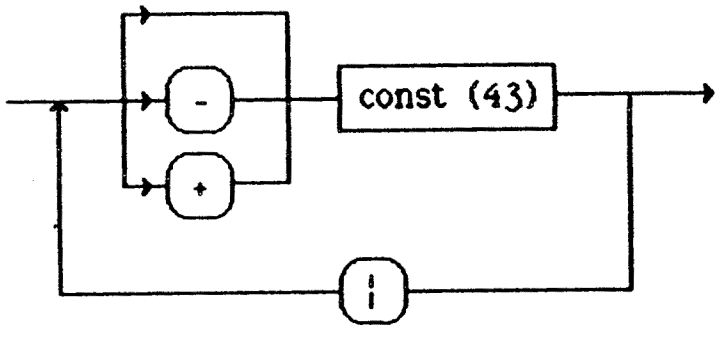
30 : listevariable



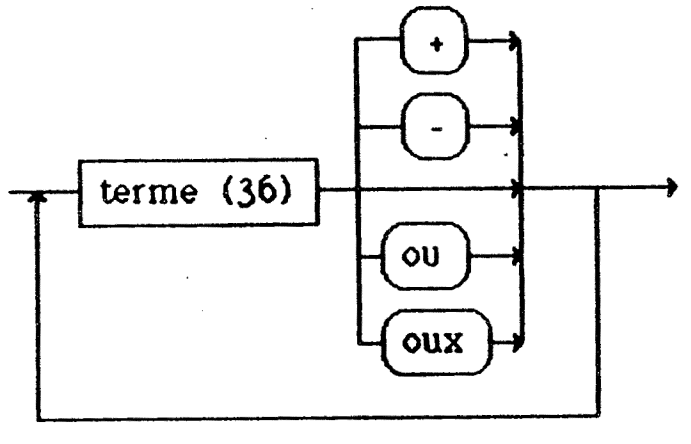
31 : varconcat



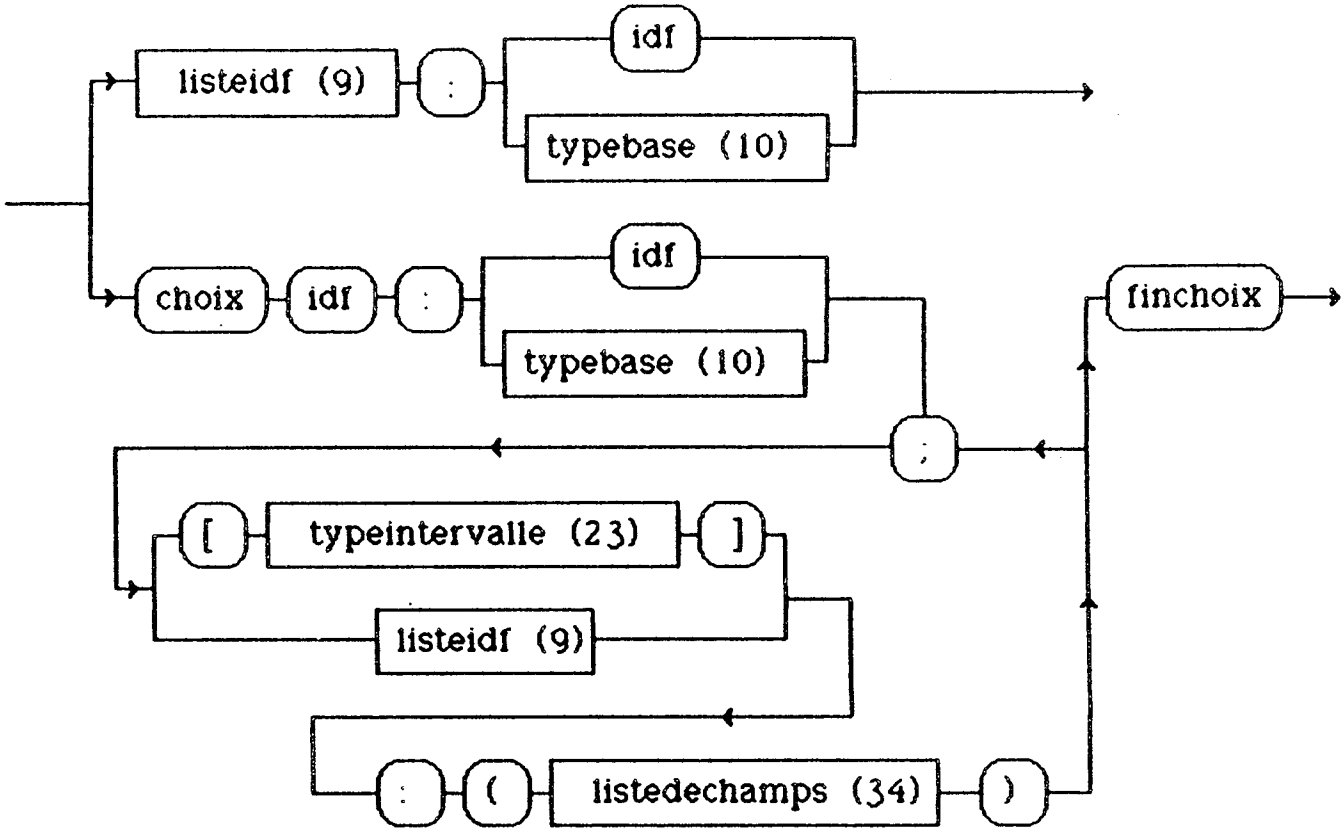
32 : constconcat



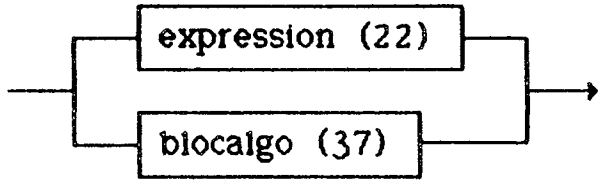
33 : expressionsimple



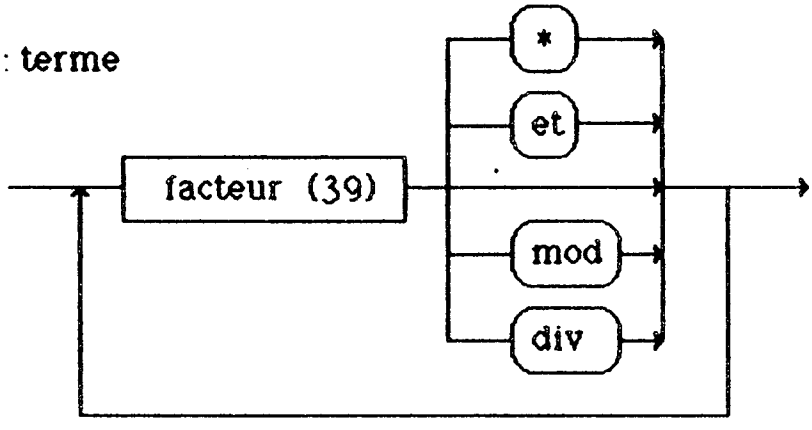
34 : listedechamps



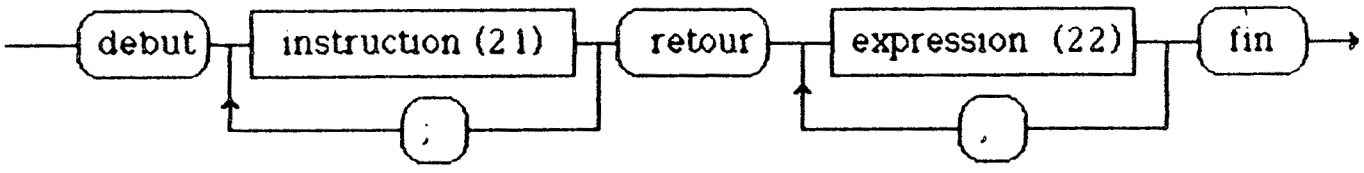
35 : algo



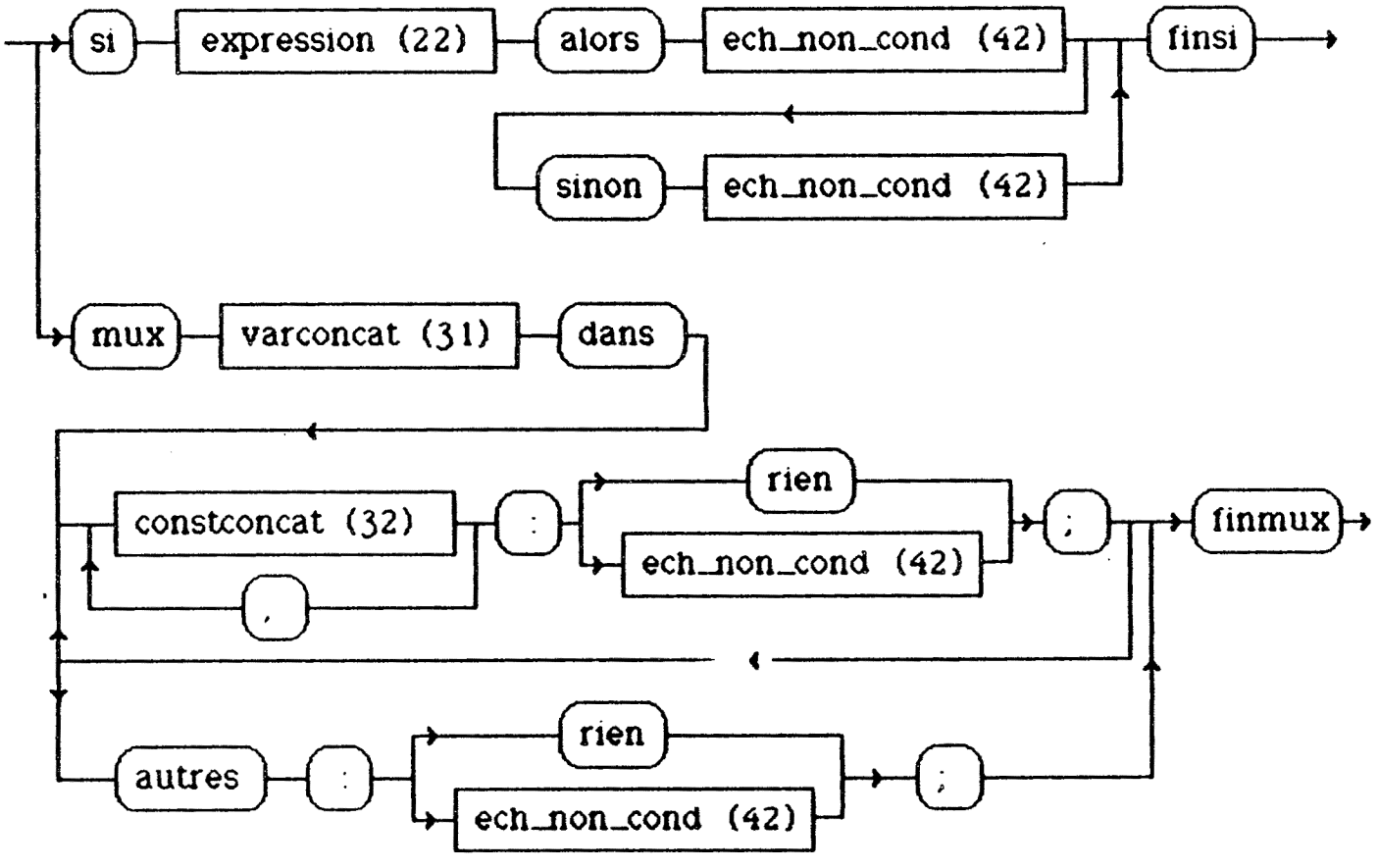
36 : terme



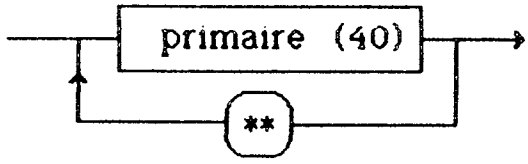
37 : blocalgo



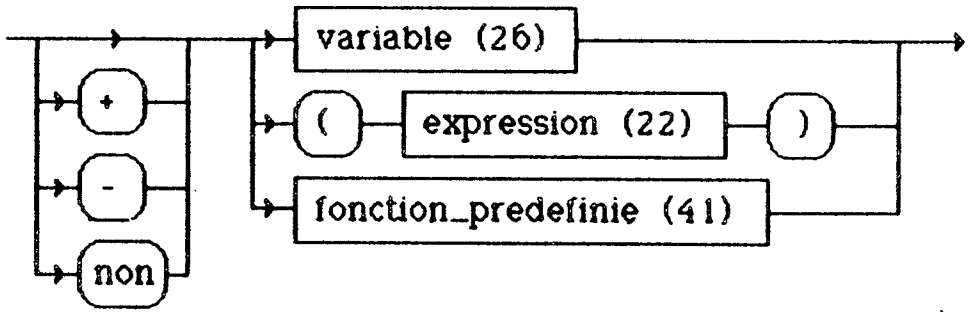
38 : blocduree1



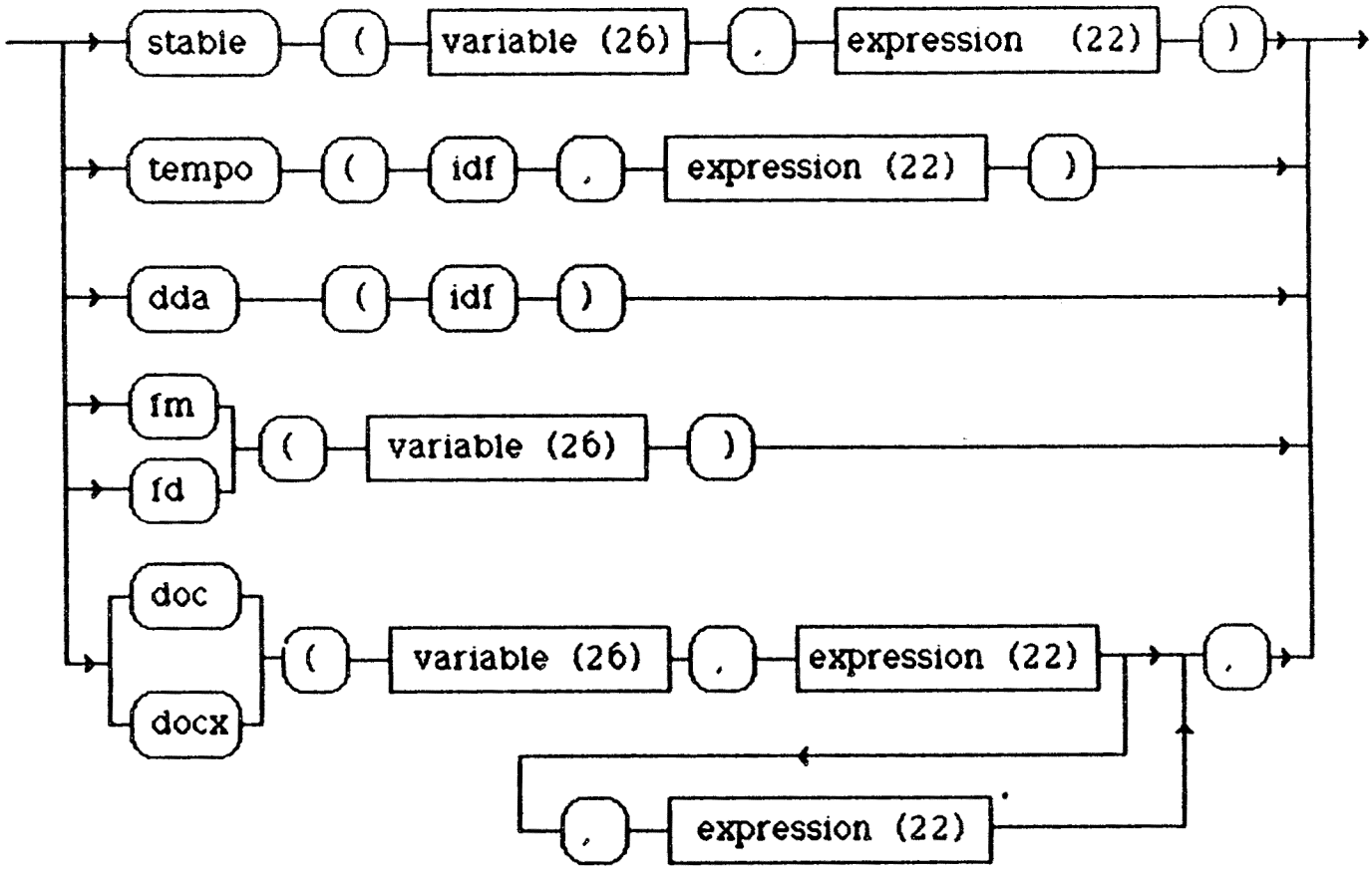
39 : facteur



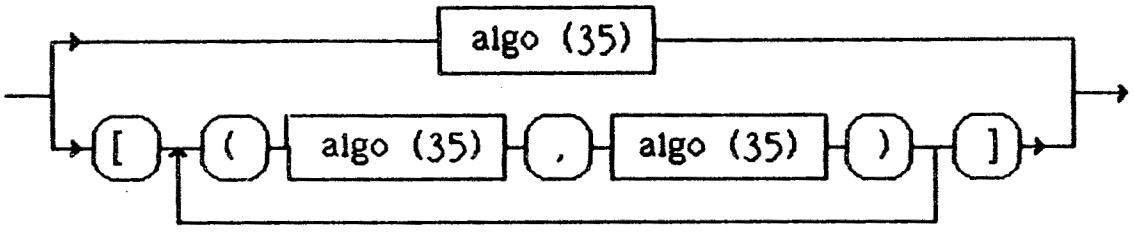
40 : primaire



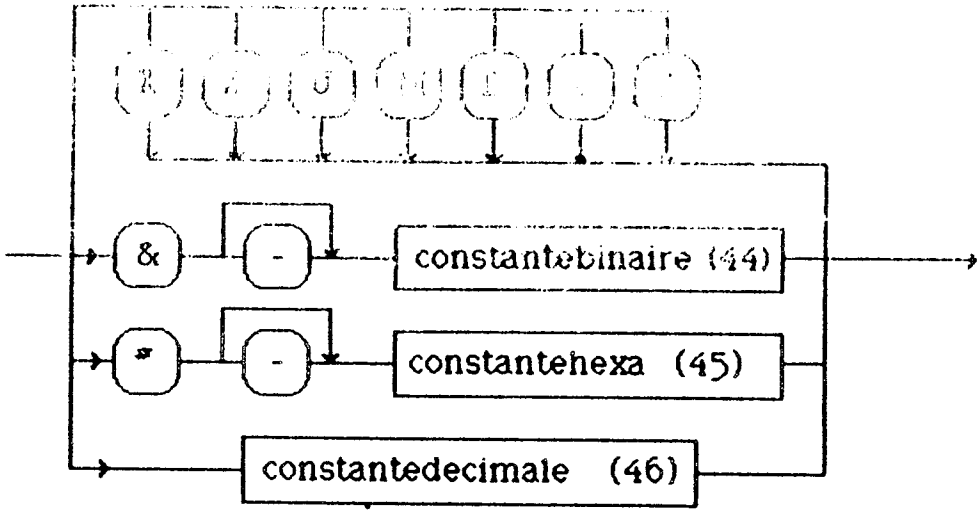
41 : fonction_predefinie



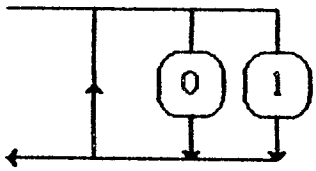
42 : ech_non_cond



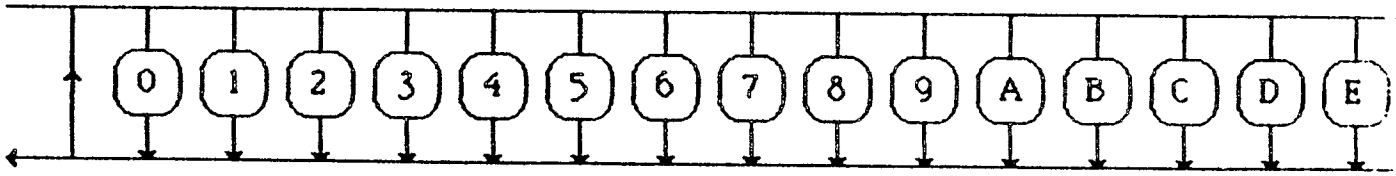
43 : const



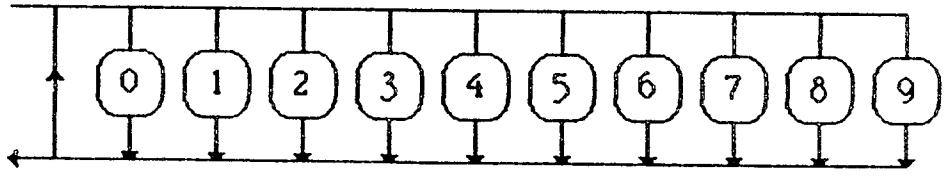
44 : constantebinaire



45 : constantehexa



46 : constantedecimale



MOTS RESERVES		SYNONYMES	
FRANCAIS	ANGLAIS	FRANCAIS	ANGLAIS
action	action		
alors	then		
asst	stass	assertions_statiques	static_assertions
autres	otherwise		
bidir	bidir	bidirect	
bool	bool	booleen	
change	change		
choix	case		
connect	connect	connectique	
const	const		
ctrl	ctrl	controleur	controller
dda	lda		
dans	in	parmi	in
de	of		
debut	begin		
depuis	since		
div	div		
doc	loc		
docx	locx		
ecrire	print		
enreg	record		
entier	integer		
entree	input		
eref	reft		
et	and		
faire	do		
fchoix	ecase	finchoix	endcase
fd	fe		
ffaire	edo	finfaire	enddo
fin	end		
fm	re		
fmux	emux	finmux	endmux
.fonction	function		
front	edge		
fselon	echoice	finselon	endchoice
fsi	esi	finsi	endif

MOTS RESERVES		SYNONYMES	
FRANCAIS	ANGLAIS	FRANCAIS	ANGLAIS
graphe	graph		
init	init		
jusqua	to		
mux	mux		
mod	mod		
non	not		
op	op	operateur	operator
ou	or		
oux	xor		
param	param		
pas	step		
pause	pause		
rconst	compr	ressources_constituantes	component_resources
rctrl	ctrlr	ressources_controlees	controlled_ resources
retour	return		
rga	gar	ressource_generique _algorithmique	algorithmic_resource
rgf	gfr	ressource_generique _fonctionnelle	generic_functional _resource
rien	nop		
selon	choice		
si	if		
sinon	else		
sortie	output		
stable	stable		
tableau	array		
tantque	while		
tempo	tempo		
toute	all		
type	type		
var	var		
varint	intvar	interne	internal
verrou	locked		

EXEMPLES

```
RGF REG_GEN : OP ;

 TYPE
   INT_4 = [0..15] ;
   INT_5 = [0..31] ;

 ENTREE
   LOAD_REG_GEN : FRONT ;
   SHIFT_REG_GEN : FRONT ;
   OE_REG_GEN, RAZ_REG_GEN : BOOL ;
   IN_REG_GEN : INT_5 ;

 SORTIE
   S3_REG_GEN : INT_4 ;
   S_REG_GEN : INT_4 ;
   BIT0_REG_GEN : [0..1] ;

 VARINT
   INT_REG_GEN : INT_5 ;

 FONCTION
 ACTION
   P0_BUS_REG_GEN : S3_REG_GEN := MUX OE_REG_GEN DANS
                                    F : Z ;
                                    T : INT_REG_GEN ;
                                        AUTRES : U ;
                                  FMUX ;
   P1_BUS_REG_GEN : S3_REG_GEN := INT_REG_GEN ;
   P0_REG_GEN : INT_REG_GEN, BIT0_REG_GEN, S_REG_GEN := U ;
   P1_REG_GEN : INT_REG_GEN, BIT0_REG_GEN, S_REG_GEN := 0 ;
   P2_REG_GEN : ;
   P3_REG_GEN : INT_REG_GEN := IN_REG_GEN //
                BIT0_REG_GEN := IN_REG_GEN MOD 2 //
                S_REG_GEN := IN_REG_GEN ;
   P4_REG_GEN : INT_REG_GEN := INT_REG_GEN DIV 2 //
                BIT0_REG_GEN := (INT_REG_GEN DIV 2) MOD 2 //
                S_REG_GEN := INT_REG_GEN DIV 2 ;

 GRAPHE
   TBUS0_REG_GEN : P0_BUS_REG_GEN - P0_BUS_REG_GEN : V : CHANGE (OE_REG_GEN) ;
   TBUS1_REG_GEN : P0_BUS_REG_GEN - P1_BUS_REG_GEN :
                   (OE_REG_GEN = T) : CHANGE (INT_REG_GEN) ;
   TBUS2_REG_GEN : P1_BUS_REG_GEN - P1_BUS_REG_GEN :
                   (OE_REG_GEN = T) : CHANGE (INT_REG_GEN) ;
   TBUS3_REG_GEN : P1_BUS_REG_GEN - P0_BUS_REG_GEN : (OE_REG_GEN <> T) ;
   T1_REG_GEN : P0_REG_GEN - P1_REG_GEN : RAZ_REG_GEN ;
   T2_REG_GEN : P1_REG_GEN - P2_REG_GEN : NON RAZ_REG_GEN ;
   T3_REG_GEN : P2_REG_GEN - P1_REG_GEN : RAZ_REG_GEN ;
   T4_REG_GEN : P2_REG_GEN - P3_REG_GEN : NON RAZ_REG_GEN : FM (LOAD_REG_GEN) ;

   T5_REG_GEN : P3_REG_GEN - P2_REG_GEN : V ;
   T6_REG_GEN : P2_REG_GEN - P4_REG_GEN : NON RAZ_REG_GEN : FM (SHIFT_REG_GEN) ;

   T7_REG_GEN : P4_REG_GEN - P2_REG_GEN : V ;
   T8_REG_GEN : P2_REG_GEN - P2_REG_GEN : V : CHANGE (OE_REG_GEN) ;

 INIT
   P0_REG_GEN, P0_BUS_REG_GEN ;

FIN REG_GEN
```

```
RGF REG_MQ : OP ;

 TYPE
   INT_4 = [0..15] ;

 ENTREE
   LOAD_REG_MQ : FRONT ;
   SHIFT_REG_MQ : FRONT ;
   OE_REG_MQ, RAZ_REG_MQ : BOOL ;
   CIN_REG_MQ : [0..1] ;

 SORTIE
   BIT0_REG_MQ : [0..1] ;

 BIDIR
   IN_OUT_REG_MQ : INT_4 ;

 VARINT
   INT_REG_MQ : INT_4 ;


 FONCTION

 ACTION
   P0_BUS_REG_MQ : IN_OUT_REG_MQ := MUX OE_REG_MQ DANS
                                      F : Z ;
                                      T : INT_REG_MQ ;
                                      AUTRES : U ;
                                    FMUX ;
   P1_BUS_REG_MQ : IN_OUT_REG_MQ := INT_REG_MQ ;

   P0_REG_MQ : INT_REG_MQ, BIT0_REG_MQ := U ;
   P1_REG_MQ : INT_REG_MQ, BIT0_REG_MQ := 0 ;
   P2_REG_MQ : ;
   P3_REG_MQ : INT_REG_MQ := IN_OUT_REG_MQ //
               BIT0_REG_MQ := IN_OUT_REG_MQ MOD 2 ;
   P4_REG_MQ : INT_REG_MQ := CIN_REG_MQ * 8 + INT_REG_MQ DIV 2 //
               BIT0_REG_MQ  := (CIN_REG_MQ * 8 + (INT_REG_MQ DIV 2)) MOD 2 ;

 GRAPHE
   TBUS0_REG_MQ : P0_BUS_REG_MQ - P0_BUS_REG_MQ : V : CHANGE (OE_REG_MQ) ;
   TBUS1_REG_MQ : P0_BUS_REG_MQ - P1_BUS_REG_MQ :
                  (OE_REG_MQ = T) : CHANGE (INT_REG_MQ) ;
   TBUS2_REG_MQ : P1_BUS_REG_MQ - P1_BUS_REG_MQ :
                  (OE_REG_MQ = T) : CHANGE (INT_REG_MQ) ;
   TBUS3_REG_MQ : P1_BUS_REG_MQ - P0_BUS_REG_MQ : (OE_REG_MQ <> T) ;

   T1_REG_MQ : P0_REG_MQ - P1_REG_MQ : RAZ_REG_MQ ;
   T2_REG_MQ : P1_REG_MQ - P2_REG_MQ : NON RAZ_REG_MQ ;
   T3_REG_MQ : P2_REG_MQ - P1_REG_MQ : RAZ_REG_MQ ;
   T4_REG_MQ : P2_REG_MQ - P3_REG_MQ : NON RAZ_REG_MQ : FM (LOAD_REG_MQ) ;
   T5_REG_MQ : P3_REG_MQ - P2_REG_MQ : V ;
   T6_REG_MQ : P2_REG_MQ - P4_REG_MQ : NON RAZ_REG_MQ : FM (SHIFT_REG_MQ) ;
   T7_REG_MQ : P4_REG_MQ - P2_REG_MQ : V ;
   T8_REG_MQ : P2_REG_MQ - P2_REG_MQ : V : CHANGE (OE_REG_MQ) ;

 INIT
   P0_BUS_REG_MQ, P0_REG_MQ ;
FIN REG_MQ
```

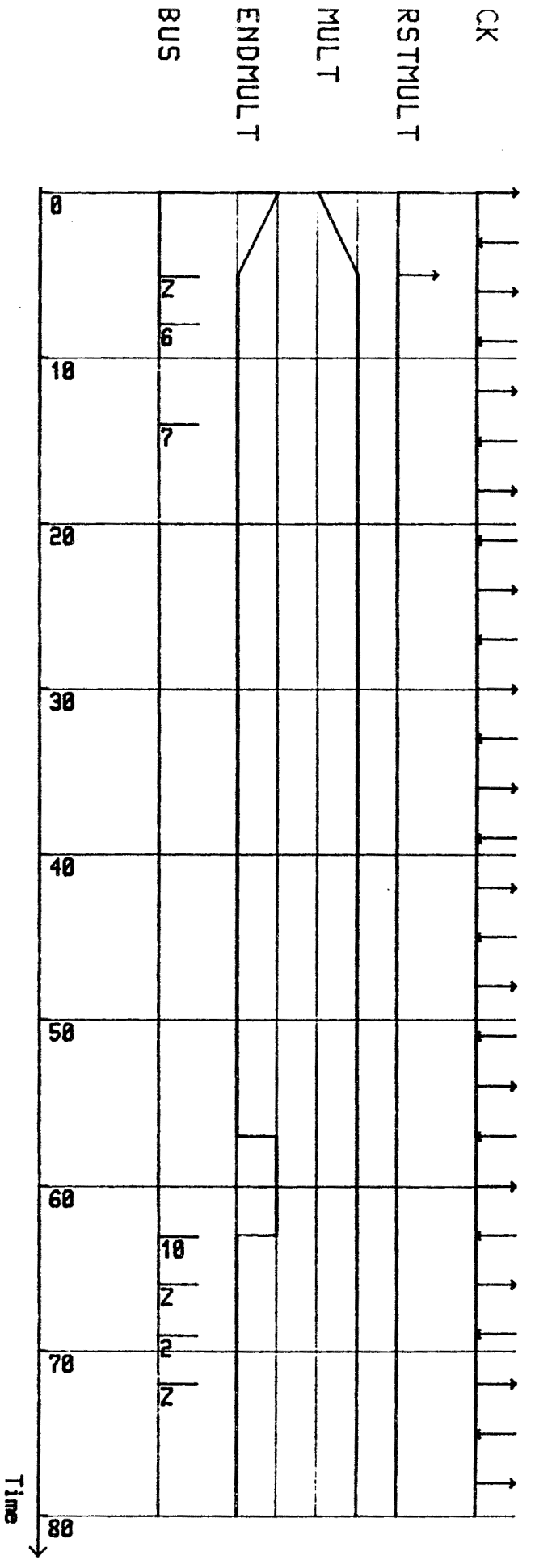
```
RGF REG_SIMPLE : OP ;

 ENTREE
   IN_REG_SIMPLE : [0..15] ;
   LOAD_REG_SIMPLE : FRONT ;

 SORTIE
   S_REG_SIMPLE : [0..15] ;


 FONCTION

 ACTION
   P0_REG_SIMPLE : S_REG_SIMPLE := U ;
   P1_REG_SIMPLE : S_REG_SIMPLE := IN_REG_SIMPLE ;

 GRAPHE
   T0IREG_SIMPLE : P0_RPG_SIMPLE - P1_REG_SIMPLE : V : FM (LOAD_REG_SIMPLE) ;
   T1_REG_SIMPLE . P1_REG_SIMPLE - P1_REG_SIMPLE : V : FM (LOAD_REG_SIMPLE) ;

 INIT
   P0_REG_SIMPLE ;

FIN REG_SIMPLE
```

```
RGF CPTR_GEN : OP ;

 ENTREE
   LOAD_CPTR_GEN : FRONT ;
   DECR_CPTR_GEN  : FRONT ;

 SORTIE
   Z_CPTR_GEN : BOOL ;

 VARINT
   INT_CPTR_GEN  : ENTIER ;


 FONCTION

 ACTION
   P0_CPTR_GEN : Z_CPTR_GEN, INT_CPTR_GEN := U ;
   P1_CPTR_GEN : Z_CPTR_GEN := F // INT_CPTR_GEN := 4 ;
   P2_CPTR_GEN : INT_CPTR_GEN := INT_CPTR_GEN -1 //
                 Z_CPTR_GEN := SI INT_CPTR_GEN = 1 ALORS T SINON F FSI ;

 GRAPHE

   T1_CPTR_GEN  : P0_CPTR_GEN - P1_CPTR_GEN  : V : FM (LOAD_CPTR_GEN) ;
   T2_CPTR_GEN  : P1_CPTR_GEN - P2_CPTR_GEN  : V : FM (DECR_CPTR_GEN) ;
   T3_CPTR_GEN  : P2_CPTR_GEN - P1_CPTR_GEN  : V : FM (LOAD_CPTR_GEN) ;
   T4_CPTR_GEN  : P2_CPTR_GEN - P2_CPTR_GEN  : V : FM (DECR_CPTR_GEN) ;

 INIT
   P0_CPTR_GEN ;

FIN CPTR_GEN
```

```
RGF ADD_GEN : OP ;

 ENTREE
   E1_ADD_GEN , E2_ADD_GEN  : [0..15] ;

 SORTIE
   S_ADD_GEN  : [0..31] ;


 FONCTION

 ACTION
   PO_ADD_GEN : S_ADD_GEN := E1_ADD_GEN + E2_ADD_GEN ;

 GRAPHE
   T1_ADD_GEN : PO_ADD_GEN - PO_ADD_GEN : V : CHANGE (E1_ADD_GEN, E2_ADD_GEN) ;

 INIT
   PO_ADD_GEN ;

FIN ADD_GEN
```

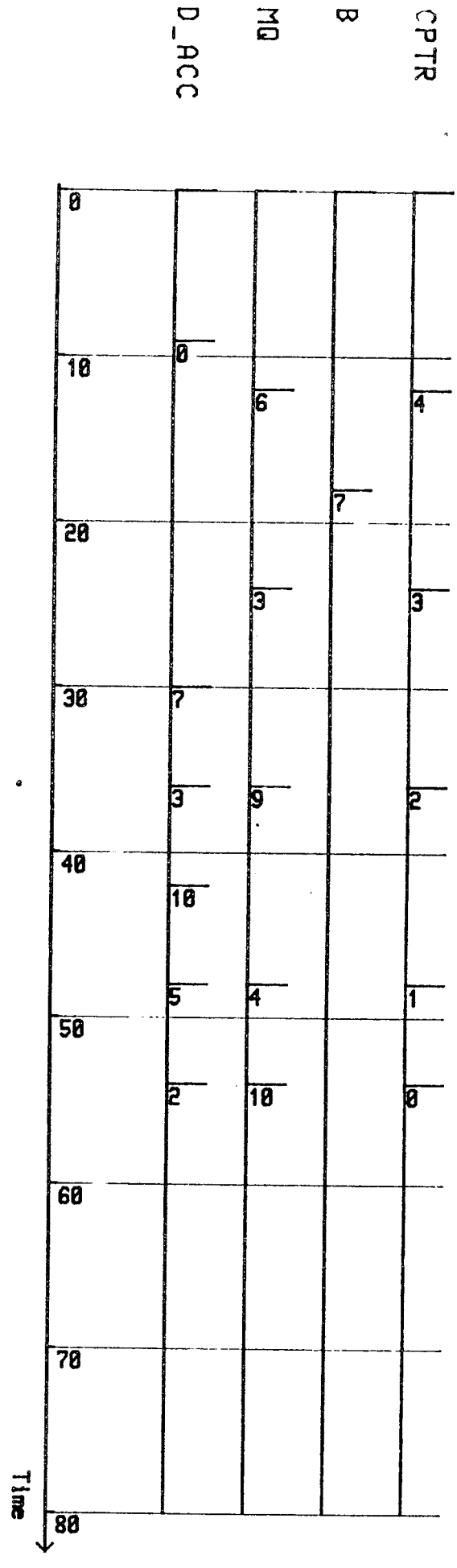
```
RGF MULT_PC : OP ;

 ENTREE
   CK : FRONT ;
   RSTMULT : BOOL ;
   MULT : BOOL ;
   MQ0 : [0..1] ;
   Z_CPTR : BOOL ;

 SORTIE
   OE_DACC, OE_MQ : BOOL ;
   ENDMULT : BOOL ;
   LOAD_CPTR, LOAD_MQ, LOAD_B : FRONT ;
   LOAD_DACC, SHIFT, DECR_CPTR : FRONT ;
   RAZ : BOOL ;

 BIDIR
   BUS : [0..15] ;


 FONCTION

 ACTION
   PI : ;
   P0 : OE_DACC, OE_MQ := F // ENDMULT := F // RAZ := [(T,0)(F,FM(CK))] ;
   P1 : LOAD_CPTR := [(M,FM(CK))] //
        LOAD_MQ := [(M,FM(CK))] ;
   P2 : LOAD_B := [(M,FM(CK))] ;
   P3 : LOAD_DACC := [(M,FM(CK))] ;
   P4 : SHIFT := [(M, FM(CK))] //
        DECR_CPTR := [(M , FM (CK))] ;
   P5 : ENDMULT := V ;
   P6 : OE_MQ := [(T,0)(F,FM(CK))] // ENDMULT := F ;
   P7 : OE_DACC := [(T,0)(F,FM(CK))] ;

 GRAPHE
   T0  : PI - P0 : V : FM (RSTMULT) ;
   T1  : P0 - P1 : MULT : FD (CK) ;
   T2  : P1 - P2 : MULT : FD (CK) ;
   T3  : P2 - P3 : ((MQ0 MOD 2) = 1 ) : FD (CK) ;
   T31 : P2 - P4 : ((MQ0 MOD 2) <> 1) : FD (CK) ;
   T4  : P3 - P4 : V : FD (CK) ;
   T5  : P4 - P4 : (Z_CPTR <> T) ET ((MQ0 MOD 2) <> 1) : FD (CK) ;
   T6  : P4 - P3 : (Z_CPTR <> T) ET ((MQ0 MOD 2) = 1 ) : FD (CK) ;
   T7  : P4 - P5 : (Z_CPTR = T) : FD (CK) ;
   T8  : P5 - P6 : MULT : FD (CK) ;
   T9  : P6 - P7 : MULT : FD (CK) ;
   T10 : P7 - P0 : MULT : FD (CK) ;

 INIT
   PI ;

FIN MULT_PC
```

circuit MULTB : résultats de simulation (ports d'E/S)



CADOC SYSTEM
Circuit : multb

circuit MULTB : résultat de simulation (variables internes)



CADOC SYSTEM
Circuit : multb

circuit MULTB : résultats de simulation (trace chronologique)

```
SIMULATION RESULTS
******************

stimuli file = multb.dep

          0          0    CK---------------------------------          M EXT
          3          0    CK---------------------------------          D EXT
          5          0    RSTMULT-----------------------------          M EXT
          5          0    MULT--------------------------------          1 EXT
          5          1    ENDMULT-----------------------------          0
          5          1    BUS---------------------------------          Z
          6          0    CK---------------------------------          M EXT
          8          0    BUS---------------------------------          6 EXT
          9          0    CK---------------------------------          D EXT
          9          1    D_ACC-------------------------------          0
         12          0    MQ----------------------------------          6
         12          0    CPTR--------------------------------          4
         12          0    CK---------------------------------          M EXT
         14          0    BUS---------------------------------          7 EXT
         15          0    CK---------------------------------          D EXT
         18          0    B-----------------------------------          7
         18          0    CK---------------------------------          M EXT
         21          0    CK---------------------------------          D EXT
         24          0    CPTR--------------------------------          3
         24          0    MQ----------------------------------          3
         24          0    D_ACC-------------------------------          0
         24          0    CK---------------------------------          M EXT
         27          0    CK---------------------------------          D EXT
         30          0    D_ACC-------------------------------          7
         30          0    CK---------------------------------          M EXT
         33          0    CK---------------------------------          D EXT
         36          0    CPTR--------------------------------          2
         36          0    MQ----------------------------------          9
         36          0    D_ACC-------------------------------          3
         36          0    CK---------------------------------          M EXT
         39          0    CK---------------------------------          D EXT
         42          0    D_ACC-------------------------------         10
         42          0    CK---------------------------------          M EXT
         45          0    CK---------------------------------          D EXT
         48          0    CPTR--------------------------------          1
         48          0    MQ----------------------------------          4
         48          0    D_ACC-------------------------------          5
         48          0    CK---------------------------------          M EXT
         51          0    CK---------------------------------          D EXT
         54          0    CPTR--------------------------------          0
         54          0    MQ----------------------------------         10
         54          0    D_ACC-------------------------------          2
         54          0    CK---------------------------------          M EXT
         57          0    CK---------------------------------          D EXT
         57          1    ENDMULT-----------------------------          1
         60          0    CK---------------------------------          M EXT
         63          0    CK---------------------------------          D EXT
         63          1    ENDMULT-----------------------------          0
         63          1    BUS---------------------------------         10
         66          0    BUS---------------------------------          Z
         66          0    CK---------------------------------          M EXT
```

```
69          0     CK--------------------------------          D EXT
69          1     BUS-------------------------------- 2
72          0     BUS--------------------------------          Z
72          0     CK--------------------------------          M EXT
75          0     CK--------------------------------          D EXT
75          1     ENDMULT----------------------------- 0
75          1     BUS--------------------------------          Z
78          0     CK---------------------------------          M EXT
```

circuit MULTB : résultats de simulation (histoire des places)

TRACE OF THE PLACE HISTORY
****

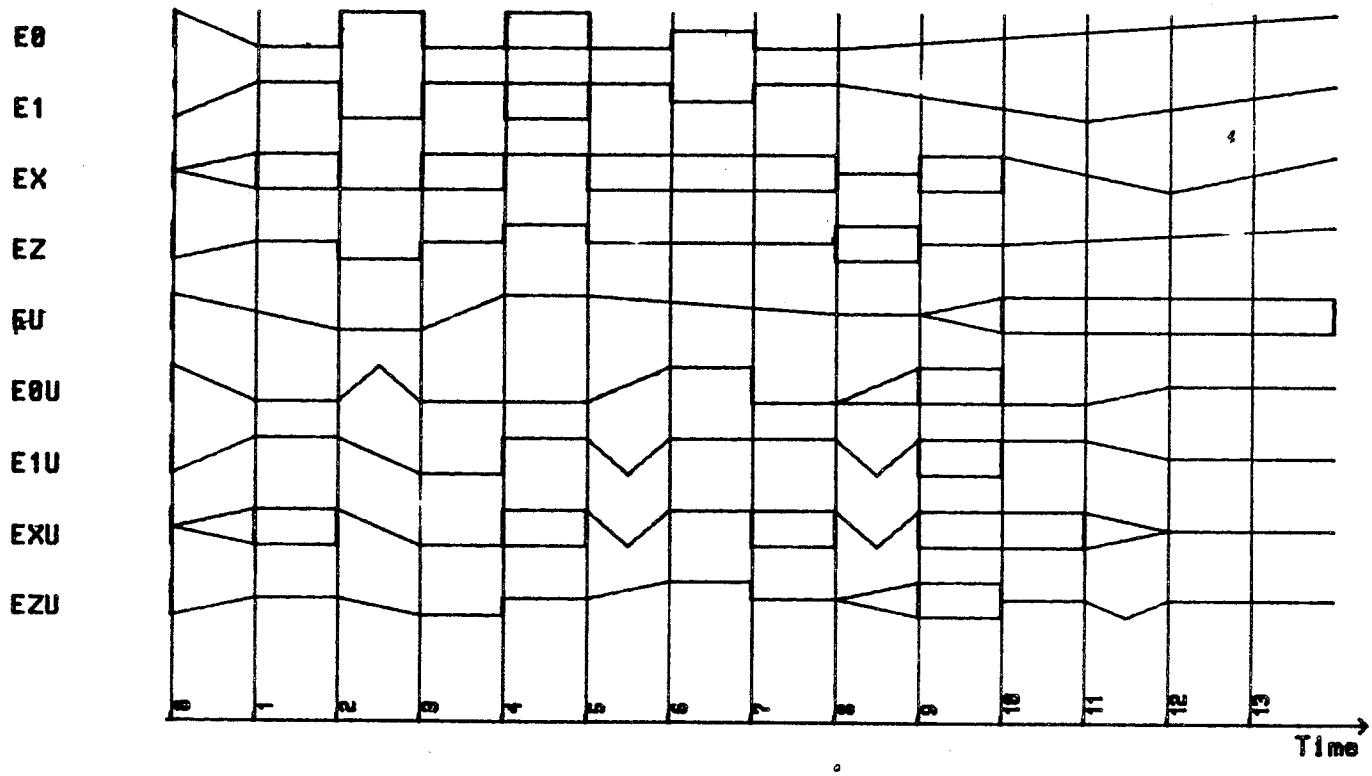
0 -	0	: PI
5 -	1	: P0
9 -	1	: P1
15 -	1	: P2
21 -	1	: P4
27 -	1	: P3
33 -	1	: P4
39 -	1	: P3
45 -	1	: P4
51 -	1	: P4
57 -	1	: P5
63 -	1	: P6
69 -	1	: P7
75 -	1	: P0

circuit MULTB : résultats de simulation (trace par variables)

```
EXHAUSTIVE SIMULATION TRACE
***************************
trace of the variable CK
time :        -1 ,   0 value =            U
time :         0 ,   0 value =            M
time :         3 ,   0 value =            D
time :         6 ,   0 value =            M
time :         9 ,   0 value =            D
time :        12 ,   0 value =            M
time :        15 ,   0 value =            D
time :        18 ,   0 value =            M
time :        21 ,   0 value =            D
time :        24 ,   0 value =            M
time :        27 ,   0 value =            D
time :        30 ,   0 value =            M
time :        33 ,   0 value =            D
time :        36 ,   0 value =            M
time :        39 ,   0 value =            D
time :        42 ,   0 value =            M
time :        45 ,   0 value =            D
time :        48 ,   0 value =            M
time :        51 ,   0 value =            D
time :        54 ,   0 value =            M
time :        57 ,   0 value =            D
time :        60 ,   0 value =            M
time :        63 ,   0 value =            D
time :        66 ,   0 value =            M
time :        69 ,   0 value =            D
time :        72 ,   0 value =            M
time :        75 ,   0 value =            D
time :        78 ,   0 value =            M

 * * * * * * * * * * * * * * * * * * * * * * * *
trace of the variable RSTMULT
time :        -1 ,   0 value =            U
time :         5 ,   0 value =            M

 * * * * * * * * * * * * * * * * * * * * * * * *
trace of the variable MULT
time :        -1 ,   0 value =            U
time :         5 ,   0 value =            1

 * * * * * * * * * * * * * * * * * * * * * * * *
trace of the variable ENDMULT
time :        -1 ,   0 value =            U
time :         5 ,   1 value =            0
time :        57 ,   1 value =            1
time :        63 ,   1 value =            0
time :        75 ,   1 value =            0

 * * * * * * * * * * * * * * * * * * * * * * * *
trace of the variable BUS
time :        -1 ,   0 value =            U
time :         5 ,   1 value =            Z
time :         8 ,   0 value =            6
time :        14 ,   0 value =            7
time :        63 ,   1 value =           10
time :        66 ,   0 value =            Z
time :        69 ,   1 value =            2
```

```
time :          72 ,    0 value =             Z
time :          75 ,    1 value =             Z

 * * * * * * * * * * * * * * * * * * * * * * * *
trace of the variable CPTR
time :          -1 ,    0 value =             U
time :          12 ,    0 value =           4
time :          24 ,    0 value =           3
time :          36 ,    0 value =           2
time :          48 ,    0 value =           1
time :          54 ,    0 value =           0

 * * * * * * * * * * * * * * * * * * * * * * * *
trace of the variable B
time :          -1 ,    0 value =             U
time :          18 ,    0 value =           7

 * * * * * * * * * * * * * * * * * * * * * * * *
trace of the variable MQ
time :          -1 ,    0 value =             U
time :          12 ,    0 value =           6
time :          24 ,    0 value =           3
time :          36 ,    0 value =           9
time :          48 ,    0 value =           4
time :          54 ,    0 value =          10

 * * * * * * * * * * * * * * * * * * * * * * * *
trace of the variable D_ACC
time :          -1 ,    0 value =             U
time :           9 ,    1 value =           0
time :          24 ,    0 value =           0
time :          30 ,    0 value =           7
time :          36 ,    0 value =           3
time :          42 ,    0 value =          10
time :          48 ,    0 value =           5
time :          54 ,    0 value =           2

 * * * * * * * * * * * * * * * * * * * * * * * *
```

Trace de toutes les variations sur une
variable booléenne (0,1,X,Z,U)



CADOC SYSTEM
Circuit : test_chro

BIBLIOGRAPHIE

[AGR 80] **V.D. AGRAVAL et al.**
"A mixed mode simulator"
17th DAC, pp. 618-625, Minneapolis, USA, Juin 1980.

[ALM 84] **C.B. ALMEIDA, M.J.A. LANCA**
"An eight value modelling technique for logic simulation of transmission gates and buses"
European Conf. on Electronic Design Automation (EDA 84), pp. 23-27, Warwick, UK, Mars 1984.

[AMB 83] **P. AMBLARD, M. CRASTES DE PAULET et al.**
"CADOC : a functional specification tool"
Int. Conf. on Computer Aided Design (ICCAD 83), pp. 65-66, Santa Clara, USA, Septembre 1983.

[AMB 84] **P. AMBLARD, M. CRASTES DE PAULET et al.**
"CADOC : a functional specification and simulation tool for VLSI"
European Conc. on Electronic Design Automation (EDA 84), pp. 147-151, Warwick, UK, Mars 1984.

[BAC 81] **J. BACKUS**
"Function level program as mathematical objects"
ACM Conf. on Functional Programming Languages and Computer Architectures, pp. 1-10, Portsmouth, UK, Octobre 1981.

[BAR 84] **M.R. BARBACCI**
"ADA as a hardware description language : an initial report"
Research Report CMU CS-85-104, Carnegie-Mellon University, Pittsburgh, USA, Décembre 1984.

[BAR 85] **M.R. BARBACCI et al.**
"Representing time and space in an object oriented hardware description language"
Research Report CMU CS-85-105, Carnegie-Mellon University, Pittsburgh, USA, Janvier 1985.

[BAT 81] **J. BATALI et al.**
"The DPL / Daedalus Design environment"
First Int. Conf. on Very Large Scale Integration (VLSI 81), pp. 183-192, Edinburgh, UK, Août 1981.

[BEL 82] **C. BELLON, R.VELAZCO et al.**
"Automatic generation of microprocessor test programs"
19th DAC, pp. 566-573, Las Vegas, USA, Juin 1982.

[BEL 83] **C.BELLON, E. KOLOKITHAS et al.**
"Génération automatique de programmes de test pour microprocesseurs"
Laboratoire Circuits et Systèmes, Rapport final de contrat ADI-ESD-IMAG, 1983.

[BEL 85] **C. BELLON, M. CRASTES DE PAULET et al.**
"CADOC system : a tool for multilevel description and test generation for VLSI circuits"
7th Int. Symp. on Computer Hardware Description Languages and their Applications (CHDL 85), pp. 364-380, Tokyo, Japon, Août 1985.

[BER 85] **J.L. BERGERAND, P. CASPI et al.**
"Outline of a real data flow language"
IEEE Int. Symp. on Real Time (à paraitre), San Diego, USA, Décembre 1985.

[BOR 81] **D. BORRIONE**
"Langages de description de systèmes logiques"
Thèse d'état, INPG, Grenoble, France, Juillet 1981.

[BOR 85] **D. BORRIONE, C. LE FAOU**
"Overview of the CASCADE multilevel hardware description language and its mixed-mode simulation mechanisms"
7th Int. Symp. on Computer Hardware Description Languages and their applications (CHDL 85), pp. 239-260, Tokyo, Japon, Aout 1985.

[BRE 72] **M.A. BREUER**
"A note on three valued logic simulation"
IEEE Trans. on Computers, pp. 399-402, Avril 1972.

[BRY 81] **R.E. BRYANT**
"A switch level simulation model for integrated logic circuits"
Research Report MIT LCS TR259, Massachussets Institute of Technology, Cambridge, USA, Mars 1981.

[BRY 84] **R.E. BRYANT**
"A switch level model and simulator for MOS digital systems"
IEEE Trans. on Computer, pp. 160-177, Fevrier 1984.

[BRY 85] **R.E. BRYANT**
"Performance evaluation of FMOSSIM, a concurrent switch level fault simulator".
22th DAC, pp. 715-719, Las Vegas, USA, Juin 1985.

[CHA 75] **B.R. CHAWLA, H.K. GUMMEL**
"MOTIS an MOS timing simulator
IEEE Trans. on Circuits and Systems, pp. 901-910, decembre 1975.

[CHE 84] **C.F. CHEN et al.**
"The second generation MOTIS mixed mode simulator"
21th DAC, pp. 10-17, Albuquerque, USA, Juin 1984.

[CHU 74] **Y. CHU**
"Introducing CDL"
IEEE Computer, pp. 42-44, Décembre 1974.

[CIS 84] **M. CISNEROS GASCON**
"Programmation parallèle et programmation fonctionnelle : proposition pour un langage"
Thèse de troisième cycle, INPG, Grenoble, France, Octobre 1984.

[COR 81] **W.E. CORY, W.M. VAN CLEEMPUT**
"Symbolic simulation for functional verification using ADLIB and SDL"
18th DAC, pp. 82-89, Nashville, USA, Juin 1981.

[CRA 85] **M. CRASTES DE PAULET, G. SAUCIER**
"Functional approach of low level modelling using CADOC description language"
Rapport interne, Laboratoire Circuits et Systèmes, IMAG, France, Mars 1985.

[CRA 85] **M. CRASTES DE PAULET, G. SAUCIER**
"Simulation fonctionnelle fine : le langage CADOC.LD"
Col. Nat. Conception de Circuits à la Demande, pp. 280-295, Grenoble, France, Mai 1985.

[DIE 74] **D.L. DIETMEYER**
"Introducing DDL"
IEEE Computer, pp. 34-38, Décembre 1974.

[DIE 83] **D.L. DIETMEYER et al.**
"WISLAN : a CONLAN member for gate array design"
6th Int. Symp. on Computer Hardware Description Languages and their applications (CHDL 83), pp. 31- 42, Pittsburgh, USA, Mai 1983.

[DIJ 75] **E.W. DIJKSTRA**
"Guarded commands, nondeterminacy and formal derivation of programs"
CACM pp. 453-457, Août 1975.

[ENG 83] **W.L. ENGL et al.**
"Device Modelling"
Proc. of the IEEE, pp. 10-33, Janvier 1983.

[FEL 83] **S.I. FELDMAN**
"The circuit language Xi"
Int. Conf. on Computer Design (ICCD 83), pp. 652-655, New York, USA, Octobre 1983.

[FLA 80] **P.L. FLAKE, P.R. MOORBY, G. MUSGRAVE**
"Logic simulation of bidirectional tri-state gates"
IEEE Int. Conf. on Circuits and Computers (ICCC 80), pp. 594-600, New York, USA, Octobre 1980.

[FLA 81] P.L. FLAKE et al.
3HILO MARK II hardware description language"
5th Int. symp. on Computer Hardware Description Languages and their applications (CHDL 81), Kaiserslautern , RFA, Septembre 1981.

[FLA 83] **P.L. FLAKE, P.R. MOORBY, G. MUSGRAVE**
"An algebra for logic strength simulation"
20th DAC, pp. 615-618, Miami Beach, USA, Juin 1983

[GOR 81] **M. GORDON**
"A very simple model of sequential behaviour of NMOS"
First Int. Conf. on Very Large Scale intgration (VLSI 81), pp. 85-94, Edinburgh, UK, Août 1981.

[GRU 83] **J.W. GRUNDMANN**
"Event driven MOS timing simulation"
Int. Conf. on Computer Aided Design (ICCAD 83), pp. 141-142,
Santa Clara, USA, Septembre 1983.

[HAC 81] **G.D. HACHTEL, A.L. SANGIOVANNI VINCENTELLI**
"A survey of third-generation simulation techniques"
Proc. of the IEEE, pp. 1264-1280, Octobre 1981.

[HAL 84] **N. HALBWACHS**
"Modélisation et analyse du comportement des systèmes informatiques temporisés"
Thèse d'état, USMG/INPG, Grenoble, France, Juin 1984.

[HAY 82] **J.P. HAYES**
"A unified switching theory with applications to VLSI design"
Proc. of the IEEE, pp. 1140-1151, octobre 1982.

[HAY 82b] **J.P. HAYES**
"A fault simulation methodology for VLSI"
19th DAC, pp. 393-399, Las Vegas, USA, Juin 1982.

[HAY 84a] **J.P. HAYES**
"An experimental MOS fault simulation program CSASIM"
21th DAC, pp. 2-9, Albuquerque, USA, Juin 1984.

[HAY 84b] **J.P. HAYES**
"A systematic approach to multivalued digital simulation"
Int. Conf. on Computer Design (ICCD 84), pp. 177-182,
Port Chester, USA, Octobre 1984.

[HEN 85] **B. HENNION, D. COQUELLE**
"A new algorithm for third generation circuit simulator : the one step relaxation method"
22th DAC, pp. 137-143, Las Vegas, USA, Juin 1985.

[HIL 79] **D.D. HILL**
"ADLIB : a modular, strongly-typed computer design language"
4th Int. Symp. on Computer Hardware Description Languages and their applications (CHDL 79), pp. 75-81, Palo Alto, USA, Octobre 1979.

[HIL 80] **D.D. HILL**
"Language and environment for multilevel simulation"
Technical Report 185, Computer Systems Laboratory, Stanford University, Stanford, USA, Mars 1980.

[HIL 74] **D.D. HILL**
"Introducing AHPL"
IEEE Computer, pp. 28-30, Décembre 1974.

[HOA 79] **C.A.R. HOARE**
"Communicating sequential processes"
CACM, pp. 666-677, août 1979.

[HSP 83] **THOMSON EFCIS**
"Notice d'utilisation HSPICE"
Août 1983.

[JEN 73] **F.S. JENKINS et al.**
"MOS Device Modeling for computer implementation"
IEEE Trans. on Circuit Theory, pp. 649-658, Novembre 1973.

[KNU 82] **M.S. KNUDSEN**
"A nine valued logic simulation for digital nMOS circuits"
12th Int. Symp. on Multiple Valued Logic, pp. 293-297, Paris, France, Mai 1982.

[LEI 81] **S.M. LEINWAND**
"Process-oriented logic simulation"
18th DAC, pp. 511-517, Nashville, USA, Juin 1981.

[LEL 82] **E. LELARASMEE, A. SANGIOVANNI VINCENTELLI**
"Relax : a new circuit simulator for large scale MOS integrated circuits"
19th DAC, pp. 682-690, Las Vegas, USA, Juin 1982.

[LIN 73] **H.C. LIN, W.N. JONES**
"Computer analysis of double diffused MOS transistor for integrated circuits"
IEEE Trans. on Electron Devices, pp. 275-282, Mars 1973.

[MAY 83] **D. MAY**
"OCCAM"
SIGPLAN Notices, Vol.18, n°4, Avril 1983.

[MER 85] **J. MERMET**
"Several steps towards a circuit integrated CAD system : CASCADE"
7th Int. Symp. on Computer Hardware Decsription Languages and their applications (CHDL 85), pp. 226-238, Tokyo, Japon, Août 1985.

p
[MIL 84] **R.E. MILNE**
"A tutorial for LTS"
Internal Technical Memorandum 225.84.1, Standard Telecommunication Laboratories, Harlow, UK, Janvier 1984.

[MOA 81] **M. MOALLA**
"Spécification et conception sûre d'automatismes discrets complexes, basées sur l'utilisation du GRAFCET et des réseaux de Pétri."
Thèse d'état, USMG/INPG, Grenoble, France, Juillet 1981.

[MUN 83] **T. MUNTEAN**
"Introduction à OCCAM : langage parallèle issu de CSP pour la programmation des systèmes de transinateurs".
Laboratoire de Génie Informatique, Rapport de Recherche n° 430, IMAG, Grenoble, France, Décembre 1983.

[MUN 85] **T. MUNTEAN**
Application d'OCCAM à la description du matériel : communication personnelle"
Laboratoire de Génie Informatique, IMAG, Grenoble, France, 1985.

[NEW 79] **A.R. NEWTON**
"Techniques for the simulation of large scale integrated circuits"
IEEE Trans. on Circuits and Systems, pp. 741-749, Septembre 1979.

[NHA 80] **H.N. NHAM, A.K. BOSE**
"A multiple delay simulator for MOS LSI circuits"
17th DAC, pp. 610-617, Minneapolis, USA, Juin 1980.

[OCC 83] **OCCAM Programming Manual**
INMOS Limited, 1983.

[OKA 83] **K. OKAZAKI et al.**
"A multiple media delay simulator for MOS LSI circuits"
20th DAC, pp. 279-285, Miami Beach, USA, Juin 1983.

[PAW 81] **A. PAWLAK, J. JEZEWSKI**
"MODLAN - a language for multilevel description and modeling of digital systems"
5th Int. Symp. on Computer Hardware Description Languages and their applications (CHDL 81), pp. 79-93, Kaiserslautern, RFA, Septembre 1981.

[PAW 82] **A. PAWLAK**
"Digital logic modeling based on MODLAN"
19th DAC, pp. 763-770, Las Vegas, USA, Juin 1982.

[PIL 83] **R. PILOTY et al.**
"CONLAN report"
Springer-Verlag, 1983.

[PIL 85] **R. PILOTY et al.**
"The CONLAN project, concepts, implementations and applications"
IEEE Computer, pp. 81-92, Février 1985.

[PIL 85b] **D. PILAUD**
Réunion du projet C^3, Paris, France, Mars 1985.

[QUI 83] **P.QUINTON**
"Algorithmes systoliques : de la théorie à la pratique"
Publication Interne n° 196, IRISA, Rennes, France, Mars 1983.

[RAM 79a] **F.J. RAMMIG**
"The implementation of the computer hardware description language CAP and its applications"
4th Int. Symp. on Computer Hardware Description Languages and their Applications (CHDL 79), pp. 138-144, Palo Alto, USA, Octobre 1979.

[RAM 79b] **F. J. RAMMIG**
"The concurrent programming language CAP and the μ-processor oriented CAP CAD system"
Euromicro 79 (Microprocessors and their applications)
pp. 249-257, 1979.

[RAM 81] **F.J. RAMMIG**
"The CAP/DSDL system : simulation and case study"
5th Int. Symp. on Computer Hardware Description Languages and their applications (CHDL 81), pp. 213-227, Kaiserslautern, RFA, Septembre 1981.

[RAM 83a] **F.J. RAMMIG**
"Hierarchical modular description of VLSI system"
pp. 112-116, 1983.

[RAM 83b] **F.J. RAMMIG**
"Description and simulation of MOS devices in register transfert languages"
2nd Int. Conf. on Very Large Scale Integration (VLSI 83),
pp. 16-25, Trondheim, SV, Août 1983.

[RAR 85] **J. RARIVOMANANA**
"Système CADOC : génération fonctionnelle de test pour les circuits complexes
Thèse de Docteur Ingénieur, INPG, Grenoble, France,
Novembre 1985.

[ROB 83] **P. ROBINSON, J. DION**
"Programming languages for hardware description"
20th DAC, pp. 12-16, Miami Beach, USA, Juin 1983.

[ROB 83a] **C.ROBACH, P. MALECHA et al.**
"CATA : a computer aided test analysis system"
IEEE Design and Test, pp. 68-79, Mai 1984.

[ROB 84] **C.ROBACH, G.MICHEL et al.**
"Computer analysis testability : evaluation and test generation"
Test Conf., pp. 338-345, Philadelphie, Octobre 1984.

[ROI 84] **C. ROISIN**
"La description des protocoles de communication avec le langage parallèle CSP".
Thèse de Docteur Ingénieur, INPG, Grenoble, France,
Octobre 1984.

[SAL 83] **R.A. SALEM et al.**
"Iterated timing analysis in SPLICE"
Int. Conf. on Computer-Aided Design (ICCAD 83), pp. 139-147,
Santa Clara, USA, Septembre 1983.

[SAM 82] **K. SAMINADAYAR**
Cours de physique des dispositifs, DEA Electronique et Circuits Intégrés, USMG/INPG, Grenoble, France, 1982.

[SHA 85] **M. SHAHDAD et al.**
"VHSIC hardware description language"
IEEE Computer, pp. 94-102, Février 1985.

[SHE 81] **W. SHERWOOD**
"A MOS modelling technique for 4-state true value hierarchical logic simulation".
18th DAC, pp. 775-785, Nashville, USA, Juin 1981.

[SIE 74] D.P. SIEWIOREK
"Introducing ISP"
IEEE Computer, pp. 39-41, Décembre 1974.

[SIL 83] **Documentation HELIX**
SILVAR LISCO, 1983.

[SIS 82] **J.M. SISKIND et al.**
"Generating custom high performance VLSI designs from succint algorithmic descriptions"
Conference an Advanced Research in VLSI, pp. 28-40,
Massachussets Institute of Technology, Cambridge, USA, 1982.

[SOU 83] **J.R. SOUTHARD**
"Mac Pitts : an approach to silicon compilation"
Computer, pp. 74-82, decembre 1983.

[STE 83] **P. STEVENS, G. ARNOUT**
"BIMOS, an MOS oriented multilevel logic simulator"
20th DAC, pp. 100-106, Miami Beach, USA, Juin 1983.

[SUZ 85] **N. SUZUKI**
"Concurrent PROLOG as an efficient VLSI design language"
Computer, pp. 33-40, Fevrier 1985.

[TIA 85] **F. TIAR**
"CADOC : applications des mecanismes de compilation separee à un langage de description de circuits"
Rapport de DEA, Laboratoire Circuits et Systèmes, INPG, Grenoble, France, Juin 1985

[VAI 83] **A.K. VAIDYA et al.**
"WISLAN : technology transformation and optimization"
6th Int. Symp. on Computer Hardware Description Languages and their applications (CHDL 83), pp. 43-54, Pittsburgh, USA, Mai 1983.

[VER 82] **D. LE VERRAND**
"Le langage ADA, Manuel d'évaluation"
Dunod, Paris, France, 1982.

[WAL 85] **R.A. WALKER, D.E. THOMAS**
"A model of design representation and synthesis"
22th DAC, pp. 453-459, Las Vegas, USA, Juin 1985.

[WEE 73] **W.T. WEEKS et al.**
"Algorithms for ASTAP, a network analysis program"
IEEE Trans. on Circuit Theory, pp. 628-634, novembre 1973.

[WHI 80] **M.H. WHITE et al.**
"High accuracy MOS models for computer aided design"
IEEE trans. on Electron Devices, pp. 899-906, mai 1980.

AUTORISATION de SOUTENANCE

U les dispositions de l'article 3 de l'arrêté du 16 avril 1974

U les rapports de présentation de

. Madame le Professeur G. SAUCIER
. Monsieur G. MICHEL

Monsieur Michel CRASTES DE PAULET

st autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de)OCTEUR-INGENIEUR, spécialité "Microélectronique".

Fait à Grenoble, le 12 novembre 1985

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble