Système CADOC : génération fonctionnelle de test pour les circuits complexes

Jens A. Rarivomanana

▶ To cite this version:

Jens A. Rarivomanana. Système CADOC : génération fonctionnelle de test pour les circuits complexes. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1985. Français. NNT : . tel-00319028

HAL Id: tel-00319028
https://theses.hal.science/tel-00319028

Submitted on 5 Sep 2008

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

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

THESE

présentée à

l'Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR INGENIEUR
«Informatique»

par

Jens A. RARIVOMANANA

SYSTEME CADOC

GENERATION FONCTIONNELLE DE TEST POUR LES CIRCUITS COMPLEXES.

Thèse soutenue le 28 novembre devant la commission d'examen.

G. MAZARÉ	Président
M. CHEIN	Examinateurs
G. MICHEL	
G. SAUCIER	
C. BELLON	

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	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 & chin industrielle

****

A ma famille,
et à ma chère épouse.

Remerciements

Je tiens à remercier

Madame Le Professeur Gabriêle SAUCIER, Directeur du Laboratoire de recherche "CIRCUITS & SYSTEMES" de l'Institut IMAG, qui a accepté de diriger cette thèse,

Monsieur Le Professeur G. MAZARE, enseignant à l'ENSIMAG (INPG), qui a bien voulu présider la commission d'examen de cette thèse,

Monsieur G. MICHEL, Responsable du Département "Architecture des Micro-systèmes" du CNET (Grenoble), qui a accepté d'être rapporteur et membre du jury,

Madame C. BELLON, Maître-Assistant à l'ENSIMAG (INPG), pour les remarques et les suggestions sur la partie "application de CADOC.LD pour le test des circuits complexes" présentée dans la section II de ce document,

Monsieur Le Professeur M. CHEIN de bien vouloir accepter de faire partie du jury,

Messieurs M. CRASTES DE PAULET et S. HANRIAT, ainsi que Mademoiselle F. TIAR pour leur collaboration dans la réalisation de ce travail.

TABLE DES MATIERES

SECTION I : LANGAGE DE DESCRIPTION DE MATERIEL
LE LANGAGE CADOC.LD

CHAPITRE I : LANGAGES NON SPECIFIQUES

I - LANGAGES IMPERATIFS

I.1 - ADLIB

I.2 - Description de circuits en ADLIB

I.3 - Manipulation du temps

I.4 - Remarques sur ADLIB

I.5 - Autres approches

II - LANGAGES PARALLELES

II.2 - OCCAM

II.1.1 - Processus élémentaires

II.1.2 - Construteurs

II.1.3 - Déclaration / Typage

II.1.4 - Descriptions de circuits en OCCAM

II.3 - ADA

II.2.1 - Brefs rappels sur ADA

II.2.2 - Description de circuits en ADA

II.2.3 - Remarques

III - LANGAGES FONCTIONNELS ET DECLARATIFS

III.1 - Introduction

III.2 - LUSTRE

III.2.1 - Expressions

III.2.2 - Structure d'un programme

III.2.3 - Remarques

III.4 - Conclusion

IV - LANGAGES D'INTELLIGENCE ARTIFICIELLE

IV.1 - PROLOG

IV.2 - LISP

CHAPITRE II : LANGAGES SPECIFIQUES

I - HILO MARK II
I.1 - Généralités
I.2 - Description structurelle
I.3 - Description fonctionnelle
I.3.1 - les actions
I.3.2 - la manipulation du temps : événements

II - CAP/DSDL
II.1 - Généralités
II.2 - Variables et types de données
II.3 - Manipulation du temps
II.4 - Définition de nouveaux modèles
II.5 - Modélisation au niveau transistor

III - MODLAN
III.1 - Généralités
III.2 - Description structurelle : SM
III.3 - Description fonctionnelle : FM
III.4 - Description à contrôle explicite : CM
III.5 - Remarques

IV - VHDL
IV.1 - Généralités
IV.2 - Interface
IV.2.1 - ports d'entrées/sorties
IV.2.2 - paramètres génériques
IV.2.3 - types de données
IV.2.4 - définition d'attributs
IV.2.5 - assertions
IV.3 - Les corps d'une entité de conception
IV.3.1 - modèle temporel
IV.3.2 - spécification comportementale
IV.3.3 - spécification architecturale
IV.4 - Remarques

V - APPROCHE CONLAN
V.1 - Généralités
V.2 - Modularité
V.3 - Langage BCL
V.3.1 - modèle temporel
V.3.2 - les types de valeurs
V.3.3 - les types de porteuses
V.3.4 - les structures de données
V.3.5 - exemples
V.4 - Exemple de dérivation de langage : WISLAN
V.5 - Remarques

CHAPITRE III : LE LANGAGE CADOCL.LD DU SYSTEME CADOC

I - INTRODUCTION

II - TYPES DE DONNEES ET VARIABLES
II.1 - Types simples
II.1.1 - type booléen
II.1.2 - type entier
II.1.3 - type front
II.1.4 - type énénuméré
II.2 - Types structurés
II.2.1 - enregistrement
II.2.2 - tableau

III - DESCRIPTION FONCTIONNELLE D'UNE RESSOURCE
III.1 - En-tête
III.2 - Partie déclarative d'une description
III.2.1 - les constantes
III.2.2 - les types
III.2.3 - les variables
III.2.4 - les assertions statiques
III.2.5 - exemple

III.2.6 - les RGF utilisées et leurs occurrences
III.2.7 - les connectiques

III.3 - Partie fonction d'une description
III.3.1 - Modes de définition de la fonction
III.3.2 - Graphe interprété et temporisé : GIT
III.3.2.1 - expressions, blocs algorithmiques, chronogrammes
III.3.2.2 - types d'affectation généralisée
III.3.2.3 - places
III.3.2.4 - transitions
III.3.2.5 - modèle temporel
III.3.2.6 - règles d'évolution - simulation
III.3.3 - Assertions dynamiques
IV - RESSOURCES ALGORITHMIQUES
IV.1 - Constitution
IV.2 - Déclaration
IV.3 - Utilisation

V - CONCLUSION

SECTION II : TEST DES CIRCUITS COMPLEXES

CHAPITRE I: INTRODUCTION ET DEFINITIONS

I - DEFINITION
I.1 - Défaillance - Panne - Erreur
I.2 - Test de distinction - Test d'identification

II - LA GENERATION DES VECTEURS DE TEST
III - L'APPLICATION DU TEST
IV - L'ANALYSE DES RESULTATS

CHAPITRE II : LES METHODES DE GENERATION DE TEST

I - LE TEST ALEATOIRE
I.1 - Test aléatoire des circuits combinatoires
I.2 - Test aléatoire des circuits séquentiels
I.3 - Conclusion

II - TEST AVEC VECTEURS PREDETERMINES
II.1 - Test de distinction
II.1.1 - méthode de sensibilisation de chemin : D-Algorithme
II.1.2 - différence booléenne
II.1.3 - méthode de POAGE
II.1.4 - extension du D-Algorithme
II.1.5 - S-Algorithme
II.1.6 - génération à partir du graphe de transformation d'états
II.1.7 - technique développée par GTE lab.
II.1.8 - conclusion
II.2 - Test d'identification
II.2.1 - test exhaustif
II.2.2 - méthode d'identification d'automate
II.2.3 - graphe abstrait d'exécution
II.2.4 - technique développée par Texas Instruments
II.2.5 - conclusion

II.3 - Récapitulation sur le test avec vecteurs prédéterminés

III - APPROCHE MULTINIVEAU : TEST FONCTIONNEL DES CIRCUITS A CONTROLEURS
III.1 - Stratégie multiniveau
III.2 - Test d'identification d'un contrôleur
III.3 - Test de la partie opérative
III.4 - Conclusion

IV - CONCLUSION

CHAPITRE III : OUTIL D'EXECUTION SYMBOLIQUE TEMPORISEE

I - INTRODUCTION

II - TEMPORISATION D'UN CHEMIN
II.1 - Noeuds de référence
II.2 - Définition des instants de référence
II.3 - Temporisation d'un chemin

III - CONDITION DE CHEMIN TEMPORISEE
III.1 - Définition
III.2 - Prise en compte des contraintes de stabilité
III.3 - Prédicats complexes
III.4 - Les boucles
III.5 - Prise en compte de l'opérateur "CHANGE"

IV - ECHEANCIERS SYMBOLIQUES DES SORTIES

V - CONCLUSION

CHAPITRE IV : GENERATION FONCTIONNELLE DE TEST

I - STRATEGIE DE TEST

II - FICHIER DE CONNAISSANCES D'UNE RESSOURCE

- II.1 - Les connaissances
- II.2 - Représentation des connaissances
- II.3 - Visibilité des objets définis dans une règle
- II.4 - Evaluation de la partie "condition"
- II.5 - Construction des échéanciers globaux d'une variable
- II.6 - Les variables d'une règle
- II.7 - Exécution symbolique et règles
- II.8 - Conclusion

III - RECHERCHE DES ACTIVATIONS FONCTIONNELLES GLOBALES

- III.1 - Phase de consistance
  - III.1.1 - étape 1 : graphe de pseudo-précédence
  - III.1.2 - étape 2 : marquages des règles
  - III.1.3 - étape 3 : propagation en avant
  - III.1.4 - unification et marquages des règles
- III.2 - Phase de propagation
  - III.2.1 - étape 4 : graphe de propagation
  - III.2.2 - étape 5 : propagation en avant

IV - CONCLUSION

SECTION I

SECTION I : LANGAGE DE DESCRIPTION DE MATERIEL
LE LANGAGE CADOC.LD

La conception de langages de description de matériel, dont les premières étaient faites vers les années 74 [CHU 74], constitue un important domaine de la recherche. Sept conférences internationales sur les langages de description de matériel ont déjà eu lieu : la première en 1973.

D'une manière simple, les langages de decription de matériel ont été conçus pour pouvoir simuler les comportements des circuits logiques que l'on ne peut pas facilement exprimer avec les langages classiques de programmation. C'est pourquoi, nous verrons dans le chapitre I de cette section une étude sur l'application des langages classiques de programmation à la description de matériel éventuellement en définissant des extensions.

Dans le chapitre II, nous ferons une brève présentation des derniers langages spécifiques. Les remarques que nous avons pu faire à l'issue de cette étude nous a permis de concevoir le langage de description CADOC.LD présenté dans le chapitre III. Ce langage est le langage de base de tous les outlis constituant le système CADOC (Computer-Aided Design Of complex Circuits).

Mais auparavant, nous voulons donner certaines définitions.

- DEFINITIONS

Dans les chapitres de cette section, 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 CHDL.
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 recoivent parfois des interprétations différentes dans le domaine des langages, selon les auteurs.

Notion de niveau de description

Par niveau de description on entendra niveau d'abstraction de la description, pour chaque niveau nous donnerons les éléments matériels primitifs des descriptions. On peut remarquer ici la confusion entre niveau de description et niveau de conception (primitives du niveau de description).

Les termes de structurel et fonctionnel seront définis ultérieurement, notons simplement qu'à tout niveau de description on peut associer une représentation fonctionnelle.

<u>Exemple</u> : niveau logique .

<u>Une</u> représentation fonctionnelle de :

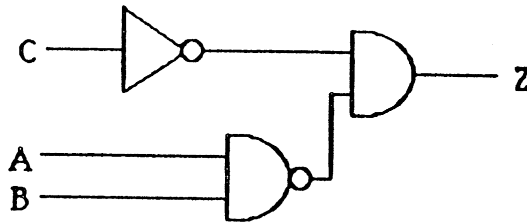


Figure 1-1.

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

Par ordre d'abstraction croissant on définira les niveaux suivants :

- <u>niveau implantation</u> (layout)
  Le circuit est représenté par ses masques, les éléments primitifs de la description étant des formes géométriques (rectangles, ...). Ce niveau ne concerne pas directement les langages de description de matériel.

- <u>niveau électrique</u>
  Le circuit est représenté comme un assemblage d'éléments dont le comportement est une fonction continue du temps (transistors,résistance, ...) Ce niveau est implicitement lié à une technologie donnée et à des outils de simulations.

- <u>niveau logique</u>
  Le circuit est représenté comme un assemblage d'éléments à comportement discret (portes logiques).

- niveau transfert de regitres (RT-1)

Un circuit est défini par l'ensemble de ses points de mémorisation en termes de registres, mémoires ; le chemin de données est modélisé par des opérations sur ces points de mémorisation (un transfert est décrit par une affectation, un circuit combinatoire est décrit par une expression booléenne).

- niveau algorithmique

Le circuit est décrit par un algorithme de calcul éventuellement temporisé, sans référence à une implémentation en terme de chemins de données et points de mémorisation.

Langages de descriptions fonctionnels et structurels

Un circuit peut être défini à tous les niveaux comme un assemblage structurel de circuits déjà existants (construits ou primitifs). Nous entendons par langages de description fonctionnel un langage ne possédant pas d'objets primitifs prédéfinis (au sens description non accessible ni modifiable), mais plutôt offrant aux utilisateurs des mécanismes de définition d'objets primitifs. Cette définition nous permettant de classer les langages RT-1 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).

Hiérarchie et multiniveau

Par hiérarchie, nous entendons description d'un circuit comme un assemblage structurel de sous-circuits, eux-même décrits hiérarchiquement ou non.

Si les circuits "feuilles" de la hiérarchie sont décrits à des niveaux différents d'abstraction (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.

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, sinon on parlera de langage non-procédural (notion d'état défini comme étiquette conditionnelle).

CHAPITRE I : LANGAGES DERIVES DES LANGAGES DE PROGRAMMATION

I - LANGAGES IMPERATIFS

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

I.1 - ADLIB [HIL 79]

ADLIB est un langage de description de matériel dérivé de PASCAL. Le choix de Pascal comme base d'un CHDL a été motivé par les raisons suivantes : importante communauté d'utilisateurs, langage fortement typé permettant des vérifications statiques, 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.

Nous ne décrirons pas ce qui concerne la partie déclarative d'une description ADLIB (types, variables et vérifications effectuées à la compilation sont directement dérivés de Pascal) mais nous allons présenter les primitives de contrôle (instructions) disponibles. Ces primitives peuvent être regroupées en 2 catégories :

- **primitives Pascal**

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

- <u>primitives ADLIB</u>

* **Affectation temporisée**

<u>assign</u> <expr> <u>to</u> <nom_de_port> <clause_temporelle>

Cette affectation permet d'affecter la valeur de l'expression <expr> à la variable <nom _de _port> après un retard ou à l'instant spécifié par <clause_temporelle>.
Par défaut la valeur du retard est 0. La clause temporelle peut faire référence à la variable <u>time</u> représentant le temps courant de simulation ou à une horloge et à ses phases définies par l'utilisateur (<u>sync</u> et <u>phase</u>).

Exemple

<u>assign</u> r.carry <u>to</u> line_1 <u>sync</u> ck1 <u>phase</u> 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**

<u>wait for</u> <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 (<u>delay</u>), soit par une évènement (<u>sync</u> et <u>phase</u>), 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>.

Exemple

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

Figure 1-2.

* 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 (wait for), 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 exécutés indépendament 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 booléenne est évaluée, si elle est vérifiée la liste d'instructions associées 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.

I.2 - DESCRIPTIONS DE CIRCUITS EN 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])

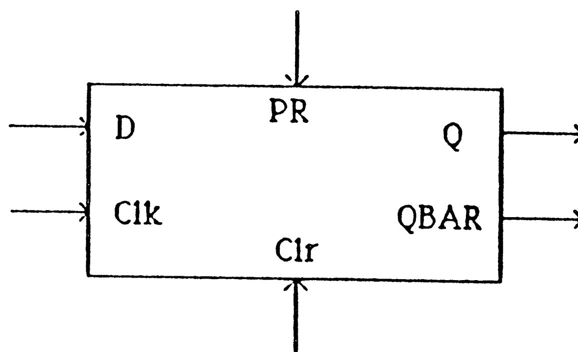


Figure 1-3a.

```
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 deCLK *)
          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

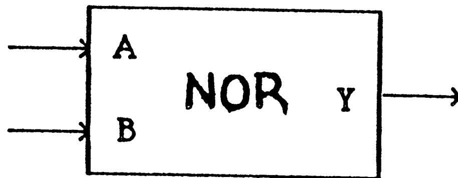


Figure 1-3b.

```
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 ;
```

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

I.3 - 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)
```

Ceci définit une horloge "phi" à 4 phases de durée 10.

I.4 - REMARQUES SUR ADLIB

1)- 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.

2)- Aucune vérification autre que celles faites dans le langage PASCAL ne semble prévu, les vérifications étant effectuées statiquement (pas de notions d'assertions vérifiées dynamiquement en cours de simulation).

3)- 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.

4)- Le fait d'utiliser Pascal comme langage de 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.

5)- La description de la circuiterie MOS et des problèmes y afférent n'est pas abordée (cf. [CRA 85]).

II.5 - 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 [DIO 83]. Dans l'approche basée sur MODULA-2 la partie fonctionnelle et la partie structurelle d'un circuit sont décrits séparement, l'exécution de la partie structurelle générant la hiérarchie du circuit en vue de son implantation.

II - LANGAGES PARALLELES

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 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 ; nous allons donc nous intéresser ici aux langages parallèles et en particulier à OCCAM (qui est l'implémentation du modèle CSP de Hoare) et à ADA.

Dans un premier paragraphe, nous allons résumer 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.1 - OCCAM

Le langage OCCAM est une réalisation directe des notions développées dans le modèle CSP proposé par [HOA 78] 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 communiquent par l'intermédiaire de composants élémentaires de synchronisation (ou canaux) qui sont les seuls objets pouvant être partagés entre processus.

II.1.1 - <u>Processus élémentaires</u>

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

<u>Exemples</u>

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

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

* "C[i]!a" :
  processus qui lorsqu'une communication sera établie avec un autre processus écrira 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.1.2 - 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 [DIJ 75] (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.

II.1.3 - 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).

II.1.4 - Descriptions de circuits en OCCAM [MUN 85]

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.

Une étude plus détaillée est faite dans [CRA 85]. Nous donnons ici comme exemple la description d'une porte NAND.

- **description d'une porte NAND**

La description en OCCAM d'une porte NAND fait intervenir la définition de 4 canaux (figure 1-4) : 3 canaux d'E/S classiques (E0,E1,S) et un canal pour arrêter le processus associé (STOPPER).

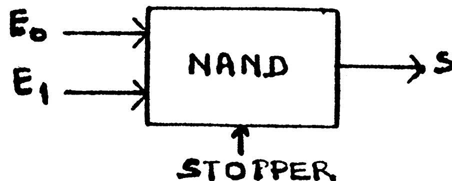


Figure 1-4.

```
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
```

Figure 1-5.

Remarques

1)- 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. Ce qui nécessite la définition et l'utilisation de processus de connexion qui a pour tâche de dupliquer les informations partagées par plusieurs processus.

Par ailleurs, la nécessité de ces processus s'explique aussi 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 par l'utilisation de processus de connexion rend difficile l'utilisation de OCCAM pour les circuits fortement couplés.

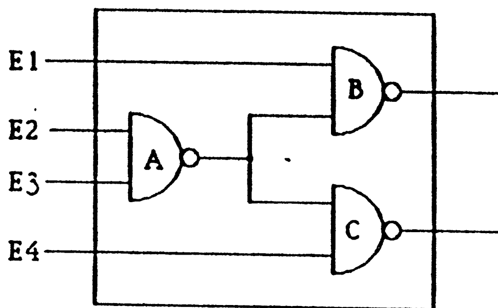


Figure 1-6a.
Représentation logique

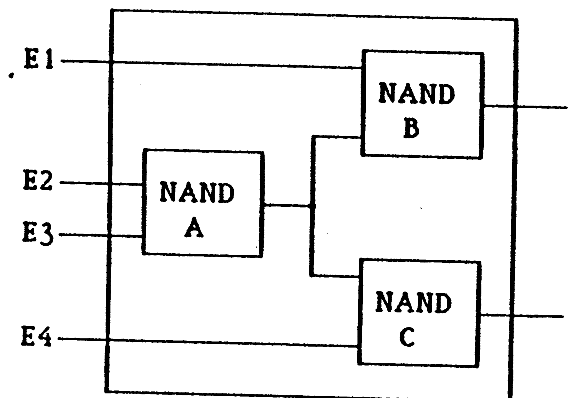


Figure 1-6b
Modélisation OCCAM "incorrecte"

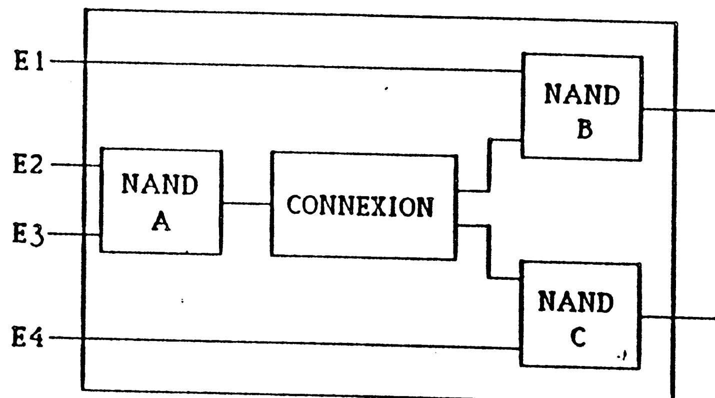


Figure 1-6c : Modélisation OCCAM "correcte"

-2) Contrôle de la simulation

Un circuit combinatoire est modélisé en OCCAM comme un ensemble de processus parallèles (décrivant chacun le comportement d'une porte logique). Par définition le processus global ne s'arrêtera que lorsque tous les processus parallèles seront terminés. Mais pour qu'ils puissent se terminer en terme de simulation) 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).

3)- 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).

4)- 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 qui est incrémenté de façon régulière.

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 - 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 d' 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.

II.2.1 - Brefs rappels sur ADA

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
- 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'unité et placée en tête de l'unité utilisatrice.

- une unité séparée est une unité 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.2.2 - 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

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
```

II.2.3 - Remarques

1) - Le mécanisme de simulation est spécifié de manière directe dans la description ; à la différence de CHDL 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. L'écriture des procédures présentent des difficultés en plus de celles de conception proprement dite.

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

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

4) - La description de la connectique utilise un ensemble de mécanismes de gestion des ports et des interconnexions entre ports ([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 econnexion 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.
- deconnecter : deconnexion de deux ports. Du fait des mécanismes de visibilité en ADA, l'opération de deconnexion est en général inutile ; la connexion étant faite au même niveau hierarchique que les objets interconnectés. 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 (équipotentielle associée).
- affecter : affectation d'une valeur à un port.
- fan_out : nombre de ports interconnectés à un port donné.

5) - 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, 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'util· ation des surcharges peut nuire à l isibilité des descriptions et que la compréhension des mécanismes de visibilité et de masquage rend leur utilisation difficile pour un utilisateur non informé.

6) - 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 quelle 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.

7) - Topologie des circuits.
Les auteurs proposent la prise en compte des caractéristiques spatiales (dimensions, forme) d'un circuit par ajout de nouveaux champs associés au type définissant le circuit considéré. De façon similaire à ce qui est proposé dans CADOC.LD, l'ajout 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 le slimitations de cette aproche.

8) - Modélisation du temps.

Dans ce qui précède, 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.

9) - 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 a [VER 82]), ces facilités étant proches de celles existantes dans le modèle CSP présenté précédement.

10) - 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 xx pour la

présentation du langage VHDL basé sur des concepts de type ADA mais possédant une syntaxe propre.

En conclusion, 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'utilisaton 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-partie susceptibles de justifier l'utilisation de ADA dans ce domaine.

III - LANGAGES FONCTIONNELS ET LANGAGES DECLARATIFS

III.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 séquentialité 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 où 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 prêtent bien à la manipulation mathématique des équations ou fonctions définissant un programme et sont donc adaptées à la vérification formelle 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 dans ce chapitre la présentation du langage déclaratif LUSTRE développé à partir des notions d'algèbre d'évènements ([HAL 84]).

III.2 - LUSTRE

Conçu pour des applications temps réel, le langage LUSTRE ([BER 85]) est susceptible d'application à la modélisation . 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 doti être définie par une équation (langage déclaratif) de la forme "X=E" ou E est une expression définissant une séquence e1,...,en,... de valeurs dans D(x) ; l'interprétation de "X=E" étant pour tout i=1,2,... $x_n=e_n$ 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.

III.2.1 - 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 e1,...,en,... alors "pre(E)" définit la séquence nil,e1,e2,...,en,...
L'opérateur pre est en général utilisé pour l'initialisation d'histoires.

- **Opérateur → (followed by)**

Si F=(f1,...) est de même type que E=(e1,...) alors E → F définit la séquence e1,f2,f3,...

Exemples :

- "X=1 → PRE(X)+1" définit une variable X dont la nième valeur xn satisfait
  $x_n=1$ si n=1
  $x_n=x_{(n-1)}+1$ si n 1 (et donc xn=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 boléenne 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
		1er cycle			2ème cycle		

X = EVERY (C)(E) → X1=E2 ou E3, X2=E5 ou E6 ou E7.

III.2.2 - Structure d'un programme LUSTRE

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

L'appel à un noeud se fait sous forme fonctionnelle : si N est un noeud dont l'entête est :
<u>node</u> N (I1:t1;...,Ip:tp) <u>returns</u> (J1:s1;...;Jq:sq) et si E1,...Ep sont des expressions de type t1,...,tp alors N(E1,...Ep) est un t-uple d'expressions de type (S1,...sq) calculées à partir des paramètres d'entrées E1,...Ep.

- **application**

Un des intérêts d'un langage du type LUSTRE étant de permettre la preuve d'équivalence entre descriptions associées à deux étapes successives de la conception d'un circuit.
Un exemple est donné en détail dans [CRA 85] illustrant cette possibilité sur une architecture systolique réalisant un produit de convolution ([PIL 85]).

III.2.3 - Remarques

1) - 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.
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.

2) - 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 aux assertions introduites dans le langage CADOC.LD.

3) - 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.

4) - Autre approche de même type : 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,...).

III.3 - Conclusion

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 à priori le but initial du langage évoqué 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.

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.

IV - LANGAGES D'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 de matériel.

IV.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 1-7a et donné figure 1-7b.

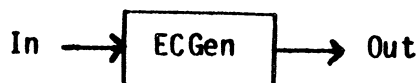


Figure 1-7a.

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

Figure 1-7b.

La notation [X/Y] représentant un liste constituée des valeurs successives sur un port donné ; le symbole ":-" définissant le comportement du circui 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 élevé. Les descriptions sont difficiles à manipuler dans la mesure ou le concepteur ne maitrise pas une syntaxe PROLOG-équivalente. Cette complexité est augmentée lorsque l'on souhaite manipuler explicitement le temps et rend le langage peu adapté à des modélisations de circuits non synchrones.

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 accés 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 (cf. paragraphe I) 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 vérificationde 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.

IV.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

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

2) - 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.

CHAPITRE II : LANGAGES SPECIFIQUES A LA DESCRIPTION DE MATERIEL

Nous avons vu dans le chapitre précédent que l'adaptation des langages de programmation pour la description de circuit présente certes des avantages (compilateurs déjà écrits, environnements importants, ...), mais auusi des inconvénients sur les points suivants qui sont propres aux systèmes logiques : modélisation des opérations asynchrones, du parallelisme, de l'architecture de circuit et la manipulation explicite de la dimension temporelle.

Après le précurseur CDL [CHU 74], 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]. Actuellement , les langages proposés sont multiniveau ou prétendent l'être. Nous étudierons dans ce paragraphe cinq approches différentes. Pour chacune d'elles, nous essaierons de donner un exemple.

I - HILO Mark II

I.1 - GENERALITES [FLA 81]

Les concepteurs de ce langage proposent une nouvelle version multiniveau par extension d'une antérieure version 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

I.2 - DESCRIPTION STRUCTURELLE

La description structurelle consiste à décrire le circuit comme une interconnection d'éléments de bas niveaux (les portes logiques, les portes trois-états, ...) ou de circuits ou de fonctions par l'intermédiaire des variables uniquement. L'ensemble des valeurs des variables constituent l'etat du circuit. Cette méthode est utilisée pour les descriptions structurelles au niveau logique.

Remarque :
Ce type de description nécessite un mécanisme de stabilisation défini dans le simulateur. Quand une variable change d'état, ,le simulateur doit calculer les nouvelles valeurs de toutes les connexions jusqu'à la stabilisation de celles-ci. En CADOC.LD nous verrons que le mécanisme de stabilisation est associé aux descriptions par l'utilisation de l'opérateur "CHANGE".

I.3 - DESCRIPTION FONCTIONNELLE

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

- les événements déclanchant le calcul des nouveaux états des variables pour éviter les boucles de stabilisations pendant la simulation;
- les registres activés par des opérateurs prédéfinis .

Exemple :

R := B loadifl A

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

Avec la sémantique définie pour les registres, on ne pourrait modéliser que les registres sensibles au niveau. Pour les registres sensibles au front, il faut utiliser une autre construction qui est l'instruction à événement. La forme syntaxique est :

<u>when</u> «événement» <u>do</u> «listes-actions».

si la partie «événement» est satisfaite les actions dans «listes-actions» seront exécutées.

I.3.1 - Les actions

Les actions peuvent être :

- des actions conditionnelles (IF ... THEN .. ELSE)
- des transferts entre registres
- des créations d'événements.

I.3.2 - Manipulation du temps

La partie «événement» d' une instruction correspond à un simple changement de valeurs d'une variable ou une séquence d'événements.

Exemple

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

<u>when</u> CLK (0 <u>to</u> 1) <u>do</u> Q := D ;

Un autre exemple est :

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

Cette exemple montre une possibilité de 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 quand la séquence d'événements CKA puis CKB s'est répétée 12 fois, suivie de l'occurrence de l'événement EVA ou 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 quelconques d'une description.

Remarques :

1) Supposons que l'on veuille spécifiér que les séquences suivantes permettent de déclencher la même action :

- E1, E2, E3, E4
- E1, E3, E2, E4
- E1, E2//E3, E4 (E2 et E3 sont des événements simultanés)

la description de ceci en HILO II sera :

```
    when ((E1 then E2 then E3 then E4) or
          (E1 then E3 then E2 then E4) or
          (E1 then (E2 and E3) then E4)
      do action_A1
```

alors ceci est facilement modélisable en graphe de type GRAFCET donné figure 1-8.

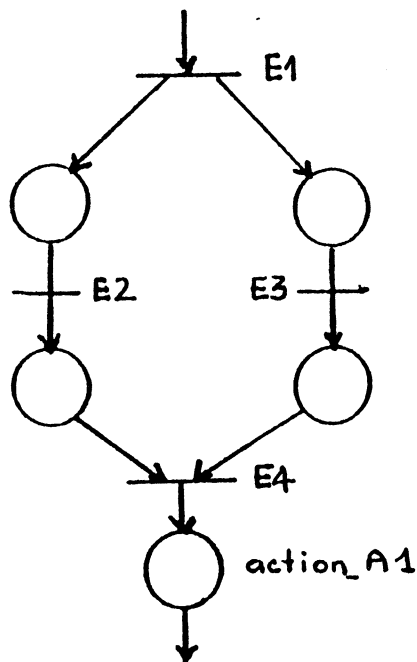


Figure 1-8.

2) On peut aussi remarquer que la temporisation est utilisée pour spécifier l'instant d'exécution de l'action mais qu'il est impossible de temporiser l'action elle-même. La notion simple d'affectation de chronogrammes n'est pas dans le langage.

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

II -CAP/DSDL

II.1 - GENERALITES [RAM 81]

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. Le format textuel de ce graphe est donné figure 1-10.

Exemple :

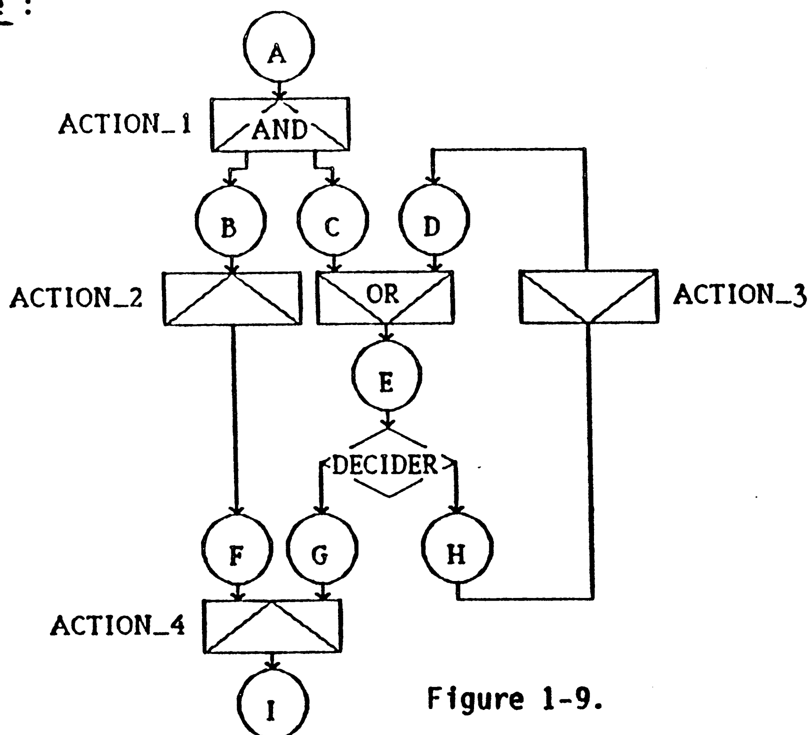


Figure 1-9.

```
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 1-10.

Les places sont déclarées comme des variables de type "place". Avec ce modèle, on peut décrire le fonctionnement de n'importe quel circut logique.

Par ailleurs, CAP/DSDSL offre aussi aux utilisateurs des constructions algorithmiques structurées. Ces constructions sont :

- seqbegin ... end : les instructions comprises dans ce bloc seront exécutées séquentiellement.
- cobegin ... end : les instructions comprises dans ce 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 comme suit :

```
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 : "1-safe", "deadlock free-reusable"... .

II.2 - VARIABLE ET TYPES DE DONNEES

Il existe deux classes de variables en CAP/DSDL :
- variables "explicites"
- variables "implicites".

Les variables dites "explicite" sont celles à mémorisation avec un signal explicitement défini qui active la fonction de mémorisation. Par contre les variables "implicites'" sont celles sans mémorisation et prennent une nouvelle valeur à chaque fois qu'une variable de l'expression de droite change de valeur. Les instructions d'affectations de ces variables 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 dits 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 décharge peut être associé à ces variables.

Exemple

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

Figure 1-11.

Les types de données de base du langage est le "bit (0,1)", étendu aux chaînes de bits. Dans l'exemple précédent, les variables N, Q, NQ, R sont des nappes de 16 fils. La manipulation de données au niveau "bit" est intéressante pour les expressions booléennes, par contre, elle est fastidieuse pour les manipulations d'expressions arithmétiques. Les autres types sont des types de structure de données : tableaux, enregistrements.

II.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 83]. Ces événements sont utilisés dans les deux formes de 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>. Par contre, la deuxième permet de spécifier une action qui est exécutée pendant le niveau "VRAI" de la variable <signal>.

Remarques :

1) La notion de "DELAY" est celle rencontrée dans les langages RTL. Celle-ci est insuffisante pour la manipulation de chronogramme, sinon cette dernière engendre une description non claire et non compacte.

2) La définition de l'objet "événement" qui est donnée dans [RAM 83] nous semble ambigüe car elle fait une confusion entre une condition booléenne et un événement utilisé dans le sens conventionnel.

II.4 - MODELISATION AU NIVEAU TRANSISTOR

[RAM 83] a présenté l'utilisation du langage CAP/DSDL pour la modélisation au niveau transistor MOS. Pour couvrir ce niveau de modélisation les concepteurs du langage ont prédéfini des objets qui sont les suivants :

- PULLUP : transistor de charge
- PULLDOWN : transistor signal
- TRANSFER : transistor de transfert.

Ces objets sont implémentés par des procédures paramétrées par la technologie utilisée (pMOS, nMOS). Les valeurs utilisées sont :

- 0 basse impédance : L0
- 1 bassse impédance : L1
- 0 moyenne impédance : M0
- 1 moyenne impédance : M1
- 0 haute impédance : H0
- 1 haute impédance : H1
- Z

Une étude plus précise sur l'algèbre multivaluée et multiforce est faite dans [CRA 85].

Remarques :

1) L'utilisation de cette approche implique que les concepteurs du langage doivent changer les modèles prédéfinis ou en ajouter d'autres pour satisfaire l'évolution croissante et rapide de la technologie utilisée pour les circuits intégrés.

2) Un modèle de type Réseau 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 (cf. CADOC.LD).
L' approche choisie par CAP/DSDL est assez ambigüe du fait de la coexistence d'un modèle de type RdP et d'objets prédéfinis. Une approche consistant à définir un ensemble de mécanismes suffisamment puissants pour permettre toute description avec éventuellement une création de bibliothèque de modèles et non création d'objets prédéfinis (corps de description inaccessible) nous semble préférable.

III - MODLAN

III.1 - GENERALITES [PAW 81][PAW 82]

Les notions de base utilisées dans le langage MODLAN sont :
- module : une partie d'un système logique
- terminal : un point de communications du module avec l'extérieur.
- interconnexion : une voie de transfert d'informations entre les terminaux.

Un module est décrit par l'un des trois types de description qui sont les suivants :

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

Le langage possède des modules primitifs (PM) tels portes logiques, retards, buffers trois-états, bus bidirectionnel, ...

Un terminal peut recevoir un des deux attributs suivants :

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

Remarque :

Le mécanisme d'attributs est très intéressant et permet de donner plus d'informations sur les objets déclarés en vue d'un traitement particulier. Mais le fait d'avoir prédéfini deux attributs correspondant à des technologies bien particulières nous semble tout à fait inutile dans la mesure où la technologie évolue rapidement.

III.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 (cf. figure 1-12).

Exemple :

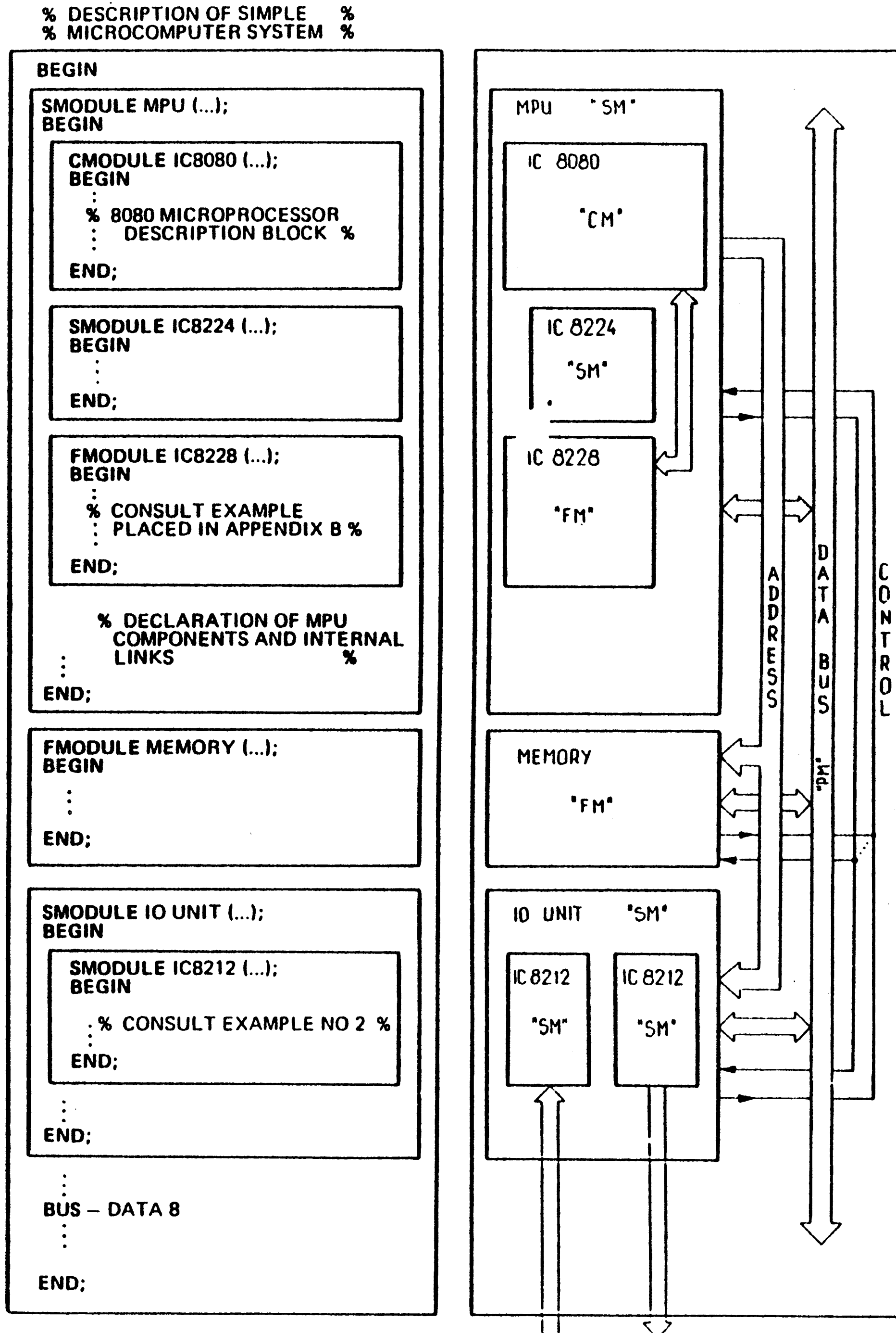


Figure 1-12.

La définition d'une description de type SM étant récursive, ceci, permet donc la hiérarchisation de la description d'un modèle.

- **valeurs manipulées**

Les valeurs manipulées dans ce type de description est au nombre de sept :

- 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 - 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). Un certain nombre des instructions se trouvent dans une partie procédurale de la description et les autres 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 bloc non-procédural 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.

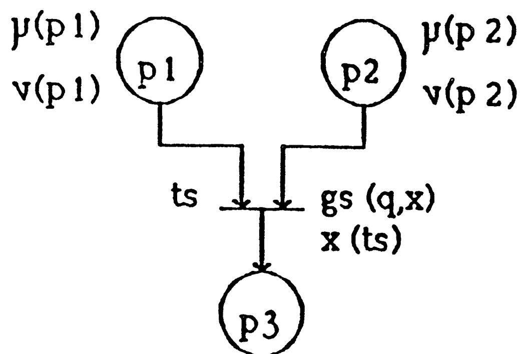
L'utilisateur peut aussi spécifier des contraintes temporelles par des déclarations de type CHECKS. Ces déclarations concernent la stabilité des entrées pendant un intervalle de temps donné.

Dans l'exemple de la figure 1-13a 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 variabl receptrices une valeur de retard. Dans l'exemple ci-dessous (figure 1-13a), la variable INTA prend la valeur 1 après un retard de THD unités de temps.

III.4 - DESCRIPTION A CONTROLE EXPLICITE (CM)

Le modèle utilisé est un graphe de type Réseau de Pétri synchronisé.



$\mu(p)$: 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.

Figure 1-14.

- **actions**

Les actions associées aux places sont décrites en utilisant les constructions pour une description de 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.5 - REMARQUES

1) Pour couvrir tous les niveaux de description de circuit, MODLAN a utilisé 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.

2) Nous avons déjà remarqué pour CAP/DSDL que utilisation d'un modèle de type RdP suffit pour décrire tous les modèles de circuits. En ajoutant le mécanismes d'interconnexion et d'hiérarchisation nous pensons que c'est suffisant pour décrire la structure et le fonctionnement d'un circuit.
La solution prise par le langage CADOC.LD est semblable à cette démarche.

IV - VHDL [SHA 85]

IV.1 - GENERALITES

Le langage VHDL (VHSIC Hardware Description Language) a été défini comme le langage de spécification de matériel pour le support de tous les outils dede 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 les possibiltés de prendre en compte d'autres informations dans leur description telles la technologie utilisée, les fan-in/fan-out, ...

Ces informations seront utilisées pour des traitements spécifiques.

La notion de base du langage est "l'entité de conception". Elle représente un circuit ou une partie de circuit. Au point de vue langage, elle est composée :

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

IV.2 - INTERFACE

L'interface d'une entité (de conception) décrit tous ses 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.

IV.2.1 - Ports d'entrées/sorties

Les ports d'entrées/sorties sont de trois classes.

- entrée
- sortie
- bidirectionnel

Ils sont tous déclarés et typés (type de données)

IV.2.2 - 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 1-15 a comme interface :

```
with ADDER_RESOURCE ;
  - spécification de l'interface
entity FOUR_BIT_ADDER
  - ports
    (A, B : in BIT_VECTOR (3..0);
    cin : in BIT
    cout : out BIT
    sum : out BIT_VECTOR (3..0))

  - paramètres
generic (DELAY : TIME := 36ns)

assertion DELAY   3ns
          SUM'FANOUT < MAX_FANOUT

end FOUR_BIT_ADDER
```

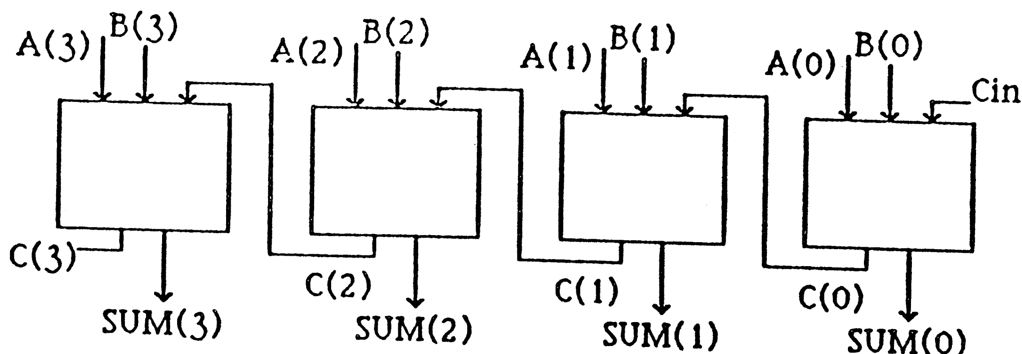


Figure 1-15.

La première instruction de cet exemple est un appel à une entité au sens ADA qui est un "package". Signalons que le but de [SHA 85] est de décrire un additionneur 4-bits puis la décomposer en additionneur 1-bit : démarche descendante. Il a donc décrit un "package" dans lequel il a spécifié toute les informations utilisées pour la description de l'additionneur 4-bits.

Le paramètre générique "DELAY" de type "TIME est fixé à une valeur de 36 ns : l'entité est donc unique dans ce cas.

IV.2.3 - Types de données

Les types de données définis dans VHDL sont :

- les types prédéfinis tels entier, réel, booléen, vecteurs de bits
- les types composés sont définis par l'utilisateur tels énumération, tableau, enregistrement.
- les types physiques tels longueur, temps, voltage, ... , avec l'unité de mesure ; dans l'exemple précédent le paramètre DELAY est de type "TIME" défini. figure 1-16.

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

Figure 1-16.

Remarque

L'utilisateur peut définir par énumération ses propres valeurs logiques, par exemple, {0, 1, Z, U, X} et donc la manipulation des valeurs.

IV.2.4 - 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.

IV.2.5 - 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).

D'autres utilisations sont : spécifications de temps de positionnement, de maintien, des caractéristiques électriques, ...

Exemple

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

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

IV.3 - LE 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)

IV.3.1 - Modèle temporel

Le modèle temporel de VHDL est le même que celui de CONLAN (cf. paragraphe V.3.1). 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.

IV.3.2 - 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 ;
```

IV.3.3 - 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 ;
```

IV.4 - REMARQUES

Ce langage possède des caractéristiques intéressantes : description comportementale, description structurelle, spécification d'attributs et

d'assertions. 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.

V - APPROCHE CONLAN

IV.1 - GENERALITES [BOR 81][PIL 83]

Le projet CONLAN 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), qui a été défini lui-même à partir d'un langage plus élémentaire PRIMITIF CONLAN (pcl).

CONLAN a 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.

V.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 : défintion 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
- 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> ]

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

           [ASSERT <assertions> ENDASSERT ]

           BODY
             [définition de segments locaux]
             [déclaration d'objets internes]
             [invocations d'opérations]
             [modification de syntaxe]

           END <identificateur>
```

V.3 - LANGAGE BCL (Base CONLAN Language)

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

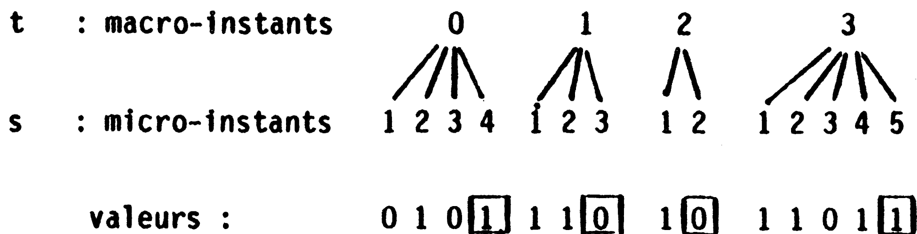
- les valeurs constantes, entières, chaînes de caractères.
- les porteuses qui peuvent contenir une valeur .

V.3.1 - Le modèle temporel

Le modèle temporel défini pour l'ensemble CONLAN est constitué de deux types d'instants : (cf. paragraphe IV.3.1)

- les macro-instants
- les mini-instants.

Exemple



Les valeurs ☐ représentent les valeurs stabilisées de la porteuse et la fin de l'évolution interne des micro-instants. Techniquement, les micro-instants correpondent à des boucles de stabilisation de simulation que l'on rencontre par exemple dans la simulation logique.

V.3.2 - Les 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 positif), nint (entier négatif), bint(m,n:int). Ce dernier 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 les types de valeurs de la séquence.

Exemple

TYPE signal(v: val_type) est la définition du type générique "SIGNAL". Si v = 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 est 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 pour n ≠ 1 est vérifiée, la valeur retournée est VRAI.

V.3.3 - Les types de porteuses

Il existe trois types :
- terminal
- variable
- 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 <-- .

V.3.4 - Les structurations de données

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

V.3.5 - Exemples

Dans les sous-paragraphes précédents, nous avons défini les langage BCL. Dans ce sous-paragraphe nous nous placons 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. La syntaxe % est la syntaxe simplifiée de l'appel de la fonction DELAY qui est définie par le concepteur du langage BCL.

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) &   (ck % (th+1)) then
           stable (d,tsu+th) & stable0(ck%th, tsu)
           & stable1 (ck, th)
     else 1
     endif
  ENDASSERT
  if [ck % (tp -1)] &   [ck% tp] then q:= d, nq :=  d;
  endif
END dff
```

commentaires

- 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. Le premier 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 entiers 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'états de la bascule. Cette condition est que ck vaut 1 à (tp-1) et 0 à (tp) (tp= temps de propagation).

Remarque :

La description fonctionnelle utilise des objets de type VARIABLE. Les instructions sont exécutées si les conditions associées (if ...) sont vérifiées. On retrouve en fait la sémantique des langages de description comportementale/fonctionnelle.
Ce type de description permet de gagner du temps pendant la simulation : les actions ne seront pas exécutées tant que les conditions d'activations ne seront pas vérifiées.

V.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 du BCL. Un ensemble de logiciels 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. [VAI 83] présente cet ensemble de logiciels.

Le niveau de descripton 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 <u>pint</u>.
- inverseur NOT.

- multiplexeur générique avec comme paramètre générique la taille .
- décodeurs génériques avec la taille comme paramètre générique. Ils ont définis un décodeur générique fonctionnant en logique positive et un autre en logique négative.
- un additionneur générique dont les paramètres génériques sont la taille de l'additionneur et le nombre de bits utilisés pour le calcul d'anticipation d retenue.
- les éléments de mémorisation sont décrits comme une interconnexion de portes logiques. Les utilisateurs n'auront plus à connaître leur structure interne. Les éléments sont : NOR-LATCH, NAND-LATCH, RS-LATCH,LSSD-LATCH, MS-JK-FLOP, MS-D-FLOP, MS-T-FLOP, D-FLOP.

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.

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 :

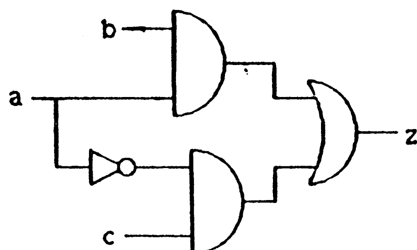


Figure 1-17.

Le type de segment qui est le seul hérité et disponible pour les utilisateurs est le type "DESCRIPTION".

V.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 sembleraient être conçus à partir de CONLAN. De plus certains membres du groupe CONLAN étudient la possibilité d'utiliser des langages comme ADA (cf. paragraphe II.2 du chapitre II).

Enfin, nous voulons seulement signaler le projet CERES (Cascade Environment for the Realisation of Electronic Systems), un projet multinational. CASCADE qui est le langage de base du projet est un langage multiniveau basé sur les concepts de CONLAN [PIL 85]. Le principal outil développé autour de CASCADE est un simulateur mixed-mode.

CHAPITRE III - LE SYSTEME CADOC

I - INTRODUCTION

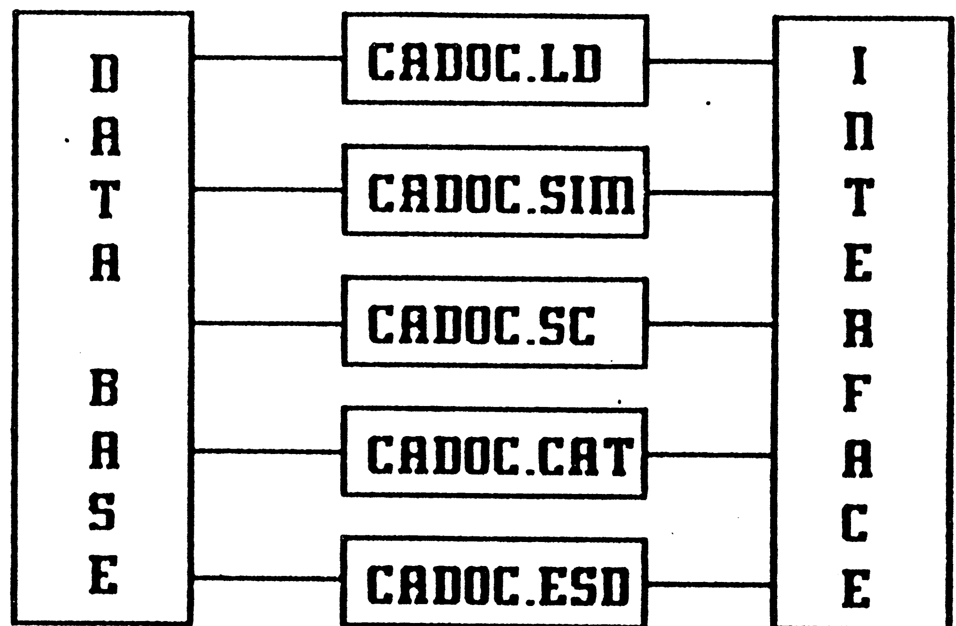
Le sytème CADOC (Computer-Aided Design Of complex Circuits) est un systme de conception assitée par ordinateur pour les circuits complexes. Ce système comme tout autre système de C.A.O offre un ensemble d'outils de base permettant de décrire des circuits et de valider leur conception. Les outils de base de ce système sont :

- le langage de description fonctionnelle CADOC.LD que nous allons présenté exclusivement dans ce chapitre.
- le simulateur fonctionnel CADOC.SIM.

D'autres outils seront intégrés dans ce système. Ils sont en cours de spécification, comme CADOC.CAT (outils de préparation du test, cf. section II), ou en cours de réalisation, comme CADOC.SC (compilateur de silicium) [HAN 85]. L'architecture finale du système est donnée en figure 1-18.

Le langage CADOC.LD est un langage de description fonctionnelle de circuits développés au sein du laboratoire "Circuits et Sytèmes" (IMAG). Il n'est dérivé d'aucun langage de programmation. Par contre, l'étude que nous avons faite sur les langages de description de matériel (ou CHDL : Computer Hardware Description Language), nous a permis de tirer un certain nombre d'idées pour la conception du langage CADOC.LD. D'autre part, nous avons aussi tiré profit du savoir-faire existant dans le laboratoire LCS en ce qui concerne les langages de description de systèmes [MOA 81] basés sur les réseaux de Pétri ou le GRAFCET.

CADOC SYSTEM



LD -> DESCRIPTION LANGUAGE
SIM -> FUNCTIONAL SIMULATOR
SC -> SILICON COMPILER
CAT -> COMPUTER AIDED TEST
ESD -> EXPERT SYSTEM FOR DIAGNOSIS

Figure 1-18.

Le langage CADOC.LD a été conçu en tenant compte des caractéristiques suivantes :

1) description hiérarchisée de circuits pour faire face aux circuits complexes qui ne peuvent plus être décrite facilement au niveau de leur réalisation détaillée (par exemple au niveau logique).

2) description temporelle très précise même à un haut niveau de description. Les concepteurs de circuits complexes doivent pouvoir décrire exactement les comportements temporels de son circuit à des fins de simulation multiniveau lorsque ce circuit est interconnecté avec d'autres circuits décrits à un niveau bas [CRA 85].
Cette deuxième caractéristique est implémentée dans le langage par l'utilisation d'un graphe interprété et temporisé, nommé GIT et la manipulation d'objets abstraits temporisés appelés échéanciers. Ces derniers représentent les chronogrammes des signaux du circuits.

Dans l'exemple donné figure 1-19a,le graphe défini l'évolution temporelle du ciruit : le circuit est synchronisé sur le front montant de l'horloge H. les actions qui sont rattachées aux noeuds (nommés places) sont des affectations de chronogrammes (échéanciers).

3) description du circuit à ses différents états de conception. La méthode de conception modulaires des circuits complexes est largement admise. Cette méthode consiste à partitionner le circuit en blocs fonctionnels ; la description de chaque bloc sera de plus en plus détaillé à chaque évolution de la conc .tion du circuit jusqu'à sa réalisation complète.
Pour assurer la cohérence entre les spécifications du circuit à ses différentes phases de conception, il est nécessaire d'utiliser le même langage pour toutes ces spécifications. Le langage CADOC.LD a été conçu pour permettre aux concepteurs d'utiliser une telle méthode.

4) le langage CADOC.LD ne possède aucun élément prédéfini à signification matérielle tel que l'on peut rencontrer dans les langages dits langages RTL (Register Transfer Level). Par conséquent, les concepteurs doivent pouvoir définir librement tous les éléments qu'ils utilisent. Nous appelons **"ressource fonctionnelles"** ces éléments. Une ressource fonctionnelle est définie soit par ses fonctions, soit par composition d'autres ressources fonctionnelles.
Pour éviter une ambiguité avec la signification communément utilisée dans le domaine du test en ce qui concerne "description structurelle", nous utiliserons dans cette thèse "description partitionnée" pour faire référence au deuxième type de description en langage CADOC.LD. Nous garderons "description structurelle" pour le test.

5) D'après ce qui précède, on peut associer à une ressource fonctionnelle, décrite par composition (hiérarchisée), un arbre de composition (cf. figure 1-19b).

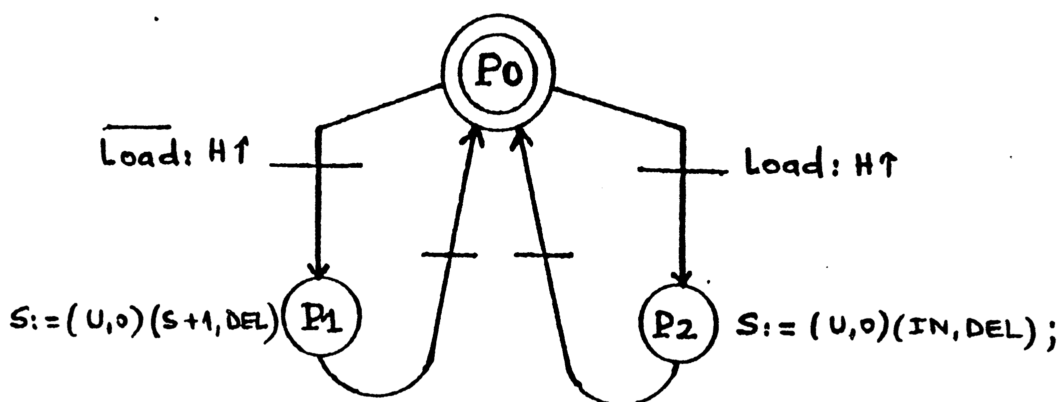


Figure 1-19a.

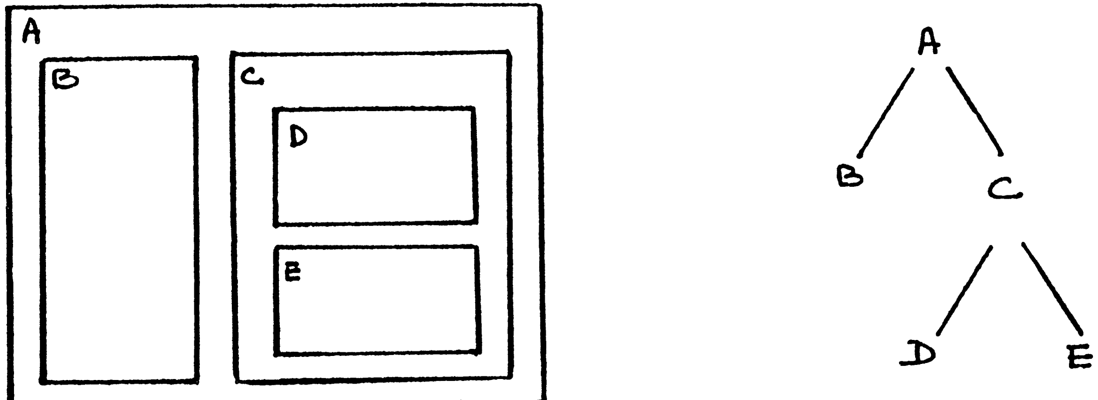


Figure 1-19.

La ressource A est décrite à partir des ressources B et C ; la ressource C étant elle-même décrite à partir de D et E. Les niveaux de description des ressources-feuilles B, D et E peuvent être différents. Le langage CADOC.LD est donc un langage permettant une description hiérarchique multiniveau.

6) Les descriptions des ressources constituantes peuvent être compilées séparément [TIA 85]. Le langage CADOC.LD inclut dans sa définition des mécanismes de compilation séparée. Les résultats de compilation peuvent être mis dans une bibliothèque et peuvent ainsi être réutilisés par d'autres concepteurs ou par lui-même lors de la définition d'autres ressources.

7) La caractéristique précédente n'est pas intéressante sans la possibilité de paramétrer les descriptions des ressources fonctionnelles. Nous avons appelé ces ressources, ressources

génériques fonctionnelles. En réalité, on utilise donc un ou plusieurs exemplaires d'une ressource générique fonctionnelle lorsqu'on définit une autre ressource générique fonctionnelle.
En d'autres termes, on décrit avec le langage CADOC.LD un modèle de circuit qui est une ressource générique fonctionnelle. Lors de l'utilisation d'une ressource générique fonctionnelle, on doit fixer les valeurs de ses paramètres..

Avec ces caractéristiques, le langage CADOC.LD est un langage de description fonctionnelle très souple et simple d'utilisation présentant un aspect très positif sur la précision temporelle compte tenu du niveau d'abstraction.
Il est adapté aussi bien à une démarche de conception ascendante qu'à une démarche de conception descendante.

Par ailleurs, il semble être bien accueilli par les concepteurs de circuits intégrés.

Le compilateur et le simulateur fonctionnel sont écrits en langage PASCAL. La première version du système CADOC est disponible sur trois types de machines :

- BULL DPS8 sous Multics
- VAX 11/780 sous le système VMS
- HP 540 sous le système UNIX.

II - TYPES DE DONNEES ET VARIABLES

Le langage CADOC.LD est un langage dit "fortement typé", c'est-à-dire, tous les ojets utilisés dans le langage doit voir un type. Nous présentons dans ce paragraphe les types de données qui peuvent attribués aux variables utilisées dans une description d'une RGF.

On distingue deux grandes catégories de types de données : les types de données dits simples et les types de données dits structurés (ou de structure de données).

II.1 - LES TYPES SIMPLES

Les types simples sont les types "**booléen, entier, front, et énuméré**" ; les trois types étant étendus par les valeurs symboliques X, Z, U dont les significations sera indiquée plus loin. Pour chaque type, on donne entre parenthèses le(s) mot(s) réservé(s) le désignant.

III.1 - **Types booléen** (BOOL, BOOLEEN)

Ce type est un type comportant 5 valeurs qui sont :

- V : représentant la valeur logique "1"
- F : représentant la valeur logique "0"
- X : représentant la valeur "indifférente",c'est-à-dire, V ou F.
- U : représente une valeur inconnue, conséquence soit d'une non initialisation soit d'un comportement de type transitoire.
- Z : représente l'état "Haute-Impédance".

Les opérateurs définis sur le type "booléen" sont :
- opérateur unaire de négation "NON"
- opérateurs binaires de comparaison "<", ">", "≤", "≥", "=", "≠"
- opérateurs binaires "ET", "OU", "OUX"

Remarque

Lors de la définition de ces opérateurs, on est confronté au problème du choix de leur sémantique : soit 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.
Mais dans la mesure où les défintions choisies pour ces opérateurs ne conviennent pas à une application donnée, l'utilisateur peut parfaitement utiliser la table de son opérateur par l'intermédiaire des constructions de type "CHOIX" (cf. paragraphe III.2.2) ou par des appels à des ressources algorithmiques (cf. paragraphe IV).

III.1.2 - Type entier (ENT, ENTIER)

Le type entier est une extension du type classique (entiers relatifs) avec les valeurs symboliques X, Z, U. X représente dans ce cas toute valeur entière.

Le sous-type "intervalle entier" est défini à partir du type ENTIER. Les bornes de l'intervalle sont des entiers. Un intervalle est de la forme :

⌊borne-inf .. borne-sup⌋

Les vérifications des bornes sont faites pendant la simulation.

Les entiers peuvent être lexicalement donnés sous plusieurs formes :
- décimale signée : exemple 328, -1024, ...
- hexadécimale signée : exemple #0F, #-0F (représentation en complément à 2 de -1), -#FF
- binaire signée : &1110, &-1110, -&-1110

En sortie, les entiers sont donnés sous forme décimale signée.

Les opérateurs sur les valeurs "entier" sont les opérateurs classiques tels : "<", ">", "=", " ", " ", "≠", "+", "-", "div", "mod", "*", "**". Ces opérateurs sont étendus pour prendre en compte lse valeurs symboliques X, Z, et U.

Les entiers sont **sans limitation de taille**, indépendamment de la machine hôte (choix d'un mode de mémorisation en listes, transparent à l'utilisateur) [HAN 84].

III.1.3 - Type front (FRONT)

Le type "front" est un type de deux valeurs symboliques qui sont :
- M : représente le front montant d'un signal
- F : représente le front descendant d'un signal

Aucune opération n'est définie sur ce type. Il a été défini pour des raisons de facilité de description de certains dispositifs sensibles à des variations de niveaux.

III.1.4 - Type énuméré

Un type énuméré est défini de manière classique par la list des éléments le composant ; cette liste étant étendue avec les valeurs symboliques X,Z,U. Les seuls opérateurs définis pour les types énumérés sont les opérateurs d'égalité, de différence et d'appartenance.

Exemple :

Les modes d'addressage d'un microprocessuer classique peuvent être décrits par le type énuméré suivant :

ADRESSAGE = (imm, dir, inx, ext, impl, rel) ;

Si ADX est une variable de type ADRESSAGE, il est possible d'écrire des expressions de la forme : ADX = impl.

III.2 - TYPES STRUCTURES

Les types structurés sont construits par l'intermédiarire des constructeurs d'enregistrements et de tableaux.

III.2.1 - 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 classiques, il est possible de définir des champs conditionnels. Ilm faut noter que les champs conditionnels sont tous accessibles (en cours de simulation) quelles que soien les valeurs des sélecteurs de champs. Dans le cas d'un enregistrement conditionnel, les secteurs sont de type de base. L'accès aux champs d'unenregistrement se fait en postfixant le nom de la variable par le nom du champs.

Exemple :
Défintion d'une micro_istruction composée de 2 champs :
- un champs mode d'adressage dont le type a été défini en III.1.4
- un champs opération qui est un entier sur 16 bits :

```
MICRO_INSTR = enreg
                 ADR : ADRESSAGE ;
                 OP : [0 .. 2**16 - 1] ;
              fin ;
```

III.2.2 - Les tableaux

Le langage CADOC.LD autorise la définition de tableaux multi-dimensionnés. Seules les bornes supérieures sont données, l'indice correspondant variant de 0 à la valeur maximale.

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 par l'enregistrement "micro_instr", pourra être défini par :

MEM = tableau [511] de MICRO_INSTR.

III - DESCRIPTION D'UNE RESSOURCE FONCTIONNELLE

III.1 - EN-TETE D'UNE RGF

Dans l'en-tête d'une ressource générique fonctionnelle, on trouve les informations suivantes :

- **l'identificateur de la RGF**
- **la liste de ses ports d'entrée/sortie (cette liste est optionnelle)**
- **la liste des paramètres de description, le cas échéant**

Exemple

L'en-tête de l'U.A.L de la figure 1-20 est la suivante:

rgf ALU_GEN (EA,EB,CIN,CODOP,S,COUT param N:entier);

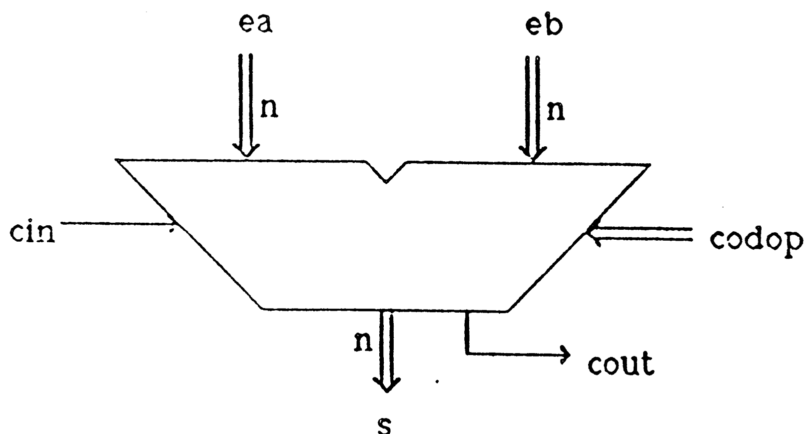


Figure 1-20.

III.2 - PARTIE DECLARATIVE D'UNE DESCRIPTION

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.

III.2.1 - Les 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

III.2.2 - Les 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.

Exemple

entier_255 = ⌊0...255⌋

entier_c2_255 = ⌊-128..127⌋

III.2.3 - Les 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. III.3.2.2).

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 vérification lors de la compilation des descriptions.

III.2.4 - Les 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 parties : une expression booléenne et un message d'erreur, ce message sera imprimé si l'expression booléenne n'est pas vérifiée.

- contraintes sur les paramètres

Une RGF étant un modèle paramétrisable 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 de chaque 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 des domaines 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
 asl :
   N1*N2 <65535 :
   ECRIRE ('assertion asl, 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 1-21, 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. Cette extension est particulièrement présentée dans [CRA 85].

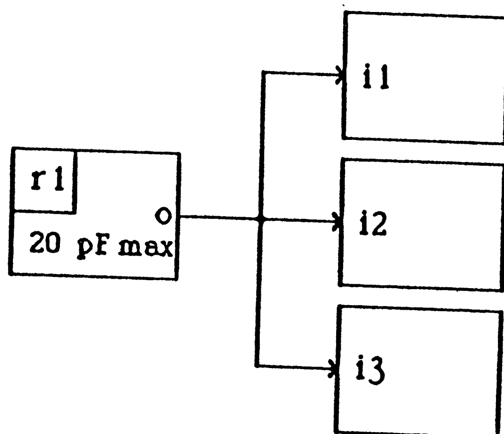


Figure 1-21.

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.

III.2.5 - Exemple

La partie déclarative de l'UAL donnée précédemment (figure 1-20) 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<15): ECRIRE ("assertions AS1 du mode ALU_GEN non vérifiée") ;
```

III.2.6 - Les RGF utilisées et de leurs occurences

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 (types, variables, ...) :

- 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. paragraphe V),
- les occurences de ces RGF avec la liste des variables et paramètres effectifs (rubrique RCONST),
- la liste (éventuelle) des connectiques explicites (rubrique CONNECT).

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 occurence d'une RGF se fera en donnant le nom de l'occurence suivi du nom de la RGF, de la liste des ports d'E/S et paramètres effectifs.

Exemple :

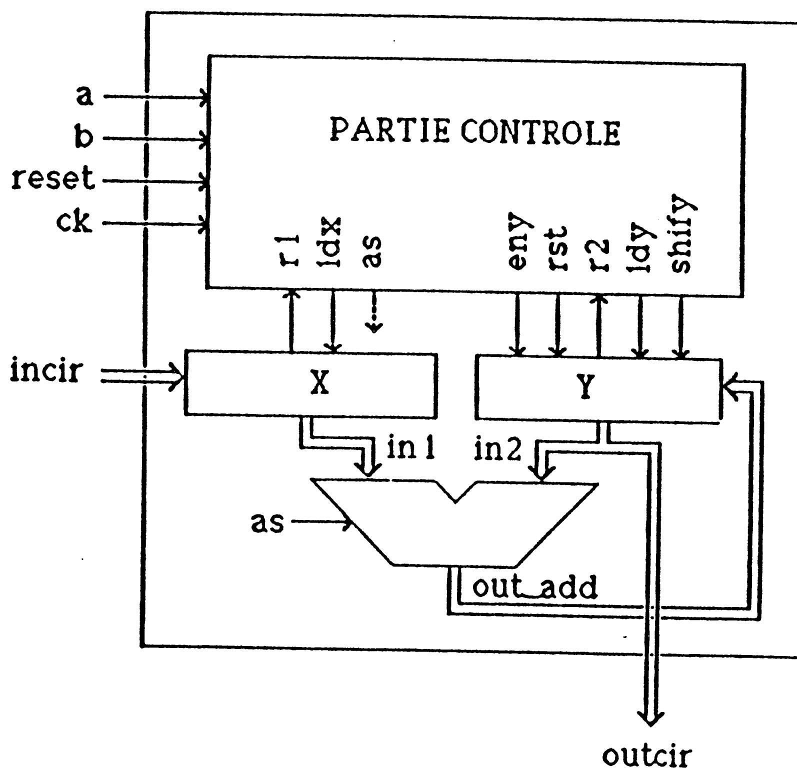


Figure 1-22.

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
  rl, r2 : BOOL ;
  ldx, ldy, rst, shfy, eny, as : BOOL ;
  out_add, inl, in2 : int_8 ;
RGF
  crtl_gen (ckl : FRONT ;
            reset, inl, 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 (inl, 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, rl, r2, ldx, ldy, rst, shfy, eny, as) ;
  x : reg_gen (incir, inl, ldx ; 8) ;
  y : srr_gen (ldy, shfy, rst, out_add, zero, in2, - ; 8) ;
  add_sub : ual_gen (inl, in2, as, out_add) ;
CONNECT
  in2 = outcir ;
```

III.2.7 - Les connectiques

Dans l'exemple précédent, nous avons vu qu'une des manières possible pour effectuer une interconnexion entre 2 ports distincts était de donner le même nom effectif à ces ports dans la déclaration des occurences des ressources ; cette méthode est utilisée figure 1-23a et 1-23b.

L'autre méthode revient à définir l'interconnexion entre deux variables dans la rubrique des connectiques explicites. Cette méthode est illustrée figure 1-24a et 1-24b.

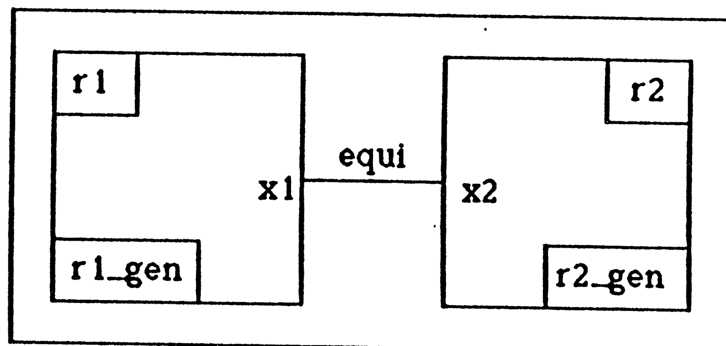


Figure 1-23a.
schéma du circuit.

```
...
VARINT
    equi : tl
...
RGF
    rl_gen (...; xl : tl ;...) ;
    rl_gen (...; x2 : tl ;...) ;
RCONST
    rl : rl_gen (...; equi ;...) ;
    r2 : r2_gen (...; equi ;...) ;
...
```

Figure 1-23b
connectique implicite,
partie déclarative

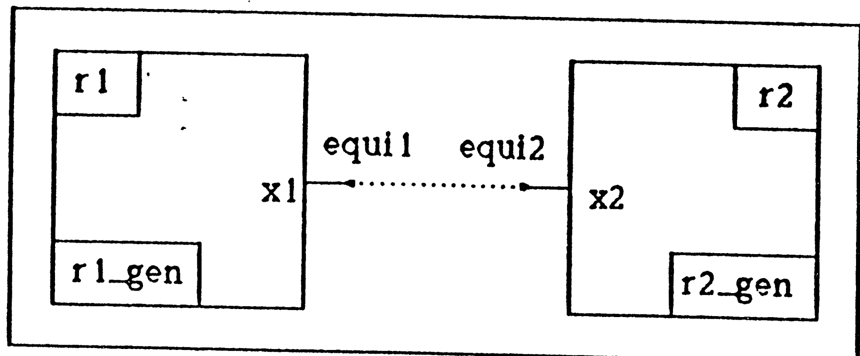


Figure 1-24a

```
...
VARINT
     equi1, equi2 : t ;
...
RGF
     r1_gen (...; x1 : t1 ;...) ;
     r2_gen (...; x2 : t2 ;...) ;
...
RCONST
     r1 : r1_gen (...; equi1 ;...) ;
     r2 : r2_gen (...; equi2 ;...) ;
...
CONNECT
     equi1 = equi2 ;
```

Figure 1-24b.

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.

III.3 - PARTIE FONCTION D'UNE DESCRIPTION

III.3.1 - Modes de définition de la fonction

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

- **représentation partitionnée** : juxtaposition des comportements des sous circuits composants (figure 1-25a) dans le cas d'une description composé 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.

- **représentation comportementale** : utilisation des primitives de définition d'un comportement (figure 1-25b) 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.

- **repésentation mixte** : utilisation simultanée de sous-circuits et d'un graphe interprété temporisé (figure 1-25c) : 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.

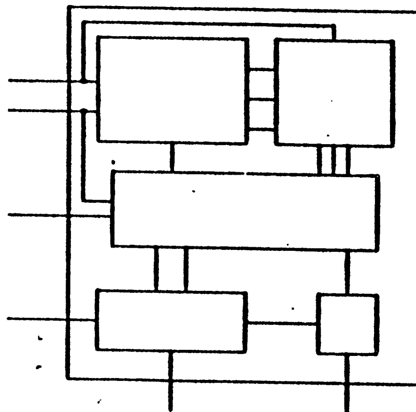


Figure 1-25a.
Description partitionnée;

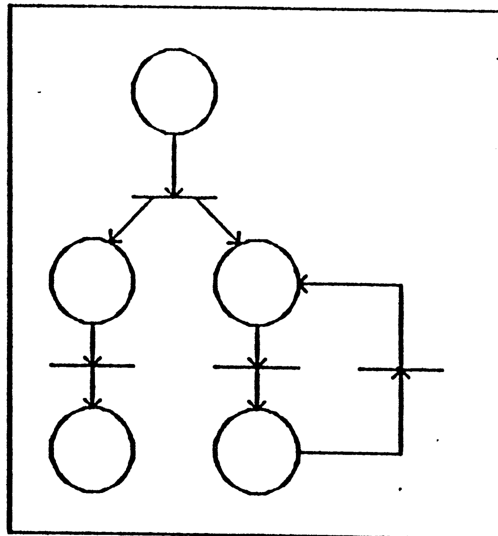


Figure 1-25b.
Descriiption comportementale

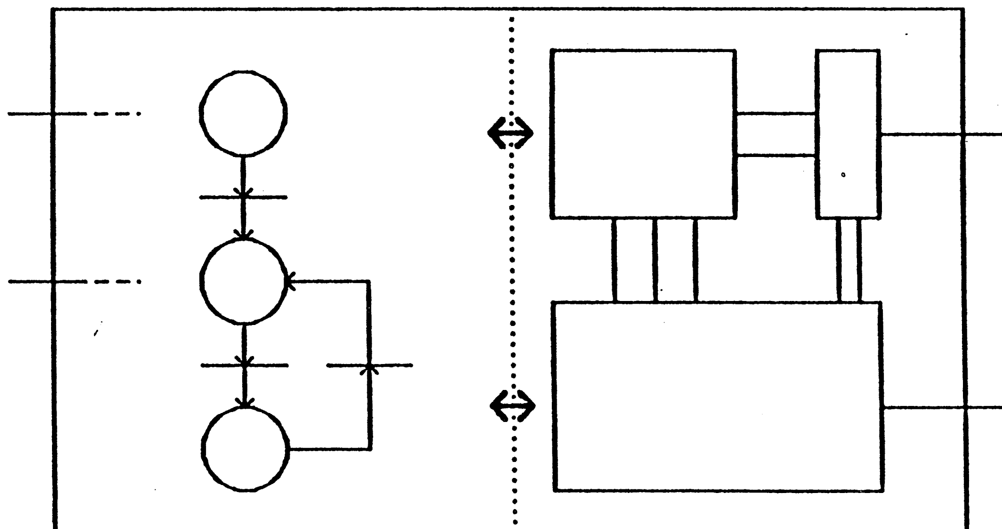


Figure 1-25c.
Description mixte

Remarques

- Dans tous ces modes de description, on peut définir des assertions dynamiques (cf. III.3.3).
- Dans le cas d'une description partitionnée, on se reportera au paragraphe III.2.6 : partie déclarative des RGF composées.

III.3.2 - Graphe interprété temporisé (GIT)

Le comportement d'un circuit 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. Il existe deux modes de représentation équivalents des GIT : un modèle graphique et un modèle textuel ; pour des raisons de lisibilité, nous utiliserons en général le modèle graphique.

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. Un squelette de GIT (non interprété, ni temporisé) est donné figure 1-26a dans sa représentation graphique, et figure 1-26b dans sa représentation textuelle.

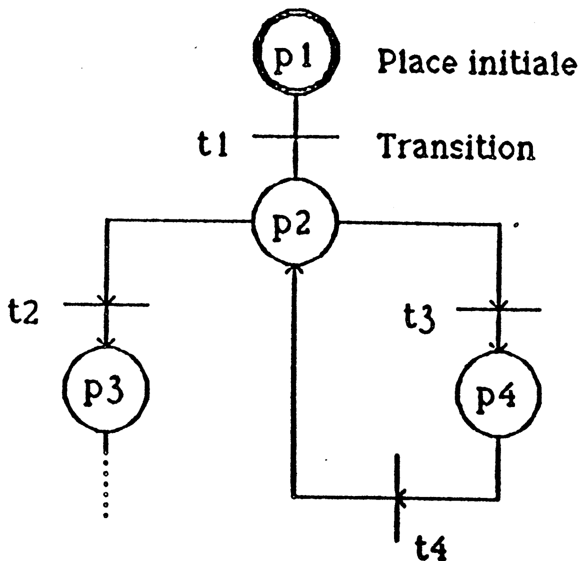


Figure 1-26a
(représentation graphique)

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

Figure 1-26b
(représentation textuelle)

III.3.2.1 - Expressions - 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 et 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 peuvent pas être exprimées 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
- instructions répétitive : tantque-faire-finfaire,
- instructions de choix : choix-dans-autres-finchoix,
- instructions 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 partant depuis l'instant 0 (début de simulation) incrémenté de 1 en 1 jusqu'à la date de fin de simulation
- présence d'une variable CST indiquant la date courante.

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 la date de la première occurence d'un front montant sur les valeurs de la variable, par rapport à l'instant courant de simulation.

- cas d'une variable booléenne : étant 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

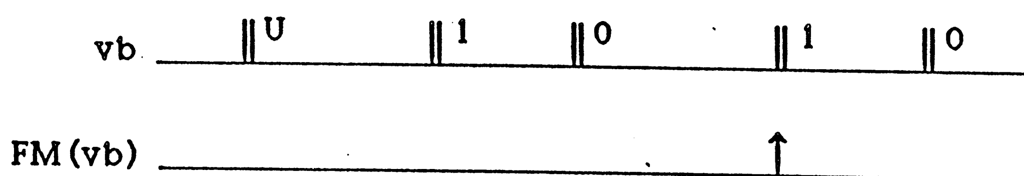


Figure 1-27.

- 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).

La généralisation de cette fonction est la fonction "fmx" avec les paramètres suivants : - variables, - rang relatif de l'occurence du front montant, - instant de référence.

Exemple :

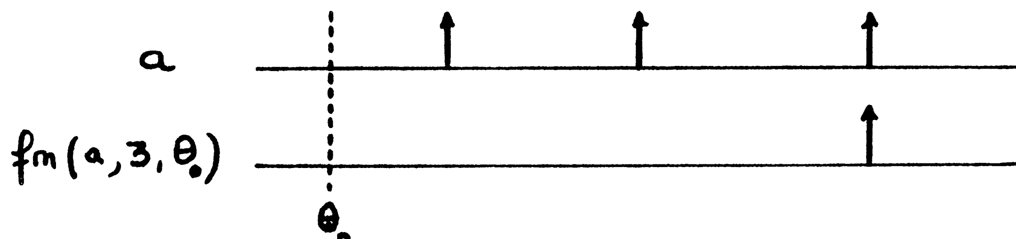


Figure 1-28.

Cette fonction sera particulièrement utilisée pour l'étude du test (cf.section II).

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

Cette fonction est similaire à la fonction fm.

*** fonction change (ch ou change)**

La fonction "change" est appelée avec comme paramètre une liste de variables (non de classe ALGO) et désigne le premier instat après l'instant courant (de simulation) où une des variables change de valeur.

"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 III.3.2.6 sur l'interprétation des GIT pour plus de précision sur l'activation et la désactivation des places.

Exemples

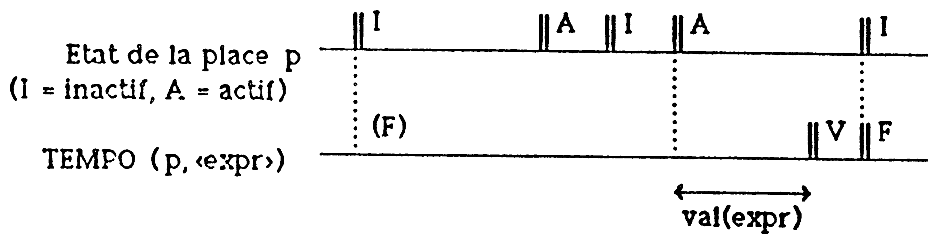


Figure 1-29.

*** 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

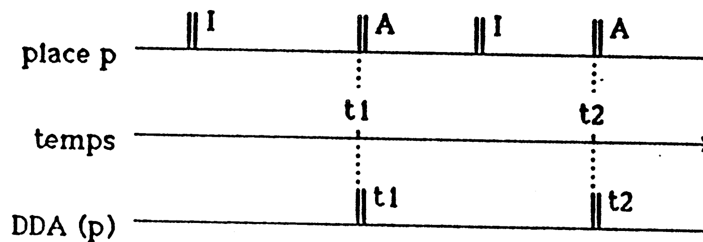


Figure 1-30.

*** 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

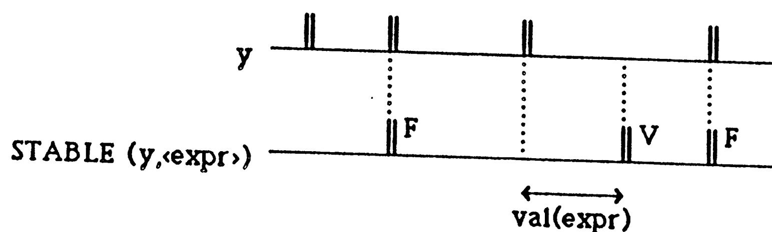


Figure 1-31.

* **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 :

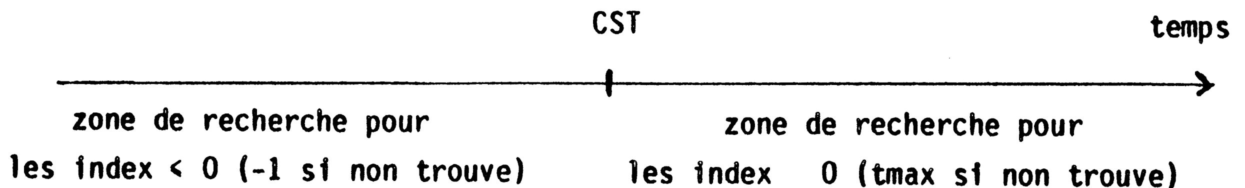


Figure 1-32.

- **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 au 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 (on verra ultérieurement les vérifications de double affectation) 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 1-33a et 1-33b donnent deux exemples de chronogrammes et leurs traductions en CADOC.LD.

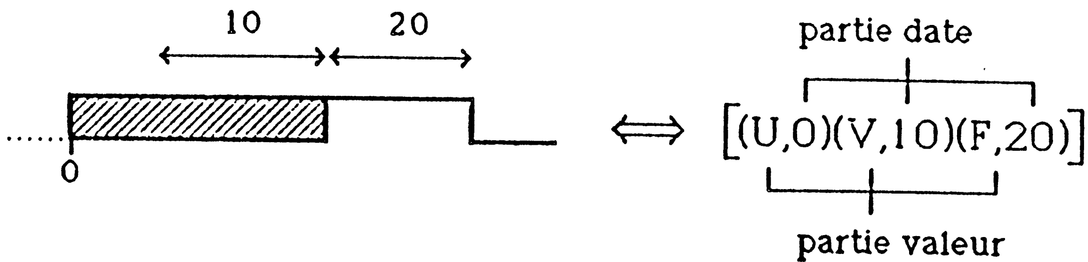


Figure 1-33a : Chronogramme (exemple 1)

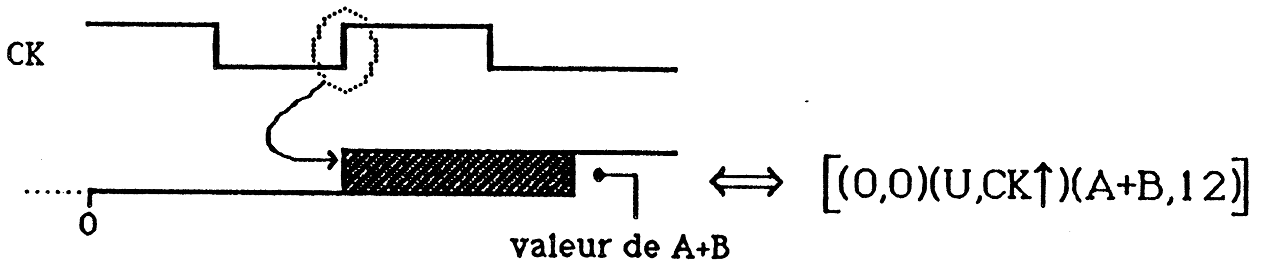


Figure 1-33b : Chronogramme (exemple 2)

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

La partie valeur 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 est soit une expression classique soit fait référence à des fonctions prédéfinies permettant d'accéder à des dates. 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 définie dans le chronogramme. Si la première expression date du chronogramme est relative, la date absolue correspondante est calculée à partir de l'instant d'évaluation du chronogramme.

Exemple

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

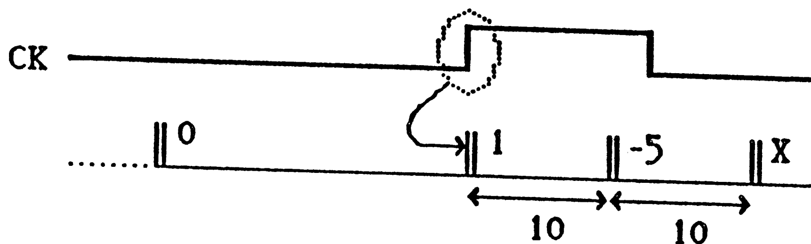


Figure 1-34.

III.3.2.2 - Types d'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.

Exemple1

Le résultat de l'exécution de S := ⌊(U,0) (NOT E, del1)⌋ sera :

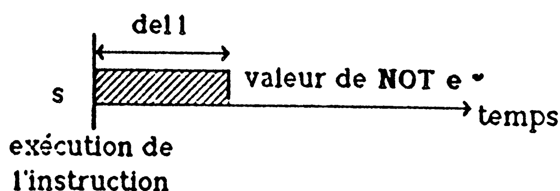


Figure 1-35.

Exemple2

nb_un :=
DEBUT
i:=0 ;
.
. /algorithme donné en 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 affectée 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.

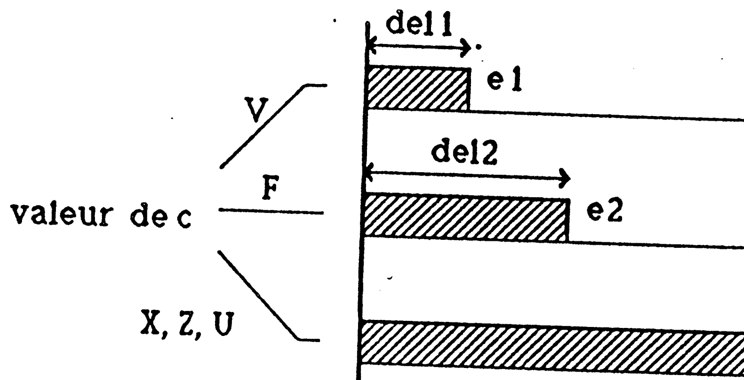


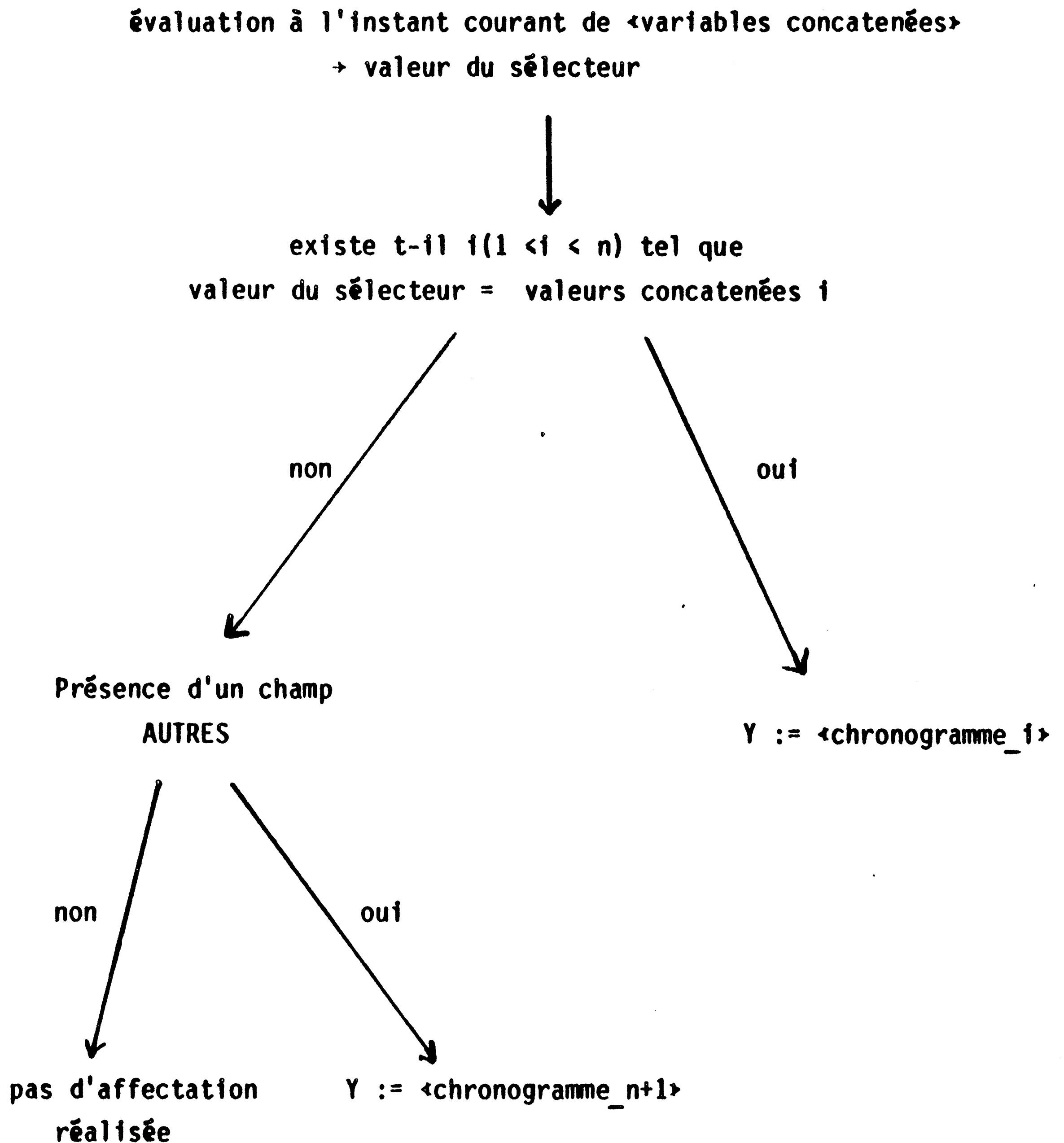
Figure 1-36.

- **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 concatenées> DANS
       <valeurs_concatenées_1> : <chronogramme_1> ;
           ...
       <valeurs_concatenées_n> : <chronogramme_n> ;
       [AUTRES : <chronogramme_n+1> ;]
     FINMUX ;
```

La sélection de la valeur affectée à Y se fera 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.3.2.3 - Places

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

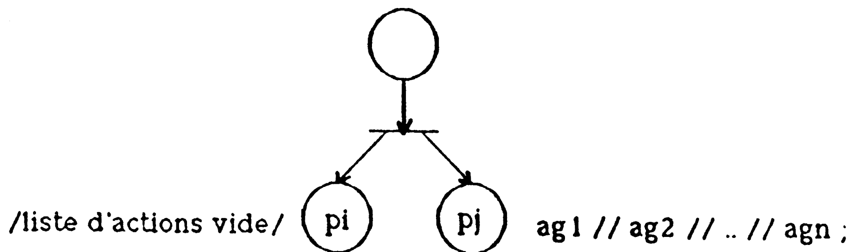


Figure 1-37.

Les actions associées à une place seront exécutées à l'instant d'activation de la place (voir paragraphe III.3.2.6).

III.3.2.4 - 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

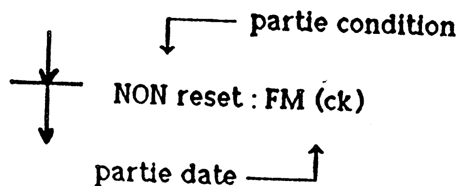


Figure 1-38.

III.3.2.5 - Modèle temporel

- Modèle temporel externe (côté utilisateur)

La notion utilisée dans le langage CADOC.LD est celle de "unité de temps" : le temps est une suite discrète de valeurs entières strictement

croissantes et d'incrément 1. Cette suite commencera à l'instant 0 qui indique le début de la simulation et se terminera à la valeur donnée par le temps maximal de simulation. Dans le temps externe, il n'existe pas d'instants accessibles entre 2 instants consécutifs.
Les manipulations se font par l'application de fonctions prédéfinies sur des variables.

- **Modèle temporel interne (coté simulateur)**

De manière interne, l'intervalle entre 2 instants (externes) consécutifs absolus est subdivisé en un nombre à priori illimité de micro-instants (figure 1-38). Une date interne n'est donc pas un singleton mais un couple (date externe, microinstant).

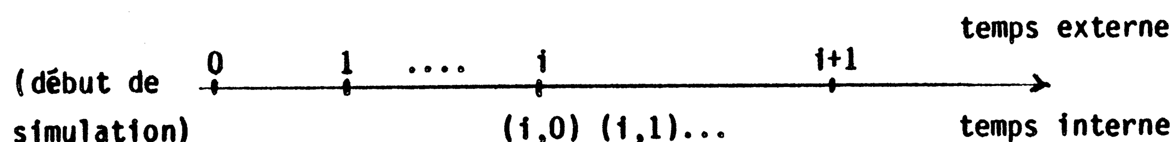


Figure 1-39.

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

III.3.2.6 - Règles d'évolution - simulation

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 marquée par un jeton, une place initiale est indiquée par deux cercles concentriques :

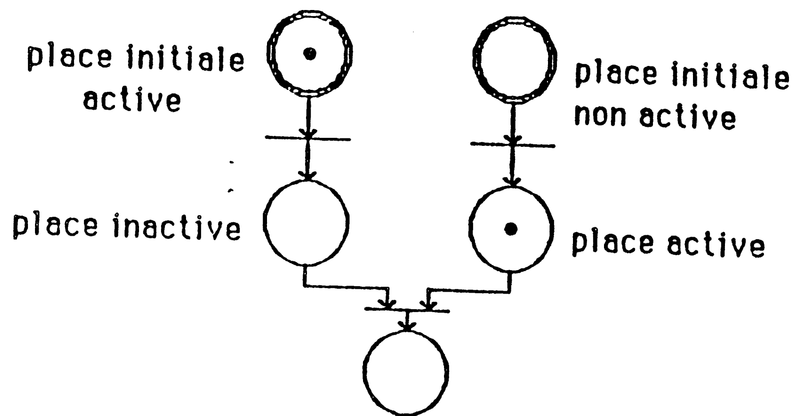


Figure 1-40.

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 1-41a et 1-41b).

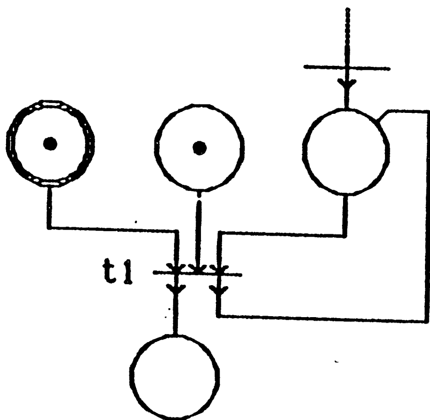


Figure 1-41a : T1 non validée

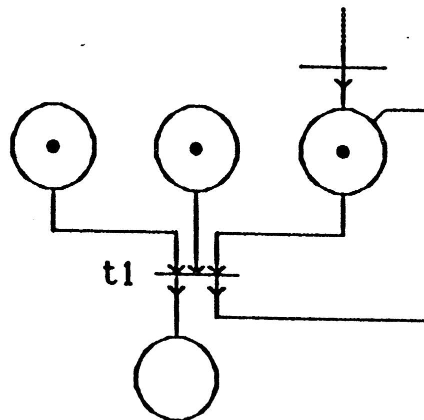


Figure 1-41b : T1 validée

- Transition franchissable

Une transition sera franchissable à l'instant θ si et seulement si :

- elle est validée à θ
- la condition booléenne de sa réceptivité est vraie cet instant.

L'instant θ 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, notée "e".

- **Appel aux fonctions FM ou FD**

Lorsque la partie date d'une réceptivité est spécifiée par un appel à la fonction FM (respectivement FD), la transition est franchie à l'instant correspondant à la première occurence d'un front montant (respectivement descendant) de la variable correspondante après l'instant courant de simulation.

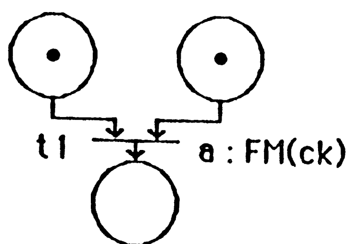


Figure 1-42a

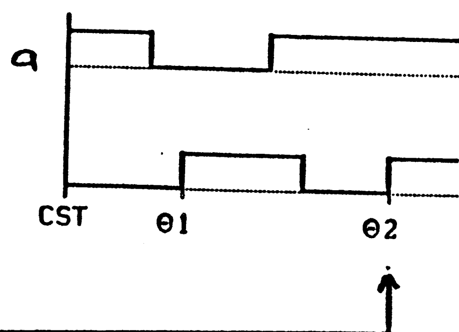


Figure 1-42b

- **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.

- **Partie date non spécifiée**

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

III.3.3 - Assertions dynamiques

Les assertions dynamiques ont été introduites en CADOC.LD pour permettre la spécification d'invariants à vérifier en cours de simulation ou des comportements particuliers du type RESET.

- **spécification d'invariants dynamiques** :

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 place ou par une liste d'évènements ; dans le cas ou la propriété donnée sous forme d'une expression booléenne n'est pas 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 1-43a ; les quatres assertions correspondantes sont définies dans la figure 1-43b.

	SHIFT	LOAD
SHIFT	ΔS	ΔSL
LOAD	ΔLS	ΔL

Figure 1-43a.

<u>ASDYN</u>

ad1 : <u>TOUTE</u> : FM(shift) : <u>DOCX</u>(shift,0) - <u>DOCX</u>(shift,-1) > ΔS
: <u>ECRIRE</u> ("AD1 non respectée au temps ", CST) ;
ad2 : <u>TOUTE</u> : FM(shift) : DOCX(shift,0) - DOCX(load,0) > ΔSL :
: <u>ECRIRE</u> ("AD2 non respectée au temps ", CST) ;
ad3 : <u>TOUTE</u> : FM(load) : DOCX(load,0) - DOCX(load,-1) > ΔSL :
: <u>ECRIRE</u> ("AD3 non respectée au temps ", CST) ;
ad4 : <u>TOUTE</u> : FM(load) : DOCX(load,0) - DOCX(shift,0) > ΔLS :
: <u>ECRIRE</u> ("AD4 non respectée au temps ", CST) ;

Figure 1-43.

- **<u>Modélisation de signaux du type RESET</u>** :

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 1-44c).

En effet, on retrouve mélangé la description du circuit, son comportement "normal" (figure 1-44a) du circuit ainsi que son comportement d'exception (figure 1-44b). Pour éviter cette confusion, ce comportement d'exception sera enoncée dans une assertion (figure 1-44d) dont les informations seront :

- la liste des places ou cette assertion sera valable (le mot réservé <u>toute</u> 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.

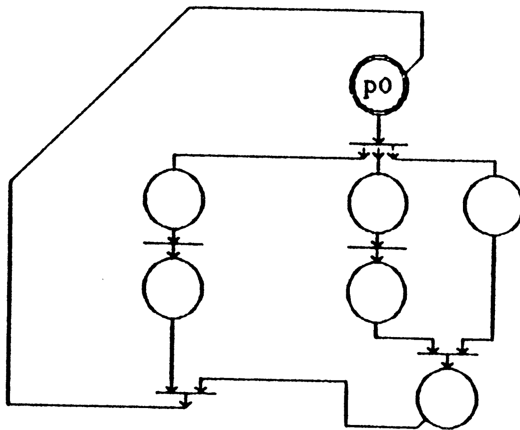


Figure 1-44a.
squelette du GIT modélisant
le comportement "normal" du circuit

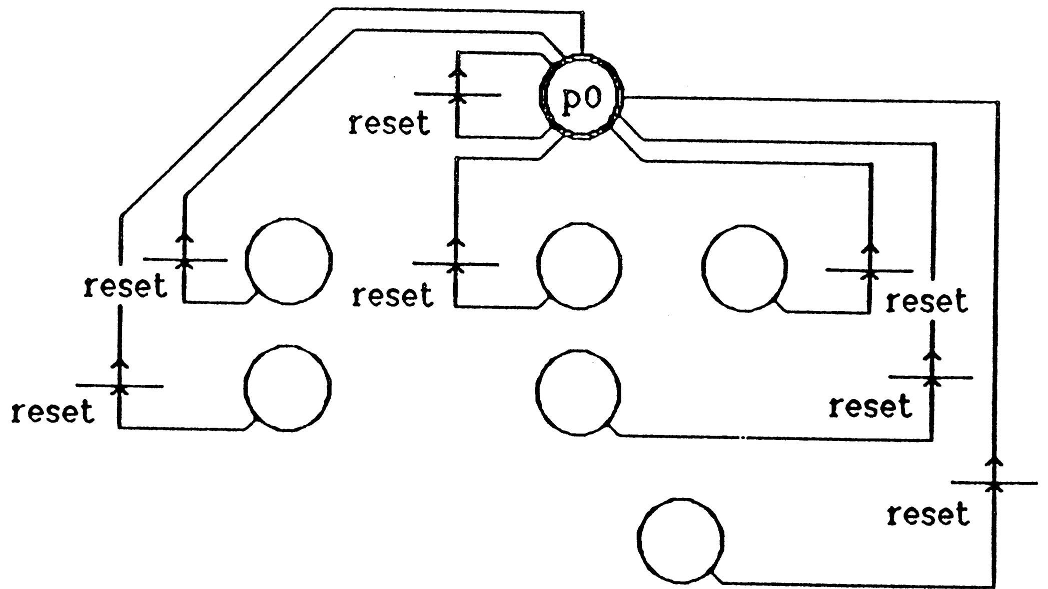
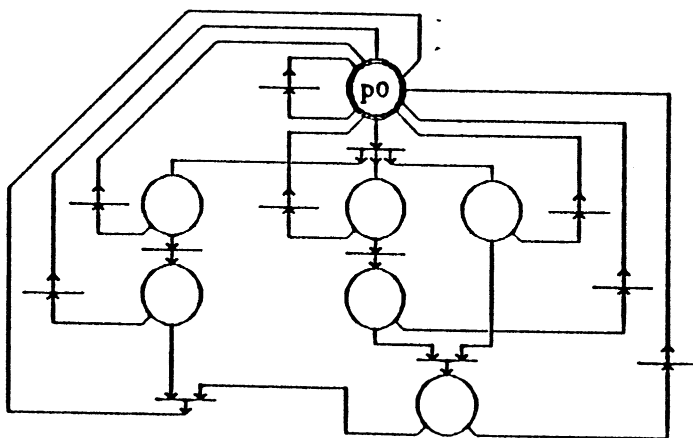


Figure 1-44b.
squelette du GIT modélisant
l'effet RESET.



ass1 : TOUTE reset : TOUTES_PLAC

à tout instant on doit
vérifier si un signal "reset
est arrivé alors toutes les
places actives sont désactiv
et la place "p0" sera activé

Figure 1-44c
Squelette du GIT avec
superposition du comportement normal
et du comportement d'exception

Figure 1-44d
définition d'une
assertion type RESET

IV - RESSOURCES ALGORITHMIQUES

Comme nous l'avons vu au paragraphe III.3.2.1, 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. Cette notion est équivalente en CADOC.LD à celle de ressource algorithmique (RGA).

IV.1 - CONSTITUTION D'UNE RESSOURCE ALGORITHMIQUE

Une description de RGA 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 (cf. paragraphe III.3.2.1).

Remarque

Une RGA peut utiliser d'autres RGA, qui doivent être déclarées (voir paragraphe IV.2), la recursivité n'étant pas admise.

Exemple

La description sous forme de ressource algorithmique du bloc algorithmique donné en III.3.2.1 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
```

IV.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 ;
```

IV.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) ;
```

SECTION II : GENERATION DE PROGRAMME DE TEST FONCTIONNEL DES CIRCUITS INTEGRES COMPLEXES

Le problème du test des circuits intégrés LSI et VLSI, et des systèmes à base de ces circuits se pose à différents niveaux :

- en fin de conception, le test est effectué sur un prototype d'un circuit. Le facteur temps n'est pas le plus primordial, par contre on voudrait, si possible, tester le prototype exhaustivement.

- en fin de fabrication, le facteur "temps de test" intervient d'une manière importante dans l'élaboration d'une méthode de test afin de ne pas ralentir la chaine de production. Il faut aussi noter que le nombre restreint d'accés des circuits intégrés LSI/VLSI rend encore plus difficile leur test à ce niveau.

- en entrée chez les utilisateurs, le problème de test se pose à trois sous-niveaux :

i) à la réception des différents circuits, l'utilisateur procède à des tests paramètriques -statiques et dynamiques- et des tests logiques dans le but de vérifier les caractéristiques fournies par le fabricant.

ii) les composants, individuellement testés, sont assemblés pour former un système (ou un élément d'un système) plus complexe ; le test consiste alors à tester le système ou l'élément d'un système (test d'une carte).
iii) si le système est programmable, par exemple à base de microprocesseurs, ce troisième niveau de test consiste à vérifier que l'ensemble "matériel" et "logiciel" est conforme à l'application prévue.

Nous examinerons dans cette section essentiellement les problèmes de test en fin de conception et en fin de fabrication des circuits intégrés.

Les méthodes de génération de test seront rappelées dans le chapitre 2. Les idées de bases ont été élaborées pour les circuits de degré de complexité faible : c'est-à-dire pour les SSI (Small Scale Integration) et les MSI (Middle Scale Integration).

Grâce au développement croissant de la technologie, on peut actuellement accroître considérablement le nombre d'éléments (transistors,...) sur un même circuit.

Mais en conséquences, le test de ces circuits devient un problème qui actuellement n'est pas complètement résolu. L'approche multiniveau semble être une solution à ce problème : nous présenterons à la fin du chapitre II une solution pour les circuits intégrés complexes à contrôleur synchrone. Cette solution utilise l'exécution symbolique temporisée comme outil de base. Mais pour faire face à tous les types de circuits (control-driven et data-driven), nous proposons un enrichissement de cet outil : ceci sera l'objet du chapitre III.

Au chapitre IV, nous proposons une approche de génération de spécification de test basée sur deux techniques : l'exécution symbolique temporisée généralisée et l'intelligence artificielle. Cette proposition fait complètement abstraction du type du circuit.

CHAPITRE 1 : INTRODUCTION ET DEFINITIONS

I - DEFINITIONS

Dans ce paragraphe, nous donnerons les définitions des termes que nous utiliserons par la suite.

I.1 - DEFAILLANCE - PANNE - ERREUR

Ces trois notions peuvent être définies par les relations suivantes :

- une **défaillance** est une imperfection physique
- une **panne** est la manifestation au niveau logique d'une ou plusieurs défaillances physiques
- une **erreur** est la manifestation au niveau fonctionnel d'une ou plusieurs pannes.

Des modèles de défaillances, de pannes et d'erreurs sont faits pour représenter des classes de défaillances, de pannes ou d'erreurs. Ces modèles sont aussi appelés hypothèses (de défaillances, de pannes ou d'erreurs).

I.2 - TEST DE DISTINCTION - TEST D'IDENTIFICATION

Nous pouvons distinguer deux méthodes classiques de génération de test d'un circuit :

Le test de distinction

Cette méthode s'appuie sur des hypothèses de défaillances de pannes ou

d'erreurs fonctionnelles. Par cette méthode, on cherche à distinguer les circuits "faux" (avec un défaut) du circuit "bon" (sans défaut).

Le test d'identification

Ce test consiste à vérifier que le circuit exécute normalement toutes ses fonctions spécifiées dans son cahier de charges.

II - LA GENERATION DES VECTEURS DE TEST

Les vecteurs de test d'un circuit peuvent être générés aléatoirement pendant le test effectif du circuit. Cette méthode, dite **test aléatoire** (cf paragraphe I du chapitre II), n'est basée sur aucune étude préalable des données de test.

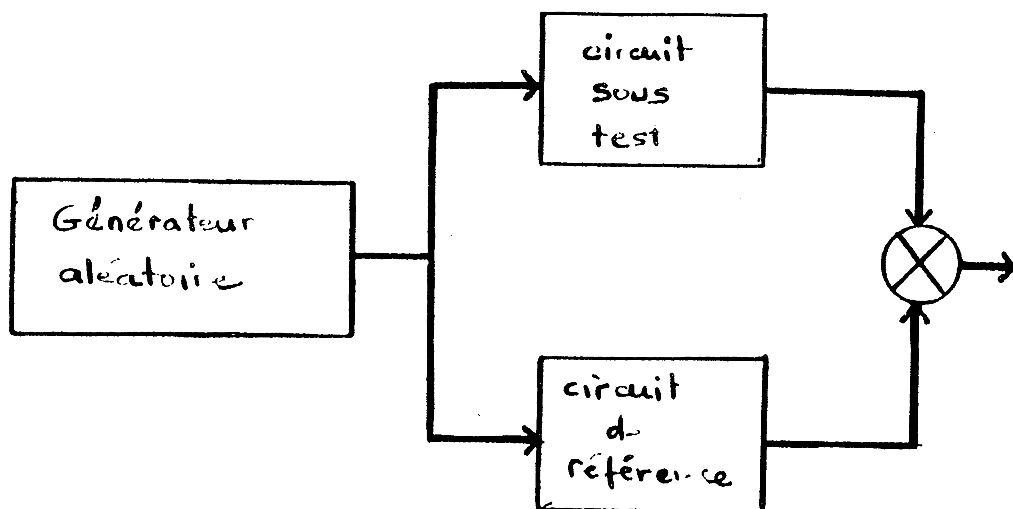


Figure 2-1.

Par contre, il existe d'autres méthodes qui consistent à déterminer les vecteurs de test avant le test effectif. Dans cette classe, dite **test avec vecteurs prédéterminés** (cf paragraphe II du chapitre II), on peut énumérer trois méthodes :

- Recherche aléatoire

Cette méthode consiste à chercher aléatoirement des vecteurs de test en utilisant un test par distinction. L'outil utilisé est alors un simulateur de fautes. Les entrées du simulateur sont générées aléatoirement. Les sorties de simulation du circuit "bon" et du circuit "faux" sont comparées. Une entrée sera un vecteur de test pour la panne considérée si leurs sorties sont différentes.

Les défauts, pannes ou erreurs injectés dans le modèle du circuit peuvent être extraits d'une liste. A la fin du processus, on peut définir le taux de couverture des défauts, pannes ou erreurs.

- Recherche déterministe

On peut citer deux méthodes déterministes qui se basent sur une description du circuit au niveau structurel ou fonctionnel :

i) par distinction, en utilisant des modèles de défauts/pannes/erreurs

ii) par identification.

- Recherche mixte

Cette méthode consiste à appliquer une recherche aléatoire des vecteurs de pannes. S'il existe des pannes non détectées alors on appliquera une recherche déterministe.

III - L'APPLICATION DU TEST

Le problème de l'application du test est de trouver une politique de test prenant en compte :

- l'objectif de l'application du test,
- les possibilités de l'équipement de test,

- la structure du circuit.

Les objectifs de l'application du test sont deux types :
- vérifier si le circuit fonctionne correctement ou pas : ce test est appelé test de détection (test **'go/no go'**)
- faire un test 'go/no go' mais si de plus on a détecté que le circuit fonctionne incorrectement, localiser la ou les parties défaillantes du circuit : **diagnostics**.

Les problèmes de génération du test se posent sur deux notions :
- la contrôlabilité du circuit,
- l'observabilité des résultats de test.

Ces deux points caractérisent la difficulté du test du circuit. L'évaluation de ces difficultés appartient au domaine de la testabilité des circuits. Des outils d'évaluation de testabilité et de spécification de programmes de test ont été développés tel CATA (Laboratoire circuits et Systèmes) [ROB 83],ces travaux sont fondés, explicitement ou implicitement sur la notion d'écoulement d'informations à travers les blocs du circuit.

IV - L'ANALYSE DES RESULTATS

Les résultats obtenus pendant ou après le test sont analysés selon l'objectif fixé pour le test. Il existe deux méthodes de base :
- comparaison des séquences de sortie avec celles préalablement établies par simulation du circuit de référence.
- observation d'une propriété des séquences de sorties en faisant un test compact ou par analyse de signature.

Une de ces deux méthodes est suffisante pour un test 'go/no go' mais pour un test de diagnostic une analyse plus poussée est faite au niveau des séquences de sorties : déterminer à partir des différentes séquences de sorties les blocs fautifs. Un outil de type "système expert" peut faire cette analyse [RAR 84a]. La localisation des éléments en défauts sera plus précise en utilisant des équipements de test permettant d'observer directement aux points internes du circuit, tels ceux bâtis autour d'un microscope électronique à balayage (MEB) [LAU 84].

CHAPITRE II - LES METHODES DE GENERATION DE TEST

Pendant ces dix dernières années, l'évolution rapide des technologies dans la fabrication des circuits intégrés a aussi entraîné les autres domaines techniques et scientifiques utilisés dans la phase de conception de ces circuits. De nouvelles méthodologies de conception ont été élaborées (conception structurée et descendante) [SAU 81]. Le domaine du test logique et paramètrique ne fait pas exception à cette évolution. Dans un premier temps, nous voulons présenter l'état de l'art dans le domaine de l'élaboration des tests. Dans un second temps, nous présenterons le problème de testabilité des circuits intégrés.

I - LE TEST ALEATOIRE

Dans cette technique, les vecteurs de test sont appliqués aléatoirement au circuit sous test et à un circuit de référence. Les sorties de deux circuits sont analysées par comparaison.

En ce qui concerne le circuit de référence, il peut être un circuit réel dit "réputé bon" (le problème est alors de trouver ce circuit) ou une description du circuit qui sera simulée.

I.1 - TEST ALEATOIRE DES CIRCUITS COMBINATOIRES [DAV 76]

- Soit Q_D la probabilité de déclarer "bon" un circuit défectueux (qualité de détection),
- soit P_{BF} la probabilité qu'un vecteur d'entrée ne manifeste pas une panne donnée.

Alors la longueur L de la séquence vérifie $L > \frac{\text{Log } Q_D}{\text{Log } P_{BF}}$

Pour un circuit combinatoire de N entrées primaires, et en considérant la panne la plus difficile à détecter (c'est-à-dire que cette panne est détectée par le plus petit nombre de combinaisons d'entrée), la longueur de la séquence de test est : $L = \frac{2^N}{\sigma} \text{ Log } \frac{1}{Q_0}$

Le problème de latence d'erreur d'une panne c'est-à-dire le nombre de vecteurs d'entrée qui sont appliqués au circuit en présence de cette panne avant qu'elle ne se manifeste à la sortie, est étudié dans [SHE 77].

I.2 - TEST ALEATOIRE DES CIRCUITS SEQUENTIELS

Le modèle utilisé pour décrire le fonctionnement de l'automate est la chaine de Markov de premier ordre, telle que les probabilités des entrées ne changent pas au cours du temps.

- Soit M la matrice de transition, qui est une matrice carré, telle que m_{ij} représente la probabilité de passer de l'état i à l'état j.
- Soit S^n le vecteur de probabilité d'état où le $j^{ième}$ élément est la probabilité que l'automate se trouve à l'état j après n pas.

Alors : $S^n = M.S^{n-1}$ et par récurrence $S^n = M^n S^0$

Il a été montré que tout circuit fortement connexe a un vecteur de probabilité d'état stationnaire unique, c'est-à-dire U = M.U. [SHE 74].
A partir de ce système d'équation, on pourra définir :

- le rapport de temps pendant lequel l'automate est dans un état donné,
- les mesures des portions de temps pendant lesquels on peut s'attendre à observer les sorties.

Comme pour les circuits combinatoires, le problème de latence d'erreur existe et a été l'objet de [SHE 76].

.3 - CONCLUSION

Il est évident que le test aléatoire utilise un système matériel simple, mais une longueur prohibitive du test. Le test aléatoire n'est pas une bonne alternative pour un haut degré de confiance, par exemple pour un test en fin de conception.

II - TEST AVEC VECTEURS PREDETERMINES

Dans cette approche, on veut déterminer à l'avance les vecteurs de test du circuit. Par ailleurs, on ne dispose que d'une ou plusieurs spécifications du circuit servant de référence. Le problème est alors de déterminer les vecteurs de test du circuit à partir de ses spécifications selon l'objectif de test choisi (test de distinction -test d'identification-).

Rappelons brièvement les notions de descriptions d'un circuit, utilisées dans le domaine du test:

- **description structurelle** : le circuit est décrit en termes d'assemblage de blocs élémentaires qui sont définis suivant l'état de la conception du circuit et la complexité d'intégration du circuit. Par exemple, un circuit SSI ou MSI est décrit en termes de portes logiques interconnectées. Par contre les circuits complexes LSI sont décrits en termes de portes logiques et de primitives plus complexes (registres, UAL, multiplexeurs,...) et les VLSI comme une composition de blocs fonctionnels plus ou moins complexes (cf langage CADOC/LD).
- **description fonctionnelle** : le but de la description est de mettre en évidence les fonctions réalisées par le circuit. Par exemple, les

fonctions des circuits SSI et MSI ont été présentées sous forme de tableau de vérité ou d'états. Les fonctions des LSI sont décrites par des langages de type RTL. Les VLSI sont décrits en utilisant des langages multiniveaux qui permettent de décrire aussi bien les fonctions du circuit que son architecture.

- **description algorithmique** : la description du circuit consiste à donner l'algorithme des fonctions exécutées par le circuit. La structure interne du circuit n'est pas prise en compte. Cette description est plutôt utilisée pour la spécification des circuits LSI/VLSI au tout premier stade de leur conception.

Complexité	Description structurelle	Description fonctionnelle	Description Algorithmique
SSI MSI	- portes logiques	- table de vérité - tableau d'états	
LSI	- portes logiques - registres, mémoires, UAL, multiplexeurs	- langage RTL	- langage de programmation
VLSI	- langage multiniveau (structure / fonctionnelle)		- langage de programmation

Figure 2-2

Nous allons brièvement présenter les méthodes existantes pour l'élaboration de tests des circuits logiques suivant l'objectif de test (test de distinction, d'identification). Pour chaque méthode présentée, nous préciserons les points suivants :

- type et niveau de description utilisée,
- l'inadéquation de la méthode pour les circuits complexes.

II.1 - TEST DE DISTINCTION

II.1.1 Méthode de sensibilisation de chemin : D-Algorithme

La description utilisée est une description structurelle en termes de portes logiques de base. L'objectif de la méthode étant de générer des vecteurs de test ou de séquences de test permettant de distinguer un circuit possédant une panne du circuit sans panne, les hypothèses de pannes utilisées sont de deux classes : collage à 0 ou collage à 1 d'une connexion quelconque.

La méthode de sensibilisation de chemin comporte trois étapes :

- manifestation de la panne,
- propagation de la panne vers une sortie primaire,
- consistance : recherche des vecteurs d'entrée.

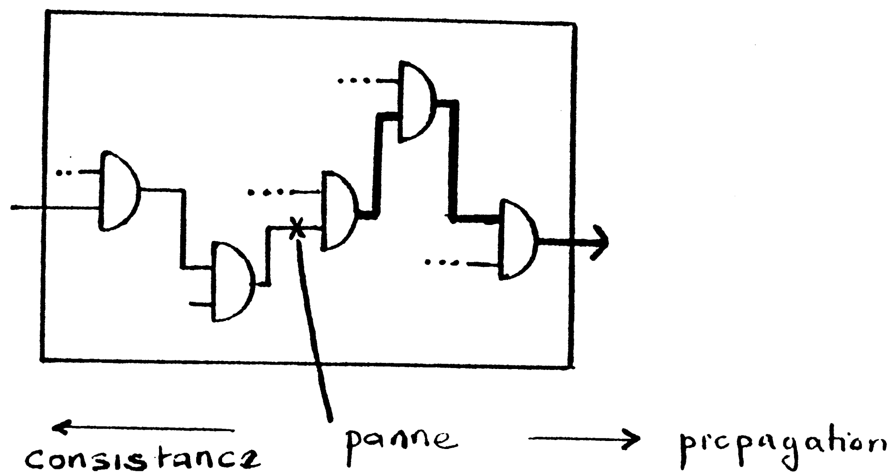


Figure 2-3.

Il existe des règles de propagation d'erreurs telles :

$$\tilde{D} + 0 = \tilde{D} \; ; \; \tilde{D} + 1 = 1 \quad \text{où } \tilde{D} = D \text{ ou } \bar{D}$$

$$\tilde{D} \cdot 0 = 0 \; ; \; \tilde{D} \cdot 1 = \tilde{D}$$

et $D = 0 \rightarrow 1$ avec 0 valeur juste
1 valeur fausse

Ces règles sont représentées par un D-cube associé à chaque type de portes. La connaissance de telles règles simplifie la propagation de la panne en utilisant un simulateur logique, et elle constitue un des avantages de cette méthode.

Le D-Algorithme, induit de cette méthode, comporte deux phases :

- phase de synthèse qui consiste à déterminer un ou plusieurs vecteurs de test détectant une panne donnée suivant la méthode précédente,
- phase d'analyse qui consiste à determiner toutes les pannes couvertes par ces vecteurs.

En ce qui concerne les circuits séquentiels, la méthode peut être appliquée en les transformant en circuits combinatoires itératifs.

L'application du D-Algorithme aux circuits complexes n'est pas une solution intéressante. En effet, décrire le circuit entier en termes de portes logiques nécessite que la conception du circuit soit au niveau portes logiques. Or la complexité de ces circuits en portes logiques est de l'ordre de une ou plusieurs dizaines de milliers de portes : la description structurelle au niveau logique est donc coûteuse sinon impossible. D'autre part, vue la forte densité d'intégration des circuits actuels (VLSI), les hypothèses de collages à 0 et à 1 utilisées pour cette méthode ne sont plus adéquates [NIC 80]. Des modèles de fautes utilisés pour le test des mémoires ont été proposé par cet auteur : intéraction avec le voisinage, etc....

II.1.2 <u>Différence booléenne</u> [SEL 68]

Le circuit combinatoire est décrit par sa fonction booléenne correspondante qui peut être qualifiée de description fonctionnelle. Les hypothèses de pannes utilisées sont les collages à 0 ou à 1. Cette méthode ressemble à la méthode de sensibilisation de chemin, dans la mesure où un chemin sensible à partir d'une connexion xi supposée collée à 0 ou à 1 est exprimée par la différence booléenne suivante :

$$\frac{dF}{dh_i} = F(x_1,\ldots,x_{i-1},0,\ldots x_n) + F(x_1,\ldots,x_{i-1},1,\ldots,x_n) = 1$$

L'ensemble des tests pour une panne en xi sera :

$$x_i \frac{dF}{dxi} \text{ pour un collage à 0}$$

$$x_i \frac{dF}{dxi} \text{ pour un collage à 1}$$

Nous pouvons faire les mêmes critiques que celles pour le D-Algorithme. En plus, nous devons pour chaque connexion interne hi trouver l'expression booléenne F dépendante des entrées et de hi. Cette contrainte est trop pénible pour les circuits complexes.

Exemple :

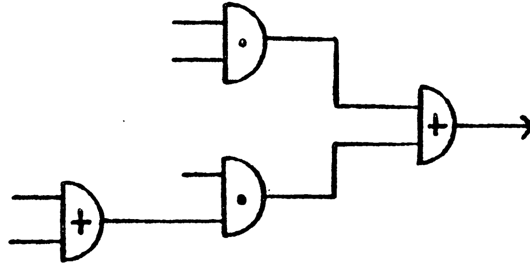


Figure 2-4.

$$F = ab + ah \quad \begin{cases} h\,\dfrac{dF}{dh} = (c+d)\bar{a} = \bar{a}c + \bar{a}d \\[2ex] \bar{h}\,\dfrac{dF}{dh} = \bar{c}\,\bar{d}\,\bar{a} \end{cases}$$

L'ensemble des vecteurs est déduit à partir de ces 2 expressions.

II.1.3 Méthode de Poage [SAU 72]

Cette méthode a été élaborée pour le test des circuits séquentiels. La description utilisée est le tableau d'états. L'objectif de la méthode est de distinguer une machine "juste" d'un ensemble de machines "fausses". Les hypothèses d'erreurs sont du type : un état est activé au lieu d'un autre.

On considère une machine de référence qui est la machine "Juste" représentée par son tableau d'états T_0 et son état initial S_0. Pour chaque erreur considérée, on définit le tableau d'état correspondant avec leur état initial respectif.
Soit M_0 la machine de référence avec (T_0, s_0),

Soit M_i une machine fausse avec (T_i,s_i), (T_i: tableaux d'états de M_i,S_i état initial).

La méthode consiste à suivre les étapes suivantes :

- faire le produit cartésien des tableaux T_0xT_i avec comme état initial (s_0,s_i) ; l'<u>état puit</u> de T_0xT_i est celui dont les sorties sont différentes (ceci permet de distinguer T_0 de T_i)
- pour rechercher une séquence minimale permettant de distinguer T_0 des T_i, faire le produit cartésien des tableaux (T_0xT_i). Soit τ = le tableau d'état correspondant, celui-ci n'est pas construit exhaustivement ; la construction s'arrête dès qu'on peut distinguer T_0 des T_i. La recherche de cette séquence minimale est faite en 2 temps :
  - recherche des ensembles minimaux d'états de τ telle qu'une séquence passant par un tel ensemble d'états permet de distinguer T_0 des T_i
  - recherche d'une telle séquence.

Ces deux étapes sont faites à partir d'une forme simplifiée ([SAU 72]) en remplaçant les états puits par un symbole *. Les ensembles minimaux d'états Em de τ sont obtenus par couverture des symboles *

En traduisant le tableau τ par son graphe d'états, la recherche de séquence de test consiste à chercher un chemin de ce graphe issu de l'état initial et passant par les états de l'ensemble Em.

[SAU 72] propose une recherche plus rapide que celle proposée par Poage.

En effet cette méthode initiale de Poage est impraticable pour des circuits séquentiels plus de 10 états.

La méthode proposée par [SAU 72] part directement des tableaux T_0xT_i, ce qui permet de réduire le temps d'élaboration du test. Par contre, la séquence trouvée n'est pas minimale. L'algorithme de cette dernière méthode est :

1) construction des tableaux T_0xT_i comme dans la méthode initiale.

2) construction du tableau τ avec la règle suivante :
   étant donné une ligne du tableau τ, l'état suivant qui sera considéré est l'état avec le nombre maximal de symboles *
3) l'algorithme s'arrêtera quand l'état formé uniquement '*' est atteint

Mais cet algorithme ne peut donner une solution que si T est fortement connexe et que les T_i ont le même nombre d'états que T .

Cette méthode apporte une amélioration du temps d'élaboration du test pour les circuits séquentiels simple (quelques dizaines d'états et quelques entrées). Il est évident que cette méthode ne peut être envisagée pour les circuits complexes LSI/VLSI actuels, par exemple pour le test de la partie contrôle d'un microprocesseur.

II.1.4 <u>Extensions du D-Algorithme</u>

Cette méthode a été proposée dans [LEV 81] et consiste à une généralisation de la notion de sensibilisation de chemin. Le circuit est décrit dans un langage de type RTL non procédural. Avec ce type de langage, on représente les connexions physiques et les points de mémorisation par des variables. La structure est implicite et le contrôle des primitives fonctionnelles (registre, multiplexeurs, ...) est décrit en utilisant des instructions de type IF.. THEN..ELSE.

Les hypothèses de pannes utilisées sont de deux types :
- collage à 0 ou à 1 d'une variable ou d'une connexion,
- erreur de séquencement, étant donné le caractère non procédural du langage.

La méthode est une généralisation du D-Algorithme. La base de cette méthode est la notion de "switching algebraic expressions".

Soit $U = \{0,1,D,\bar{D}\}$, et $S_i \in P(U)$ (ensemble de tous les sous-ensembles de U)

l'expression x^{Si} possède la signification suivante : "la valeur de x est un élément de Si".

Plusieurs règles de base sont établies, on peut en citer :

$$X^{Si} . X^{Sj} = X^{Si\ Sj} \qquad (1)$$
$$X^{Si} + X^{Sj} = X^{Si\ Sj} \qquad (2)$$
$$X^{Si} + X^{Si} . y = X^{Si} + y \qquad (3)$$
$$X^{Si} + S^{Si} y = X^{Si} \qquad (4)$$

La fonction booléenne "ET", par exemple, produira les D-équations suivantes :

$$C^{0} = (ab)^{0} = a^{0} + b^{0} + a^{D}b^{\overline{D}} + a^{\overline{D}}b^{D} \qquad (5)$$
$$C^{1} = (ab)^{1} = a^{1}.b^{1} \qquad (6)$$
$$C^{D} = (ab)^{D} = a^{D}b^{1} + a^{1}b^{D} + a^{D}b^{D} \qquad (7)$$
$$C^{\overline{D}} = (ab)^{\overline{D}} = a^{\overline{D}}b^{1} + a^{1}b^{\overline{D}} + a^{\overline{D}}b^{\overline{D}} \qquad (8)$$

Dans ces équations, on peut retrouver les D-cubes utilisés pour le D-Algorithme. Par exemple, de (5),(6),(7) et (8) on déduit : D.1 = D ; D.0 = 0 $D.\overline{D} = 0$.

Pour les primitives fonctionnelles de type registre, on utilisera l'équation d'état. Par exemple, pour la bascule Jk, $q^{+} = J\overline{q} + Kq$ d'où :

$$q^{+D} = (J\overline{q})^{1}\ (Kq)^{D} + (J\overline{q})^{D}(Kq)^{1} + (J\overline{q})^{D}(Kq)^{D}$$

Ces D-équations donnent les conditions de propagation des valeurs à travers les registres et seront donc utilisées pour la génération de tests.

Pour les primitives plus complexes tels compteurs, registre à décalage, décodeur..., les solutions proposées sont l'utilisation de tables ou des

algorithmes complexes [BRE 80].

Par contre, les instructions complexes telles que IF..THEN..ELSE, CASE..OF,.. sont facilement transformables en "switching algebraic expressions"(SAE).

Exemple :

IF A THEN Z ← B ELSE Z ← C a comme SAE : Z ← BA + CA

et $Z^D = (ZA)^D.(CA) + (BA)\ (CA)^D.(CA)^D$

oü A,B,C,Z sont des "switching algebraic expressions".

Le test de séquencement consiste à activer tous les chemins possibles, et se fera en quatres étapes :

- injection d'une panne à un site,
- détermination d'un objectif (variable de sortie à atteindre)
- propagation de la panne en faisant des choix,
- justification des décisions (consistance,.

L'algorithme s'arrête quand le but est atteint.

L'automatisation de cette approche semble être complexe ; les D-Equations pour des blocs fonctionnels plus complexes que les registres ne peuvent être données que par l'utilisateur.

II.1.5 S-Algorithme [LIN 84]

Cette méthode est basée aussi sur une description fonctionnelle du circuit au niveau RTL. Elle a pour but de générer des séquences de tests permettant de détecter des erreurs fonctionnelles. Elle utilise comme outil l'exécution symbolique.

En rappelant par la figure 1 la forme générale d'une instruction d'un langage RTL, les **hypothèses d'erreurs fonctionnelles** sont de 9 types :

(1) - k/k' : erreur d'étiquette
(2) - t/t' : erreur de temporisation
(3) - c/c' : erreur sur la condition
(4) - (R)/(R') : erreur de mémorisation
(5) - ←/←' : erreur de transfert
(6) - R/R' : erreur d'adressage de registre
(7) - (f)/(f') : erreur d'exécution d'un opérateur
(8) - →n/→n' : erreur de saut
(9) - f/f' : erreur de décodage de fonctions

Ces erreurs sont la manifestation au niveau fonctionnel de une ou plusieurs pannes (collage, court circuit...). [LIN 84] a établi quelques théorèmes qui permettent de réduire le nombre d'erreurs possibles. Par exemple, les erreurs de types k/k' sont couvertes par celles de types →n/→n'.

La syntaxe généralisée d'un langage RTL est :

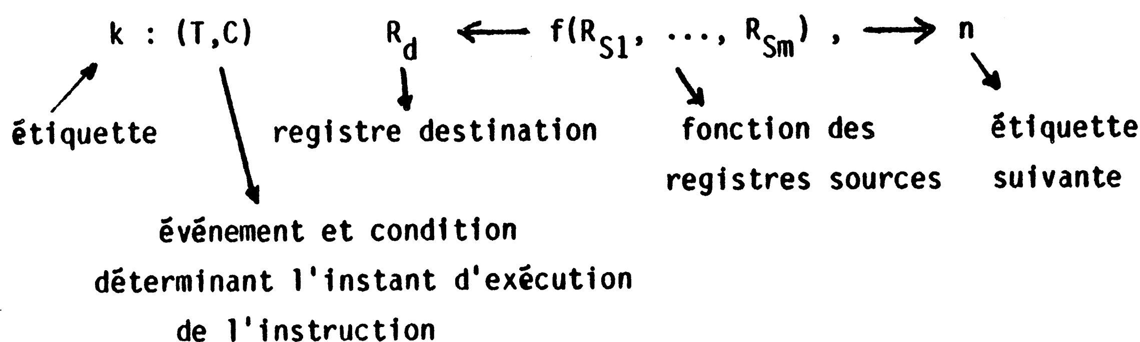


Figure 2-5.

L'idée générale de la méthode a été donnée dans [SU 82] qui consiste à

déduire les tests du circuit par comparaison des résultats d'exécution symbolique du circuit de référence ("bon") et ceux du circuit "faux".
La méthode proposée consiste en 3 parties :

1) <u>Pré-processus</u>

(1.a) - déduire toutes les fonctions possibles du circuit à partir de sa description,

(1.b) - ordonnancement des fonctions suivant le principe "start-small".

2) <u>S-Algorithme</u>

(2.a) - pour une fonction donnée, faire l'exécution symbolique de la fonction **sans erreur** et l'exécution symbolique de la fonction **avec erreur** (injection d'une erreur prise dans la liste d'erreurs possible dans cette fonctio: .

(2.b) - déduire un ensemble de tests permettant de détecter cette erreur par comparaison des résultats de la fonction symbolique de la fonction "correcte" et ceux de la fonction "incorrecte".

3) <u>Post-processus</u>

(3.a) - recherche par simulation des autres erreurs de la liste pouvant être couvertes par les tests trouvés précédemment.

- Tant qu'il y a une erreur non détectée pour la fonction choisie alors revenir en (2.a) avec la même fonction. (injection des erreurs restantes).

- Tant que la liste ordonnée de fonctions n'est pas terminée, alors revenir en (2.a) en considérant la fonction suivante.

[LIN 84] remarque que la complexité du S-Algorithme dépend du nombre d'instructions et de la complexité de l'outil d'exécution symbolique.

Les inconvénients majeurs de cette méthode, à notre avis, se trouvent à 2 niveaux :

- au niveau de la modélisation du circuit : le circuit doit être décrit à un seul niveau (RTL) ; la structure du circuit est entièrement inconnue,
- au niveau hypothèses d'erreurs : ces erreurs pouvant être la manifestation de plusieurs défaillances physiques, et surtout on n'est pas sûr que ces hypothèses couvrent toutes les défaillances du circuit.

En conclusion, cette proposition nous semble irréaliste. En effet, le nombre de circuits "faux" est très important et ,par conséquent, l'exécution symbolique de tous ces circuits faux est impossible sinon elle nécessiterait des temps de calcul prohibitifs. Par ailleurs, la couverture des défaillances par des hypothèses d'erreurs fonctionnelles est douteuse.

II.1.6 Génération à partir du graphe de transformation d'états

La solution qui a été proposée dans [LAI 83] est une solution basée sur l'utilisation de modèles de défauts multiniveaux. La génération de tests se fait à partir d'une description de haut-niveau. Le langage utilisé est un graphe appelé STG (State Transformation Graph). C'est un graphe orienté dont les noeuds représentent des opérateurs de transformations de données et les arcs représentent les chemins de données ou de contrôle.

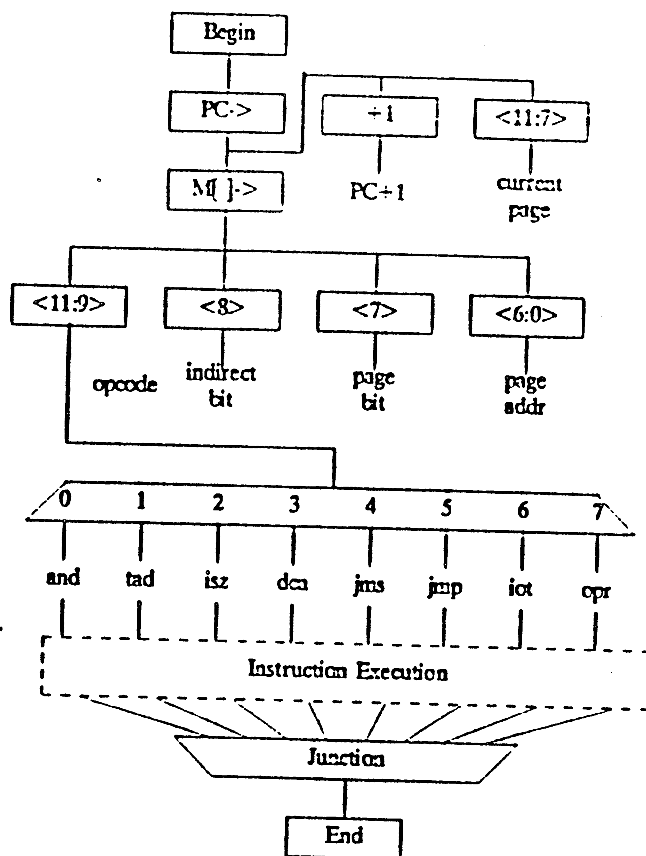


Figure 2-6.

STG du PDP-8

PC → : opération de lecture à partir du regis e PC

PC ← : opération d'écriture dans le registre PC.

Le STG permet de décrire des circuits à différents niveaux de détails allant des primitives de complexité arbitraire définies par l'utilisateur jusqu'aux portes logiques. En d'autres termes, le STG permet de spécifier à la fois les fonctions du circuit et sa structure.

Cette méthode de test semble être une voie intéressante mais la description du système selon [LAI 83] ne nous permet pas d'évaluer correctement son efficacité. La définition des modèles d'erreurs n'est pas

claire, surtout la nécessité de deux niveaux de modèles d'erreur n'est pas très bien expliquée.

D'une manière simplifiée, le noyau du système est "l'analyseur fonctionnel" (cf figure 2-7a). Celui-ci, génère un ensemble de tests paramétrés selon le modèle d'erreur fonctionnelle (c'est-à-dire au niveau graphe) choisi par l'utilisateur. La génération de test paramétré consiste en quatre étapes (cf. figure 2.7b) :

- introduction des paramètres de test dans le STG,
- propagation en arrière pour trouver les conditions nécessaires pour le test des primitives considérées,
- propagation en avant pour l'observation des résultats de tests,
- justification, c'est-à-dire, initialisation des chemins à valeur non définie.

Le deuxième élément du système est le "synthétiseur de tests". Il fait la substitution des paramètres formels des tests paramétrés (générés précédemment) par des vecteurs de tests définis au niveau inférieur (niveau primitive : collage à 0, à 1, court-circuit...) et extraits d'une base de données de tests.

Le troisième élément est le "synthétiseur de programme de test". A partir des vecteurs de test trouvés précédemment, il construit un segment de programme qui sera prêt à ê^re exécuté pour le test effectif du circuit.

En conclusion, l'approche basée à la fois sur des descriptions fonctionnelle et structurelle est très intéressante dans la mesure où l'on peut considérer des pannes structurelles et des erreurs fonctionnelles. Elle pourra servir d'idée de base pour le test des circuits complexes LSI/VLSI. Mais l'inconvénient est qu'un bloc matériel est représenté plusieurs fois dans le graphe, et que le modèle traité est l'erreur unique. Ceci pose des problèmes tels problèmes de masquages ... .

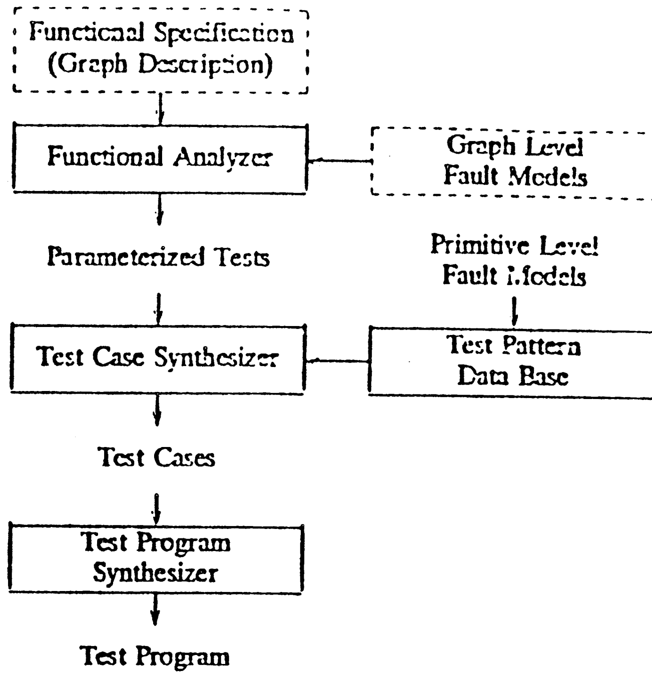


Figure 2-7a.

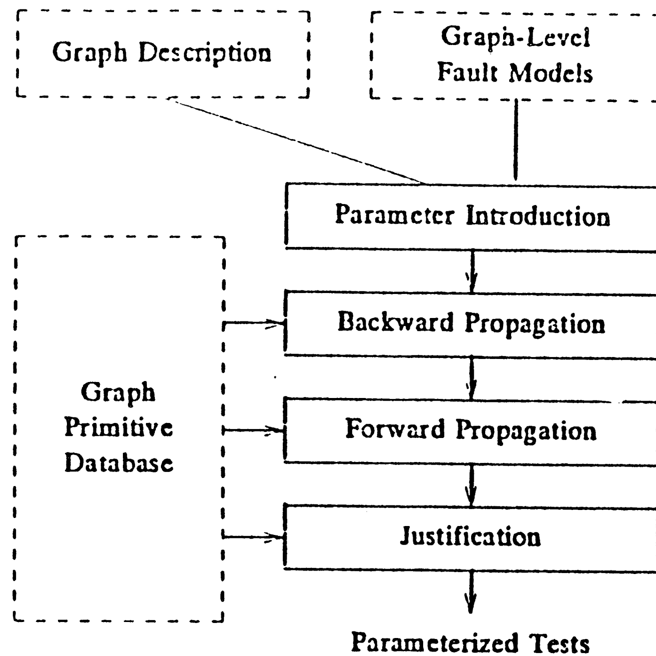


Figure 2-7b.

II.1.7 Technique développée par GTE Lab

[SON 82] propose une technique développée au laboratiore GTE. Cette technique consiste à générer des vecteurs de test à partir d'une table fonctionnelle et sur des hypothèses d'erreurs fonctionnelles prédéfinies ou définies par l'utilisateur.

La table fonctionnelle peut être déduite d'une description de haut niveau, par exemple une description algorithmique. C'est une table de couples (conditions, actions). Une traduction de la description de la figure 2-8a est donnée en figure 2-8b.

```
IF A THEN
     DO P1
     DO WHILE B;
     IF C THEN P2;
          ELSE P3;
     END;
     IF D THEN P4;
ELSE
     DO P5;
     IF E THEN P6
          ELSE P7;
     REPEAT P8 UNTIL F;
     END;
CASE G:
G1 : P9;
G2 : P10;
```

Figure 2-8a.

```
(A) : P1;
     (B)*:(c) P2;
          (¬c) P3;
     (D) : P4;
(¬A) : P5;
     (E) : P6;
     (¬E) : P7;
     (F)- : P8;
(G=G1) : P9;
(G=G2) : P10;
```

Figure 2-8b.

Les hypothèses d'erreurs fonctionnelles sont de 4 types :

- erreur de donnée,
- erreurs de contrôle,
- erreur d'opération,
- erreurs définies par les utilisateurs.

L'idée de base de la méthode est la sensibilisation de chemin avec deux étapes : propagation (forward driving) et justification (backward driving). Cette technique est intéressante dans la mesure où le test peut être étudié dès lors des premières phases de la conception du circuit. Mais le problème qui se pose est la précision des hypothèses d'erreurs du moins celles qui sont définies par l'utilisateur.

II.1.8 Conclusion sur le test de distinction

Nous avons présenté quelques méthodes de générations de test qui ont pour objectif de distinguer le circuit "juste" des circuits "faux", autrement dit de faire un test de distinction.

La méthode la plus classique est le D-Algorithme qui est basée sur une description structurelle au niveau logique, d'une part, et sur des hypothèses de pannes au niveau des connexions (structurelles). La méthode par différence booléenne utilisant l'expression booléenne respecte la même philosophie que celle du D-Algorithme. Ces méthodes sont bien adaptées pour les circuits de faible complexité. Mais après l'avènement des circuits complexes, elles ne sont pas praticables. De nouvelles méthodes se basent sur une description structurelle à base d'éléments plus complexes que les portes logiques, à savoir les multiplexeurs, les registres...; une généralisation du D-Algorithme a été proposée qui consiste à déterminer des D-cubes pour ces éléments plus complexes.

Ces méthodes deviennent à leur tour inadaptées au test des circuits LSI/VLSI. Alors les dernières propositions ont pour objectif de générer des tests de distinction mais à partir de la description fonctionnelle du circuit (description plus compacte). Les hypothèses de défauts utilisées sont de types fonctionnelles déduites à partir du langage de description fonctionnelle sans faire référence à la structure interne du circuit. Nous avons particulièrement présenté le S-Algorithme basé sur une description au niveau RTL. Un des problèmes de cette méthode est la précision dans la couverture des défauts physiques.

L'approche basée sur le STG (State Transformation Graph) considère plutôt de niveau de modèles de defauts : des erreurs fonctionnelles qui sont plutôt au niveau du graphe, et des pannes structurelles qui sont considérées au niveau de chaque primitive utilisée. Cette approche présente un problème au niveau de la précision des hypothèses de pannes structurelles ou d'erreurs fonctionnelles.

II.2 - TEST D'IDENTIFICATION

Contrairement au test de distinction, l'objectif du test d'identification est de vérifier que le fonctionnement du circuit sous test satisfait à sa spécification. La base du raisonnement est donc de couvrir tous les modes de fonctionnement du circuit indépendamment d'hypothèses de défaillances de pannes ou d'erreurs.

II.2.1 Test exhaustif des circuits combinatoires

La description servant de base est le tableau de vérité des circuits combinatoires. L'objectif du test est d'activer tous les états possibles du circuit.

Soient m le nombre d'entrées :

La longueur totale du test est $N = 2^m$.

Ce test était possible pour les circuits de faible complexité (SSI et certains MSI). Mais pour un circuit plus complexe LSI par exemple, le temps de test nécessaire pour obtenir même un taux de couverture de 50 % est bien prohibitif (cf. figure 2-9). Cette approche a été généralisée par Mc Cluskey [CLU 84] qui a défini le test pseudo-exhaustif qui consiste à partitionner le circuit en blocs à une seule sortie. Chaque bloc déterminé par des "cônes de dépendance" est testé exhaustivement.

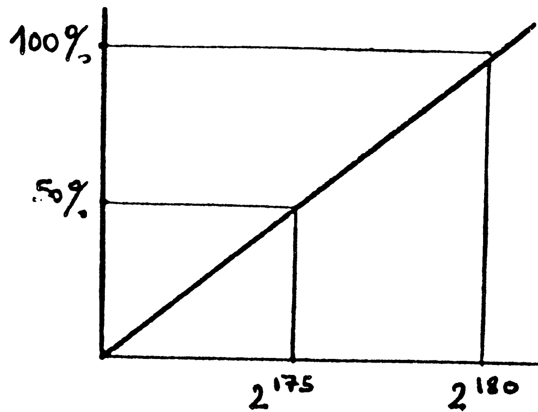


Figure 2-9.

II.2.2 - **Méthode d'identification d'automates** [KOH 70]

Le fonctionnement de l'automate est décrit par son tableau d'états. La méthode pose trois hypothèses :

- le tableau d'état est réduit,
- le graphe d'état du circuit sans la présence de fautes est fortement connexe,
- le nombre d'états n'augmente pas en présence d'une panne.

La procédure d'identification est la suivante :

(1) initialisation de l'automate dans un état connu par une séquence de synchronisation,

(2) vérification du nombre d'états (identification des états),

(3) vérfication de toutes les transitions entre états au moyen d'une séquence de distinction.

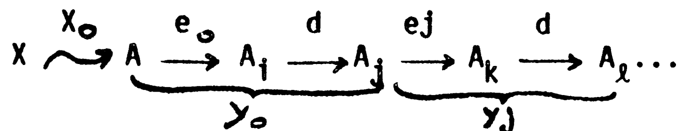
Une séquence de synchronisation est une séquence d'entrée qui met l'automate dans un état déterminé quelque soit l'état initial (inconnu).

Une séquence d'entrée X est une séquence de distinction si la séquence

de sortie produite caractérise de façon unique l'état initial avant l'application de X et l'état final.

La séquence de test d'une manière générale comporte :

- une séquence de synchronisation X ,
- des séquences Xi de type :
    * activation d'une transition : e_i
    * application de la séquence de distinction : d_i
- des séquences permettant de mettre l'automate dans un état donné pour permettre le test des transitions restantes.



Cette séquence peut être réduite en utilisant des sous séquences qui permettent en même temps d'activer des transitions et de distinguer les états.

Cette méthode est utilisable pour des automates de quelques états et une ou deux entrées. Ce qui n'est pas le cas pour les circuits complexes actuels tels les microprocesseurs.

II.2.3 - Graphe abstrait d'exécution [SAU 80]

Cette méthode de test a été proposée pour le test des microprocesseurs. Pour leurs utilisateurs, leur structure interne est inconnue. Pour les concepteurs de programme de test en fin de conception la connaissance de la structure n'est pas du tout un avantage car elle est inexploitable compte tenu de la complexité du circuit et des erreurs de conception possibles.

L'objectif de la méthode est de vérifier que le microprocesseur exécute bien toutes ses instructions. L'information de base de la méthode est le jeu

d'instructions du microprocesseur. Mais la génération effective du programme de test se fait à partir d'un format intermédiaire qui est le "graphe abstrait d'exécution" (GAE) associé à chaque instruction.

<u>Exemple</u> : le graphe abstrait d'exécution de l'instruction ADDA n,x" du MC6800 est :

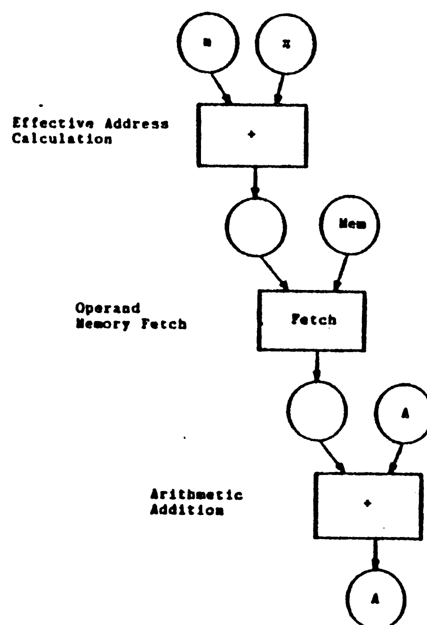


Figure 2-10.

La procédure de génération consiste en 5 étapes :

1) décrire les "graphes abstraits d'exécution" à partir de la description fonctionnelle du microprocesseur.
2) Vérification de conformité des GAE avec la spécification.
3) Partitionnement des GAE en groupe tels que les GAE du même groupe

ont la même structure,

4) Recherche d'un ordre de test des GAEs selon des stratégies type "start-small" ou "start-big",

5) Définition des opérandes de test pour chaque GAE [SAU 81].

Cette méthode permet la génération de programme de test à partir d'une description fonctionnelle (boîte-noire) dont le but est d'activer exhaustivement toutes les fonctions du circuit. Cette approche est intéressante malheureusement elle ne peut être appliquée qu'à des circuits complexes de type microprocesseurs. Elle a donné lieu, par la suite, au système de génération automatique de programme de test pour les microprocesseurs GAPT [BEL 82][BEL83]

II.2.4 - <u>Technique développée par Texas Instruments</u> [JOH 79]

Texas Instruments a développé une technique de génération de tests à partir d'une description décrite dans un langage algorithmique (de type ALGOL). Leur but est de trouver des tests indépendants de la réalisation finale du circuit, donc de préparer le test très tôt dans la phase de conception.

L'objectif de test est de vérifier que le circuit exécute bien l'algorithme spécifié. Mais étant donné qu'une instruction de la description fait appel à plusieurs objets (variables, opérateurs), le test ne peut être généré que pour un ensemble d'objets. Par exemple, l'activation de l'introduction A : = B+C permet àla fois d'activer A,B,C et l'opérateur "addition".

La procédure de génération de test comprend 4 étapes :

1) choix des objets à activer pour assurer une bonne couverture de test,

2) recherche d'un chemin à partir des entrées primaires jusqu'à l'objet-cible,

3) recherche d'un chemin à partir de l'objet-cible jusqu'aux sorties primaires,
4) résolution des conflits de valeurs le long des chemins et détermination des données de test.

Une forme intermédiaire a été définie pour faciliter l'implémentation de cette approche. Texas Instruments a utilisé un modèle graphique le "**diagraphe**". Deux diagraphes sont déduits de la description du circuit :

- un diagraphe représentant le contrôle
- un diagraphe représentant le chemin de données.

<u>Exemple</u> : A : = B+C

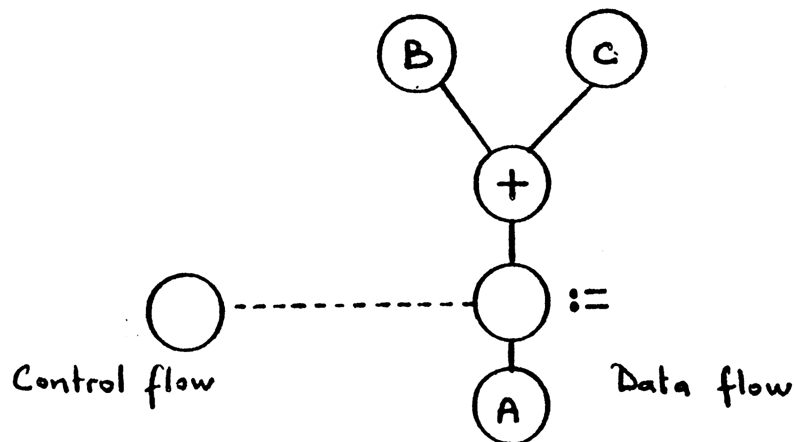


Figure 2-11.

Le principal avantage de cette approche est que l'étude du test du circuit peut être faite très tôt dans la conception du circuit. Par contre, ses inconvénients sont :

- la précision temporelle de la description algorithmique est faible

sinon nulle; ceci ne permet pas d'avoir une séquence de test temporisé. (On connait l'ordre des vecteurs de test mais on ne sait pas quand il faut les appliquer).
- problème de diagnostic, étant donné qu'on ne sait pas la structure interne du circuit.
- pour la méthodologie de conception ascendante, il faut appliquer à chaque pas un processus d'abstraction.

II.2.5 - Conclusion sur le test d'identification

Les méthodes que nous avons présentées ont été proposées pour vérifier le fonctionnement correct du circuit par rapport à sa spécification. Le test exhaustif et [ACK 78] étaient utilisables pour les circuits de très faible complexité. On peut dire que la méthode d'identification d'automate de [KOH 70] est la base de presque toutes les autres méthodes qui ont été proposées ultérieurement.

Par ailleurs, remarquons aussi que les modèles de base utilisés dans ces méthodes sont des graphes dans lesquels on recherche des chemins ; chaque chemin représente généralement une fonction du circuit.

II.3 - RECAPITULATION SUR LE TEST AVEC VECTEURS PREDETERMINES

Nous avons essayé de classer les méthodes de test qui ont été développées suivant leur principal objectif :
- test de distinction,
- test d'identification.

Dans chaque cas, nous avons présenté quelques méthodes qui diffèrent les unes des autres principalement par le type de spécification utilisée (structurelle, fonctionnelle, algorithmique) et les primitives de bases considérées (portes logique, primitives fonctionnelles, variables ...).

Nous pouvons dresser le tableau suivant résumant les méthodes de test qui ont été présentées.

TEST DE DISTINCTION

description	méthode	remarques
structurelle	D-algorithme	elts de base = portes logiques
	D-algorithme étendu	elts de base = portes + mux + + compteurs + registes
fonctionnelle	Poage	pour circuits séquentiels simples
	S-algorithme	hypothèse d'erreurs fonctionnelles + exécution symboliqe + niveau RTL
	STG	erreurs multiniveau + génération multi-étapes
algorithmique	technique GTE	

TEST D'IDENTIFICATION

description	méthode	remarques
fonctionnelle	exhaustive	pour circuits combinatoires
	Kohavi	identification d'automates simples
	graphe abstrait d'exécution	pour microprocesseurs
algorithmique	technique Texas Instruments	

Figure 2-12.

Ce tableau nous permet de conclure :

- qu'à partir d'une description fonctionnelle ou algorithmique, on peut générer des tests de distinction ou d'identification,
- qu'à partir d'une description structurelle on ne peut générer que du test de distinction.

III - APPROCHE MULTINIVEAU : TEST FONCTIONNEL DES CIRCUITS A CONTROLEUR

III.1 - STRATEGIE MULTINIVEAU

La méthode que nous allons présenter ici a été développée par [BEL 84] au Laboratoire "Circuits et Sytèmes". Elle est basée sur une approche multiniveau, à partir de la description fonctionnelle du circuit à contrôleur explicitement défini : le circuit est composé d'un contrôleur et d'une partie opérative.

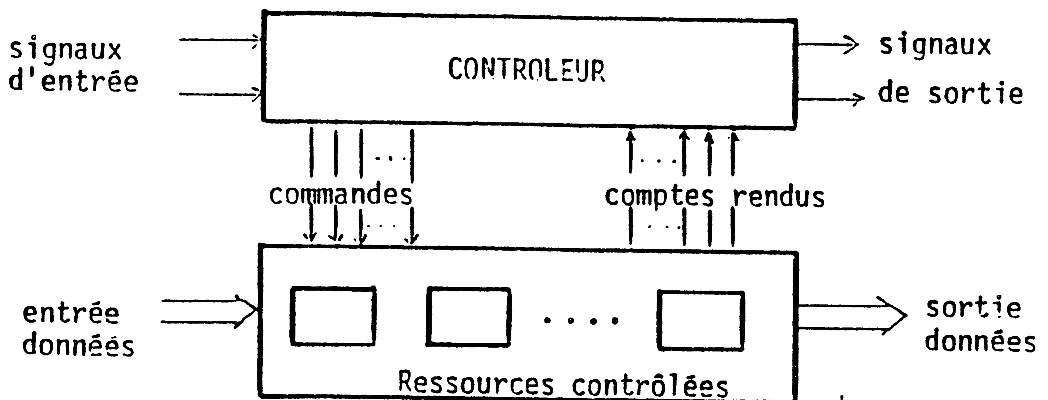


Figure 2-13.

Le contrôleur et les blocs fonctionnels de la partie opérative sont des ressources fonctionnelles en terme du langage CADOC.LD. Un circuit complexe est décrit comme un assemblage de ressources : description dite partitionnée. Chaque ressource est décrite par ses fonctions spécifiques.

La description fonctionnelle de la ressou :e contrôleur correspond à son graphe d'état, celle des ressources contrôlées spécifie l'ensemble des fonctions exécutées par celles-ci. La stratégie de partitionnement de la partie opérative est telle que les ressources contrôlées soient des ressources testables, autrement dit, pour lesquelles on sait générer des séquences ou des vecteurs de tests en utilisant les algorithmes classiques (D-Algorithmes et ses extensions, test des mémoires, test de PLA...) à partir de leur description au niveau logique par exemple.

La stratégie globale de la génération de test d'un circuit complexe qui a été proposée est multiniveau :

- étude du test au niveau de chaque ressource à partir de leur description au niveau logique
- étude de l'activation de test globale pour accomplir le test des ressources internes avec ses vecteurs de tests locaux en utilisant la description fonctionnelle du circuit en CADOC.LD.

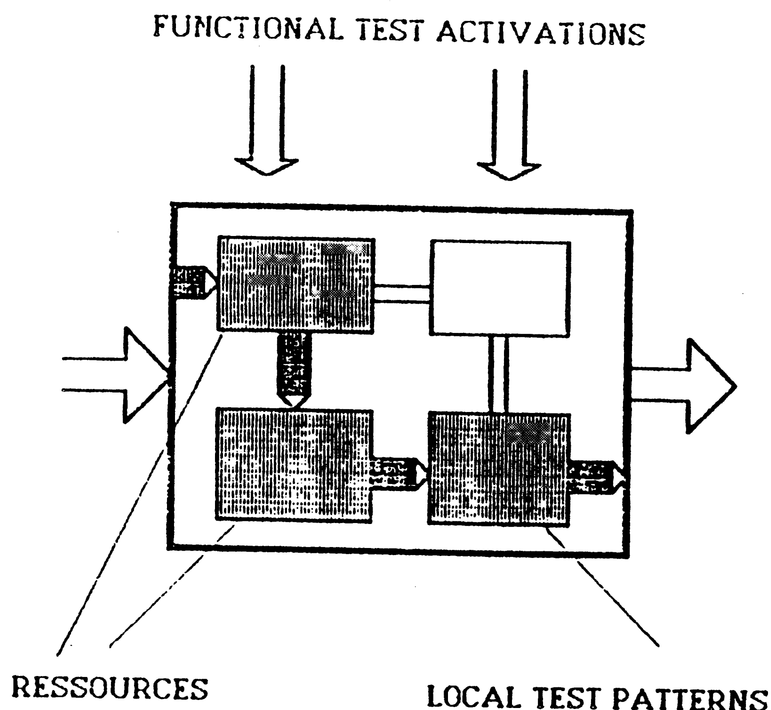


Figure 2-14.

Cette approche présente les deux avantages suivants.

(1) - précision des hypothèses de pannes et donc une meilleure couverture et localisation des pannes. En effet, les algorithmes classiques du test structurel utilisent des hypothèses de pannes structurelles, telle que collage à 0, à 1, court-circuit.... Autrement dit, utilisation d'algorithmes et d'hypothèses de fautes bien adaptées pour chaque ressource.

(2) - réduction de la complexité de la génération du test. La recherche des activations de test globales se fait à partir des descriptions fonctionnelles des ressources : les propagations en arrière et en avant utilisent ces descriptions fonctionnelles.
Un calcul de complexité est fait dans [BEL 84]. Les résultats montrent que cette approche multiniveau simplifie la génération du test, même

si la complexité des algorithmes est très grande, pour une taille des blocs (en nombre de portes logiques) assez importante.

La stratégie de génération comporte les étapes suivantes :

(1) - description du circuit partitionné en ressources de type contrôleur et en ressources contrôlées, en utilisant le langage CADOC.LD.

(2) - détermination des fonctions du circuit à partir du graphe d'états du contrôleur.

(3) - étude du test du contrôleur : test d'identification .

(4) - étude de testabilité fonctionnelle des ressources contrôlées à travers le fonctionnement normal du circuit.

(5) - étude du test d'une ressource contrôlée.
- détermination au niveau fonctionnel de l'ensemble des vecteurs ou séquences d'entrée que cette ressource peut recevoir en fonctionnement normal : ce qu'on appelle **domaine d'entrée**.
- génération du test local de cette ressource dans son domaine d'entrée.
- détermination des résultats attendus.

(6) - propagation des résultats au niveau fonctionnel.

La description fonctionnelle du contrôleur doit respecter les contraintes suivantes :
- une place du graphe (GIT) correspond à un état du contrôleur
- l'effet du signal "RESET" n'apparait pas dans le graphe et est indiqué par une assertion dynamique.

L'outil utilisé pour supporter cette approche est l'outil d'exécution

symbolique temporisée. Cet outil a été défini pour les circuits synchrones à contrôleur. Un enrichissement de cet outil est présenté au chapitre III.

III.2 - TEST D'IDENTIFICATION D'UN CONTROLEUR

[BEL 84] a étudié le test des contrôleurs dans les configurations suivantes :

- contrôleur isolé défini par ses entrées commandables (signaux externes et compte-rendus) ses sorties observables (commandes et signaux externes) et par son graphe d'états décrit en CADOC.LD
- contrôleur sans parallèlisme, avec une évolution synchrone : prise en compte des signaux asynchrones synchronisés par l'horloge du circuit.
- il existe un signal d'initialisation "RESET" qui met le contrôleur dans son état initial (place initiale).

En remarquant que le graphe d'un tel contrôleur est fortement connexe, et que les transitions activées par le signal "RESET" ne permettent pas d'identifier les états avant l'activation du signal, l'étude du test du contrôleur est faite à partir du graphe fonctionnel : dans ce graphe, toutes les transitions conditionnées par "RESET" ont été supprimées.

Le test du contrôleur consiste à vérifier que toutes les fonctions du circuit sont exécutées correctement : test d'identification du contrôleur.
Le test d'identification d'un contrôleur est constitué de trois étapes :

1 - application du "RESET" pour le mettre dans l'état initial,

2 - identification des états du contrôleur étudié à partir de son graphe fonctionnel,

3 - test de la prise en compte du "RESET".

Nous avons déjà remarqué dans le paragraphe II.1.2 que la méthode d'identification d'automate proposée par [KOH 70] n'est pas praticable pour

les automates de plus de deux entrées et de plus d'une dizaine d'états.

Cependant, le contrôleur d'un circuit complexe est un automate de Moore : il a en moyenne seulement deux fois plus d'états que de vecteurs de sortie distincts.

Exemple :
Le microprocesseur MC 6800 a 519 états et 336 vecteurs de sorties (rapport 1,52).
Le microprocesseur HSURF a 127 états et 80 vecteurs de sorties (rapport 1,59)

La méthode d'identification proposée est inspirée de la méthode classique d'identification d'automate : identification des états ou distinction de l'état courant de tout autre état. [BEL84] a enoncé un certain nombre de définitions et de propriétés basées sur l'identification des états par les sorties du contrôleur. Nous rappelerons, ici, les définitions de base utilisées dans cette méthode.

Définitions :

1 - un état est **O-identifié** s'il n'existe pas d'autres états ayant les mêmes sorties que celui-ci.

2 - un état P_i est **C-distinguable** d'un autre état Pj ayant le même vecteur de sortie si tout autre chemin de même longueur d'origine Pj n'a pas la même séquence de sortie.

3 - un état Pi est **C-identifiable** par un de ses chemins successeurs si ce chemin permet de "C.distinguer" Pi de tout autre Pj ayant le même vecteur de sortie.

Exemple :

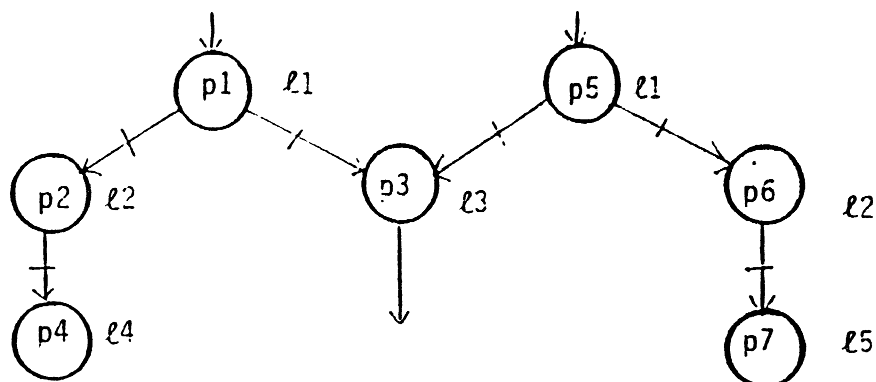


Figure 2-15.

- P1 et P5 ne sont pas O.identifiés : ils ont le même vecteur de sortie ℓ1.
- Mais P1 est C.distinguable de P5 par le chemin de longueur 3 (P1P2P3), car le chemin de longueur 3 d'origine P5 a comme séquence de sorties (ℓ1,ℓ2,ℓ5) différentes de (ℓ1,ℓ2,ℓ3).
- P1 n'est pas C.distinguable de P5 par le chemin (P1,P4).
- P1 est C.identifiable par (P1,P2,P3).

A partir des propriétés des états (O-identifié, C-identifiable, ...), on peut déterminer un ensemble de chemins du GIT permettant d'identifier tous les états.

Par cet ensemble de chemins de test (élaboré précédemment), on a vérifié en même temps la fonction de sortie du contrôleur et une partie de la fonction de séquencement (les transitions non couvertes par cet ensemble de

chemins sont à tester);

On appelle **test de couverture** d'un contrôleur un test qui cherche à activer au moins une fois toutes les transitions du graphe fonctionnel. L'ensemble minimal de chemins constituant un test de couverture peut être déduit à partir de l'arbre d'exécution symbolique en dressant un tableau de couverture.

[BEL 84] a établi un ensemble de conditions suffisantes pour qu'un test de couverture réalise un test d'identification du contrôleur :

- le contrôleur doit être un automate fortement réduit
- tous les états d'itération sont 0-identifiés.

En exécutant symboliquement tout l'ensemble de chemins élaborés précédemment, on obtient les conditions de chemins temporisés et les séquences de sorties attendues. A partir des conditions de chemins temporisées, on peut déterminer les séquences d'entrée de test en fixant :

- les instants de référence des chemins
- les valeurs des entrées.

III.3 - TEST DE LA PARTIE OPERATIVE

La partie opérative désigne l'ensemble des ressources contrôlées. Le test de la partie consiste donc à étudier le test de chaque ressource contrôlée. Ce test est fait, à travers les fonctions réalisées par le circuit : test fonctionnel.
L'outil utilisé pour mettre en oeuvre ce test est l'exécution symbolique temporisée [RAR85]. Pour chaque ressource, elle permet de déterminer :

- les domaines de valeurs en entrées des ressources, c'est à dire, l'ensemble des valeurs que ses entrées peuvent recevoir à travers les différentes fonctions du circuit.

- les ressources activées par un chemin de l'arbre des chemins d'exécution symbolique.

La description, à partir de laquelle est basée l'étude du test d'une ressource contrôlée est la description comportementale du circuit : une ressource contrôlée du type mémoire, registre ou bus est représentée par une variable interne, et une ressource contrôlée de type combinatoire est représentée implicitement au travers des opérateurs arithmétiques ou logiques du langage CADOC.LD. Autrement dit, à une place du GIT est associée l'opération effectuée par la partie opérative.
Si cette description n'est pas disponible, [BEL84] a étudié comment celle-ci peut être générée à partir de la description du contrôleur et des descriptions des ressources contrôlées, et en utilisant l'exécution symbolique. D'une manière simplifiée, l'opération effectuée par la partie opérative est le résultat de l'exécution symbolique des commandes du contrôleur en utilisant la description des ressources contrôlées.

Remarque
Un des problèmes d'implémentation que pose cette deuxième démarche est la détermination automatique des correspondances entre les variables internes de description et les ressources contrôlées d'une part, entre les opérations du langage et les parties combinatoires d'autre part.

L'étude du test d'une ressource contrôlée consiste à déterminer à partir de résultats d'une étude préalable de testabilité fonctionnelle, un ensemble de chemins qui permet :
- d'activer une fonction de la ressource contrôlée,
- d'initialiser les entrées de la ressource,
- d'observer les valeurs de sortie de la ressource.

L'étude de testabilité fonctionnelle est une étude qualitative des chemins permettant de choisir les chemins d'initialisation ainsi que

d'observation et d'évaluer la difficulté de générer le test. Ceci permet d'éviter l'exécution symbolique de tous les chemins possibles. Elle est basée sur la notion de relations de dépendances des valeurs des variables.

L'exécution symbolique de l'ensemble de chemins pour le test d'une ressource contrôlée fournit :

- les conditions de chemins temporisées
- les domaines d'entrées de la ressource . l'ensemble des valeurs que peuvent prendre les entrées de la ressource à travers cet ensemble de chemins.
- les séquences de sorties attendues aux points d'observation.

La génération du test effectif se décompose en :

- choix des valeurs effectives de test aux entrées primaires du circuit respectant les conditions de chemins et générant un ensemble de vecteurs de test de la ressource contrôlée.
- détermination des séquences d'entrée .
- détermination des séquences de sortie.

<u>Génération multiniveau de test</u>

Dans une démarche ascendante, les vecteurs de test d'un bloc sont générés indépendamment de ses domaines d'entrée. Ensuite, a partir de la description fonctionnelle du circuit, on détermine un ensemble de chemins qui permet d'envoyer ces vecteurs de test. Cette démarche nécessite la connaissance de fonctions inverses, c'est à dire celles qui font correspondre une entrée du circuit à un vecteur de test d'une ressource contrôlée, ou la résolution d'équations.

Dans une démarche descendante, un ensemble de chemins de test est premièrement déterminé. L'exécution symbolique des chemins nous informe, entre autres, sur les domaines d'entrée de la ressource à tester. Ensuite, la

génération de vecteurs de test se fait dans les domaines d'entrée de la ressource utilisant les ATPG (Automatic Test Pattern Generator) classiques.

La détermination des séquences de test se fait à partir des échéanciers des entrées en substituant les valeurs symboliques par les valeurs pour le test et en fixant les valeurs "indifférentes" (X).

La détermination des séquences de sortie attendues peut être faite de deux façons :

- à partir des résultats de l'exécution symbolique, en recalculant les expressions à partir des valeurs des entrées effectives,
- par simulation en utilisant le simulateur CADOC.SIM.

III.4 - CONCLUSION

La méthode de génération de test des ressources contrôlées à travers le fonctionnement normal du circuit a été présentée dans ce paragraphe. Cette génération est multiniveau :

- les vecteurs de test sont générés au niveau logique.
- les séquences de test en entrée et en sortie du circuit sont générées à partir de la description fonctionnelle globale du circuit en utilisant l'exécution symbolique. Les chemins de test sont déterminés après une étude de testabilité fonctionnelle.

Cette approche est très intéressante pour deux raisons :

- on considère des hypothèses de pannes structurelles ; ceci est plus réaliste en terme de couverture de défauts physiques.
- on utilise la description fonctionnelle pour la génération des séquences de valeurs des entrées primaires et des sorties primaires ; ceci permet une génération plus rapide qu'une génération classique au niveau logique.

IV - CONCLUSION

Nous avons présenté dans ce chapitre les principales méthodes qui ont été proposées pour résoudre le problème de la génération du test des circuits intégrés. Nous pouvons retenir que les approches classiques ont l'avantage d'être pratiquement expérimentées mais ne sont plus adaptées au test des circuits LSI et VLSI. C'est alors que de nouvelles méthodologies de conception sont proposées pour obtenir des circuits facilement testables : utilisation des principes du "Scan Path".

Mais cette solution n'est pas utilisable car elle nécessite une augmentation de 20 % de la surface du silicium dégradant ainsi la fiabilité du circuit. La tendance actuelle est alors vers les approches de test fonctionnel. Une approche purement fonctionnelle présente un inconvénient qui est la précision de la couverture et de la localisation des défaillances. Alors une approche multiniveau a été proposée (cf. paragraphe III), qui assurent une bonne couverture des pannes et aussi une réduction du temps de génération de test. Dans cette étude, on a considéré les types de circuits complexes avec une partie contrôle PC et une partie opérative.

CHAPITRE III : OUTIL D'EXECUTION SYMBOLIQUE TEMPORISEE

I - INTRODUCTION

La technique d'exécution symbolique a été élaborée pour les problèmes de test de programmes [KIN 76]. Elle consiste à exécuter un programme avec des valeurs symboliques au lieu de valeurs numériques. Si pour l'exécution conventionnelle (ou numérique) les opérations entre les valeurs sont évidentes, pour l'exécution symbolique, des règles d'opérations entre les symboles doivent être définies.

Par exemple, pour l'addition, il faut définir les règles suivantes :

- A + A = 2*A
- A + (A) = A - A = 0
- A + B = B + A
- (A + B) + C = A + (B + C)
- A + 0 = A
- ....

Pendant l'exécution symbolique, les valeurs des variables sont des expressions sur des valeurs symboliques au lieu de valeurs numériques.

Exemple :

```
     Program T ;
     var A : entier ;
         I : entier ;
     begin
---> | A := (J+1)*I ;
```

notation $... précède une valeur symbolique pour la différencier du nom de la variable

Si J = $J1 est I = $I1 avant l'exécution de l'instruction courante (pointée par ⟶) alors A := ($J1 + 1) * $I1

expression symbolique (désigne ($J1 + 1) * $I1)

Une valeur symbolique (précédée de $) étant une valeur fixée mais non connue, l'expression symbolique a donc aussi une valeur fixée mais non connue.

Un autre point qui est propre à l'exécution symbolique est le problème que posent les instructions conditionnelles : le problème est de décider quelle alternative prendre.

Par exemple :

```
(1) y := x ;
(2) si x < 0 alors y := y-1 ;
             sinon y := y+1 ;
```

Soit x = $X et y = $y avant l'exécution de l'instruction (1)
Après l'exécution de (1), x=$X et y=$X, le test sur la valeur de x est indéterminé. La solution est de poser des conditions sur la valeur symbolique de x.
Dans le cas où $X < 0 alors y = $X-1 ;
si $X > 0 alors y = $X+1 ;

Si on représente ces deux instructions en chemins, on aura l'arbre de la figure 2-16.

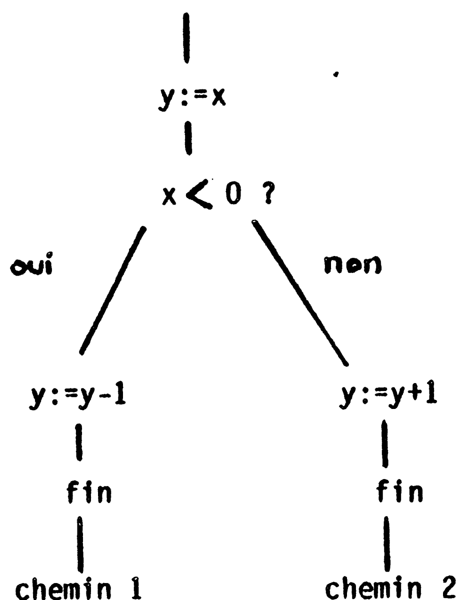


Figure 2-16.

Cet arbre représente l'arbre de chemins d'exécution symbolique du programme de l'exemple précédent. La condition $X < 0 correspond à l'exécution du chemin 1 et la condition $X > 0 à celle du chemin 2 : ces conditions sont appelées **conditions de chemins**

En résumé, à chaque point de l'exécution symbolique du programme, on peut déterminer :

- l'état des variables,
- les conditions du chemin suivi.

Un autre problème qui est analogue à ce qu'on vient de voir est le problème des boucles. Nous en reparlerons plus tard.

L'exécution symbolique a été utilisée dans deux domaines :

(1) - **le test de programmes** [KIN 76] Les résultats de l'exécution symbolique sont comparés aux résultats attendus.

(2) - **preuve de programmes** [DAN 81] en utilisant des assertions de types "supposons que" et "prouvons que". La condition supposée constitue une condition de chemin. Et les résultats de l'exécution doivent impliquer la conclusion donnée dans "prouvons que".

L'exécution symbolique a été aussi proposée pour la vérification de conception de matériel, par exemple pour prouver par comparaison la cohérence de deux niveaux successifs de descriptions d'un circuit [COR 81]. Elle a été proposée comme outil d'aide à la génération du test [LIN 84] en comparant les résultats d'exécution symbolique de la description du circuit correct et de celle des circuits "faux" (cf. paragraphe II.1.5). Dans l'approche proposée au chapitre IV, l'exécution symbolique est utilisée comme outil de génération de test dans la mesure où elle permet de sélectionner des classes de valeurs de test et de déterminer des conditions de chemins de test.

Cependant, l'exécution symbolique telle qu'elle a été définie pour les programmes ne permet de traiter que les circuits combinatoires. Pour traiter les circuits séquentiels, il est nécessaire d'introduire la dimension "temps". Nous proposons ici une technique d'exécution symbolique temporisée qui définit des séquences temporisées de valeurs symboliques d'entrées et de sorties. Une version de cette technique a été définie dans [BEL 84], mais seulement pour les circuits à contrôleur synchrone explicite : la manipulation du temps est basée sur la notion de période d'horloge. Nous proposons dans ce chapitre une version enrichie de cette technique dans laquelle la manipulation du temps est basée sur la notion d'événements.
Les tâches principales constituant l'outil d'exécution symbolique sont:

- temporisation d'un chemin. Cette tâche nous définit les instants de référence.

- détermination des conditions de chemins temporisées. Cette tâche établit les échéanciers des entrées primaires du circuit et certaines conditions sur les valeurs des entrées.

- exécution symbolique des actions associées aux noeuds qui construit les échéanciers symboliques de toutes les variables, en particulier ceux des variables de sorties.

II - TEMPORISATION D'UN CHEMIN

Il faut remarquer que les informations temporelles (réceptivité) sur le GIT sont locales, et possèdent donc comme référence la date d'activation des places en amont de la transition.

Pendant l'exécution symbolique on n'active qu'un seul chemin de l'arbre ; il est donc possible de référencer toutes les informations temporelles globalement.
Pour un chemin complètement synchronisé par rapport à un signal de type **horloge** par exemple, l'instant d'activation de la place initiale du chemin et les instants d'échantillonage des signaux asynchrones par l'horloge sont suffisants comme références. Le franchissement des transitions du chemin est complètement déterminé par le numéro de la période de l'horloge [BEL84].

Ceci n'est pas toujours le cas des circuits complexes. Leur communication avec le monde extérieur se fait par l'intermédiaire de signaux asynchrones qui ne sont pas toujours échantillonés par l'horloge.

Sur un exemple qui est un circuit de correlation (CCT8) conçu par l'Institute of Microwave Technology de Stockholm (Suède), nous avons rencontré des problèmes de temporisation des chemins d'exécution symbolique. En effet, d'après la description du CCT8 faite par les concepteurs, il existe des transitions dont les conditions sont échantillonnées sur un front d'un signal qui n'est pas l'horloge. Nous avons alors généralisé la manipulation du temps dans l'exécution symbolique temporisée du GIT : l'horloge sera considérée au même niveau que tous les autres signaux.

Quelques exemples donnés dans ce chapitre sont tirés du circuit CCT8 que nous avons étudié dans le cadre d'une étude de l'utilisation d'un Microscope Electronique à Balayage (MEB) ; le rapport détaillé de cette étude est dans [BAL 85].

II.1 - NOEUDS-DE-REFERENCE

Soit un noeud N d'un chemin donné , correspondant à une place du GIT ayant n transitions successeurs vers d'autres places différentes de N. Soient P1, P2, ... , Pn les prédicats respectifs et échantillonnés sur les événements e_{c1}, ..., e_{cn} (front montant ou descendant d'un signal, ou l'événement toujours vrai)(cf.fig.2-17)

Soit e_p (précédent événement), l'événement échantillonnant le prédicat de l'arc en amont du noeud N (cet arc est unique si on considère un chemin donné).

Le noeud N est un noeud de référence pour un chemin donné s'il satisfait l'un des cas suivants:

- les receptivités sur l'arc en amont et en aval d'un noeud n'est pas échantillonnées par le même événement.

- ou les réceptivités sont échantillonées par le même événement ; mais la somme des prédicats associés à toutes les transitions successeurs de P n'est pas une fonction booléeene toujours VRAIE ($\Sigma P_i \neq 1$).

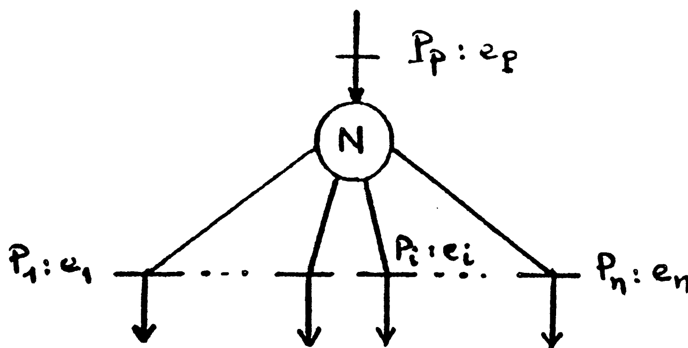


Figure 2-17.

Exemple :

Cet exemple est tiré du CCT8.

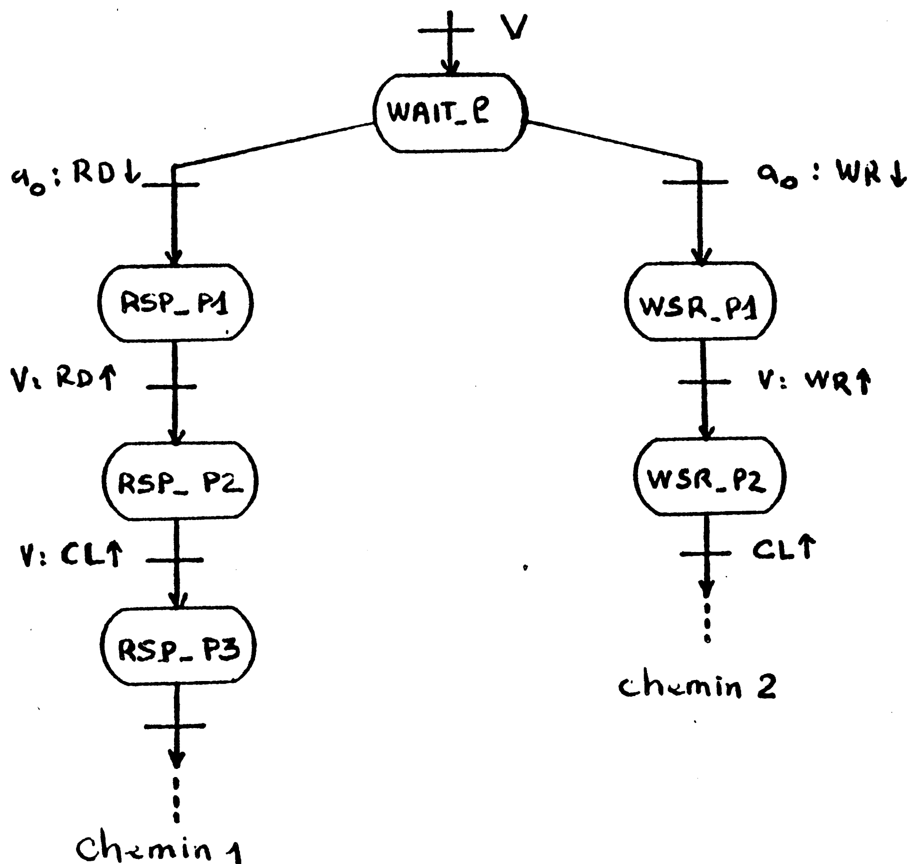


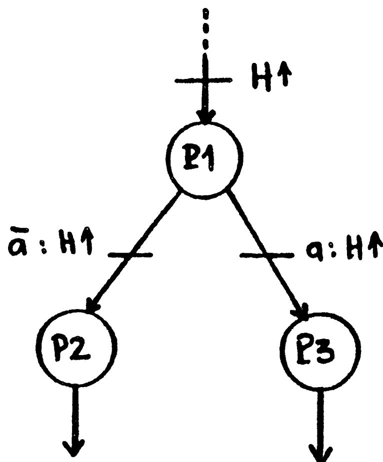
Figure 2-18.

WAIT_P est un noeud de référence quelque soit le chemin parcouru, car les prédicats en amont et en aval sont "échantillonnés par deux événements différents.

De même pour RSR_P1 dans le chemin 1.

RSR_P4 du chemin 1 n'est pas un noeud de référence.

Exemple



$\sum P_i$ = non (a) + a = 1 et ils sont échantillonnés par H↑ donc P1 n'est pas un noeud de référence.

Figure 2-19.

II.2 - DEFINITION DES INSTANTS DE REFERENCES

L'instant de franchissement de l'arc en aval d'un noeud de référence définit un instant de référence symbolique pour le chemin auquel appartiennent le noeud et l'arc, noté θ_i.

Par ailleurs, nous définissons par θ_0 l'instant de référence initial correspondant à l'instant d'activation de la place initiale.

L'arc correspondant à un instant de référence est noté par une double barre sur le chemin.

Remarque : l'instant de référence dans [BEL 84] est défini à partir de la notion de phase d'attente. Une phase d'attente est un noeud de référence qui satisfait un sous-cas de notre définition car les circuits considérés sont synchrones et donc les e_{ci} sont tous égaux sur tous les chemins. Une telle définition ne permet pas de temporiser les chemins des circuits évoluant sur

d'autres événements que les fronts d'horloge, par exemple le circuit CCT8.

Exemple :
La temporisation d'une portion de chemin du CCT8 est :

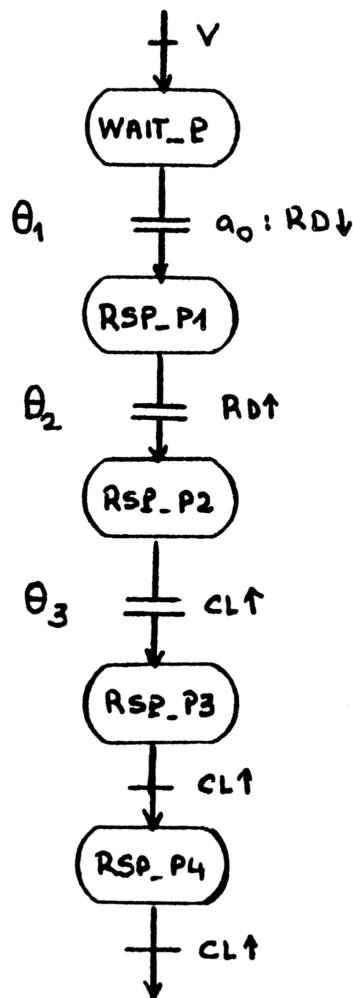


Figure 2-20.

Propriété :

Entre deux instants de référence successifs séparés par au moins un noeud, on a toujours le même événement d'échantillonage que celui correspondant au premier instant de référence.

Par conséquent, on peut définir un compteur d'événements entre deux instants de référence. Ce compteur sera utilisé par la tâche de temporisation pendant l'éxécution symbolique d'un chemin (cf. paragraphe II.3).

Pour la définition des instants de références,nous utiliserons les opérateurs temporels de CADOC.LD avec les mêmes sémantiques. Ces opérateurs sont :

- FD, FDX (front descendant (indexé))
- FM, FMX (front montannt (indexé))

Nous rappelons ici brièvement la syntaxe et la sémantique de ces deux opérateurs.

<u>Syntaxe</u>

FDX (variable, index, instant)

FMX (variable, index, instant)

<u>Sémantique</u>

(1) "index" est un entier strictement positif
FD et FM sont des cas particuliers de FDX et FMX (index = 1).

(2) "instant" est une expression temporelle pouvant inclure des opérateurs temporels (FDX, FMX,...)

(3) la signification est la suivante : FDX (resp. FMX) détermine le "index" ième front descendant (resp. front montant) de la "variable" après la date "instant".

Ces opérateurs sont applicables pour n'importe quelle variable, l'horloge est un cas particulier.

Un instant de référence est en fait défini par deux paramètres :

- α_i, la valeur au plus tôt de θ_i par rapport à l'événement précédent
- β_i, la valeur effective de θ_i, (partie condition de la réceptivité satisfaite)

Ces deux paramètres sont liés par la relation $\alpha_i < \beta_i$.

Exemple

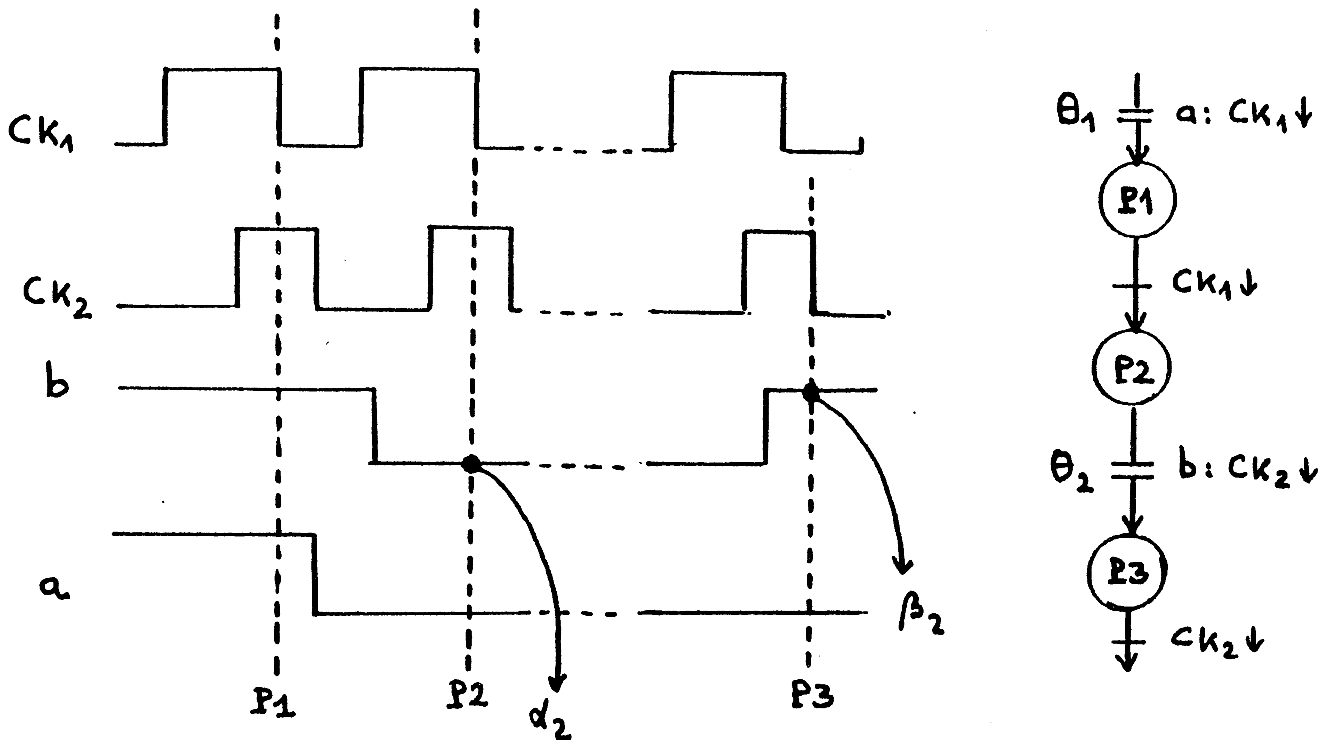


Figure 2-21

On exprimera α_2 ainsi :

$\alpha_2 = fdx(CK_1, 2, \Theta_1)$

et β_2 sera :

$\beta_2 = fdx(CK_2, 1, \alpha_2)$

Donc la définition de Θ_2 sera :

$$\Theta_2 \begin{cases} \alpha_2 = fdx(CK_1, 2, \Theta_1) \\ \beta_2 = fdx(CK_2, 1, \alpha_2) \text{ avec } \alpha_2 < \beta_2 \end{cases}$$

Mais en utilisant la récursivité de la fonction "fdx" par rapport à son paramètre "instant" (cf. sémantique donnée), on peut écrire :

$$\beta_2 = fdx(CK_2, 1, \underbrace{fdx(CK_1, 2, \Theta_1)}_{\alpha_2})$$

Ainsi donc , une seule expression définit Θ_2 et on écrira :

$$\Theta_2 = \underset{\text{partie condition}}{b} \; : \; \underbrace{fdx(CK_2, 1, fdx(CK_1, 2, \Theta_1)}_{\text{partie temporelle}}$$

Ceci veut dire que :

Θ_2 est l'instant de référence correspondant au premier front descendant de CK après que B soit égale à 1 (VRAI) et après le $2^{ième}$ front descendant suivant Θ_1.

Exemple (tiré du CCT8)

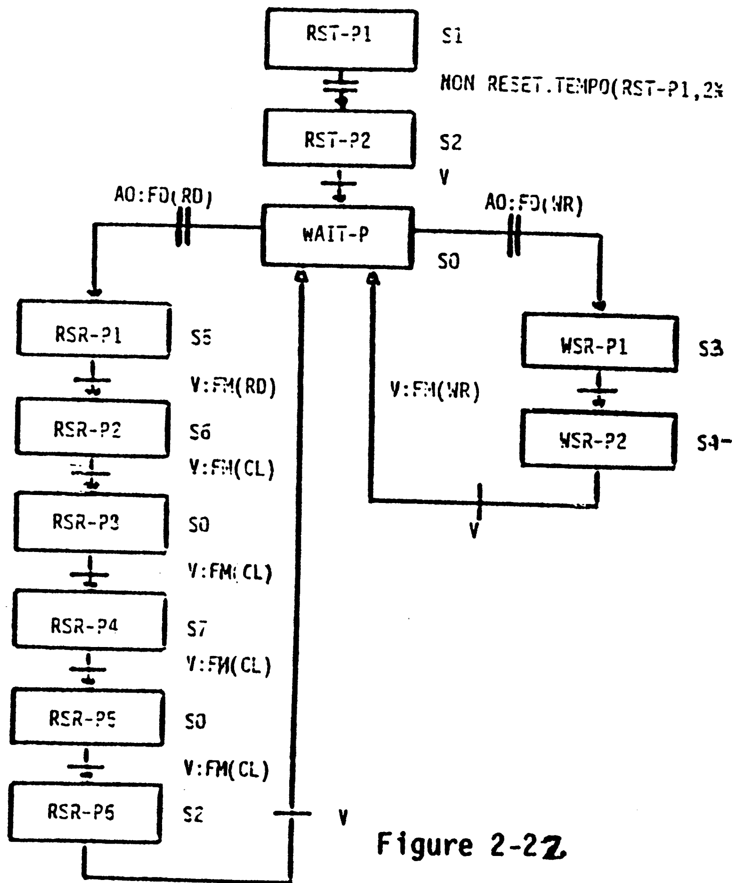


Figure 2-22

Définition des instants de référence :

Θ_0 : instant initial.

Θ_1 = non (RESET) et tempo (RST_P1, 2*T) : e (Θ_0)

Θ_2 = a_0 : fdx(RD, 1, Θ_1)

Θ_3 = fmx(RD, 1, Θ_2)

Θ_4 = fmx(CK, 1, Θ_3)

Θ_5 = a_0 : fdx(WR, 1, fmx(CK, 3, Θ_4)

Θ_6 = fmx(WR, 1, Θ_5)

- pour Θ_1, nous avons utilisé l'opérateur e(Θ_0) qui correspond à l'événement toujours présent : il signifie "événement toujours présent" après Θ_0.

- pour Θ_5, α_5 = fmx(CK, 3, Θ_4). Si nous regardons le graphe, le dernier instant de référence est Θ_4 et de plus, entre Θ_4 et Θ_5 il doit exister au moins trois fm(CK). Le compteur d'événements est à 3.

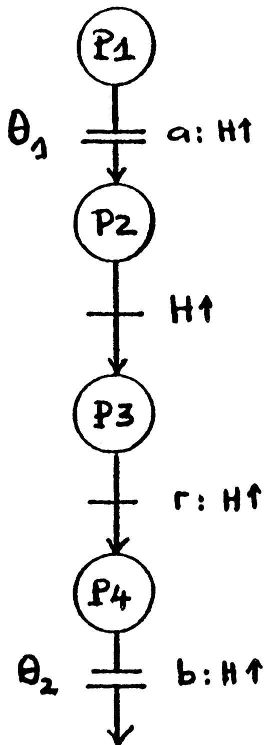
II.3 - TEMPORISATION D'UN CHEMIN

La tâche de temporisation de chemin pendant l'exécution symbolique d'une description d'une ressource est appelée à chaque évolution : une évolution est le franchissement d'un arc d'un chemin (en simulation, une évolution correspond aux franchissements des transitions franchissables [CRA 85]).
Cette tâche determine la caractéristique du noeud courant (noeud de référence ou non).

Dans le cas où c'est un noeud de référence, elle définit un nouvel instant de référence à partir du dernier instant de référence et de la valeur du compteur d'événements. Le compteur d'événements est incrémenté de 1 si le prédicat sur l'arc en aval du noeud est échantilonné sur le même événement correspondant au compteur d'événements. Ceci est fait afin de respecter la condition $\alpha_i < \beta_i$. Sinon le compteur ne sera pas incréménté (cf. exemples).
Après la définition du nouvel instant de référence, ce compteur est remis à 0 et est associé au nouvel événement.

Dans le cas où le noeud n'est pas un noeud de référence, la tâche détermine l'instant symbolique courant t qui est exprimé avec les opérateurs temporels définis plus haut. Le compteur d'événements est incrémenté de 1 (cf. propriété énoncé dans le paragraphe II.2). Et l'opérande "index" des opérateurs temporels (fmx, fdx) prend comme valeur celle du compteur d'événements. L'opérande "instant" prend la valeur du dernier instant de référence défini (cf. exemples).

Exemple



	compteurs d'événements	temps symbolique
	0	θ_1
	1	$fmx(H,1,\theta_1)$
	2	$fmx(H,2,\theta_1)$
avant	3	$fmx(H,3,\theta_1)$
après	0	$\theta_2 = b:fmx(H,1,fmx(H,3,\theta_1))$

Figure 2-23.

Deuxième exemple tiré du CCT8.

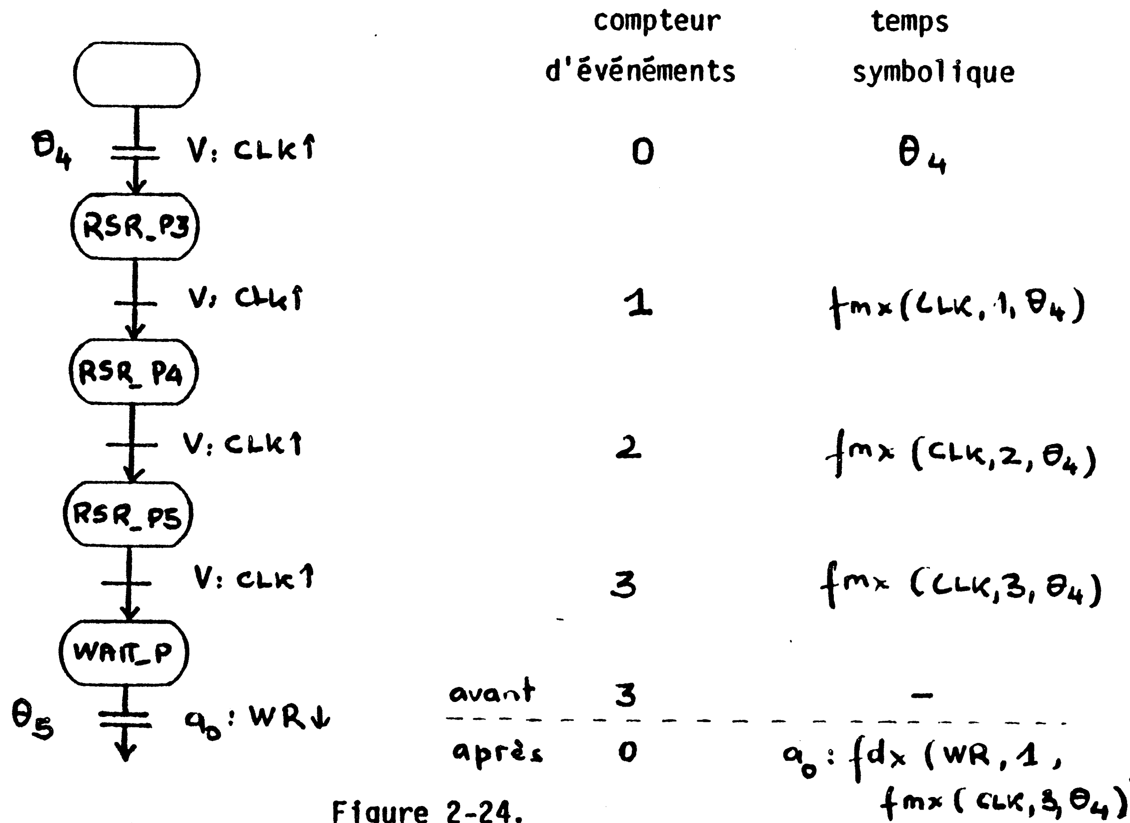


Figure 2-24.

Nous avons déterminé dans ce paragraphe les conditions temporelles pour l'exécution symbolique d'un chemin de l'arbre. Dans le paragraphe suivant nous étudierons la détermination des conditions de chemins sur les données et les états des variables (d'E/S et internes) aux noeuds terminaux des chemins.

III - CONDITION DE CHEMINS TEMPORISEE

III.1 - DEFINITION

La condition de chemins temporisée est l'ensemble des échéanciers des variables d'entrées et des relations sur ces entrées résultant de l'exécution symbolique d'un chemin du GIT d'une ressource. Ces échéanciers ne définissent pas complètement l'évolution des variables mais une classe d'évolutions acceptables (cf. chapitre IV, paragraphe II.4).

Exemple :

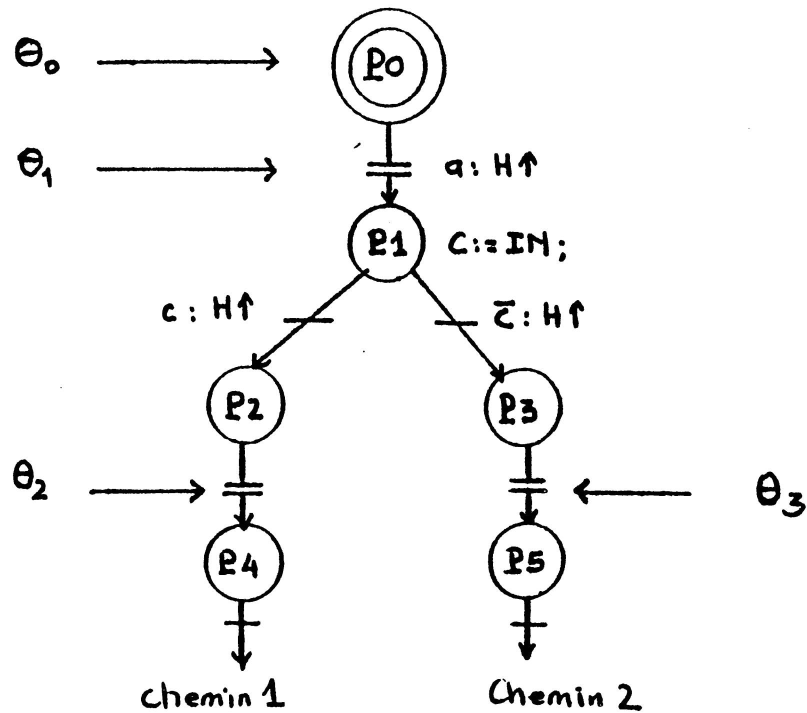


Figure 2-25.

Condition de chemin 1

(1) $a = (X, \theta_0)(1, \theta_1 - r)(X, \theta_1 + r')$

(2) $b = (X, \theta_0)(1, \theta_1 - r_1)(X, \theta_1 + r_1')$

$c = (X, \theta_0)(\$IN, \theta_1)$

et c doit être VRAI à l'instant $fmx(H, 1, \theta_1)$.

"c" étant une variable interne, et étant donné que sa dernière valeur avant $fmx(H, 1, \theta_1)$ est \$IN, nous en déduisons que :

(3) $IN = (X, \theta_0)(\ IN, \theta_1)$

et que IN = VRAI(4).

Les 4 expressions définissent la condition temporisée du chemin 1.

Condition de chemin 2

(1) $a = (X, \theta_0)(1, \theta_1 - r)(X, \theta_1 + r')$
(2) $b = (X, \theta_0)(0, \theta_3 - r\ )(X, \theta_3 + r_1')$
(3) $IN = (X, \theta_0)(\ IN, \theta_1)$ et $IN = 0$ (4)

L'exemple ci-dessus est un exemple d'un circuit synchronisé sur l'horloge. Dans [BEL 84], il n'est pas nécessaire de définir l'échéancier de l'horloge. Dans le cas général, on doit inclure dans les conditions de chemins l'échéancier de tous les variables d'entrée utilisées dans chaque chemin.

A chaque évolution de l'exécution symbolique, on rencontrera l'une des trois situations suivantes :

cas 1

La condition de chemin déjà établie ou les états des variables internes vérifient le prédicat de l'arc à franchir ; dans ce cas la condition de chemin reste la même.

cas 2

la condition de chemin déjà établie et les états des variables internes n'impliquent pas le prédicat de l'arc à franchir ; dans ce cas la condition de chemin temporisée est mise à jour. Les valeurs des variables d'entrées pour franchir l'arc sont ajoutées à la condition de chemin.

Exemple

évolution de la
condition de chemin

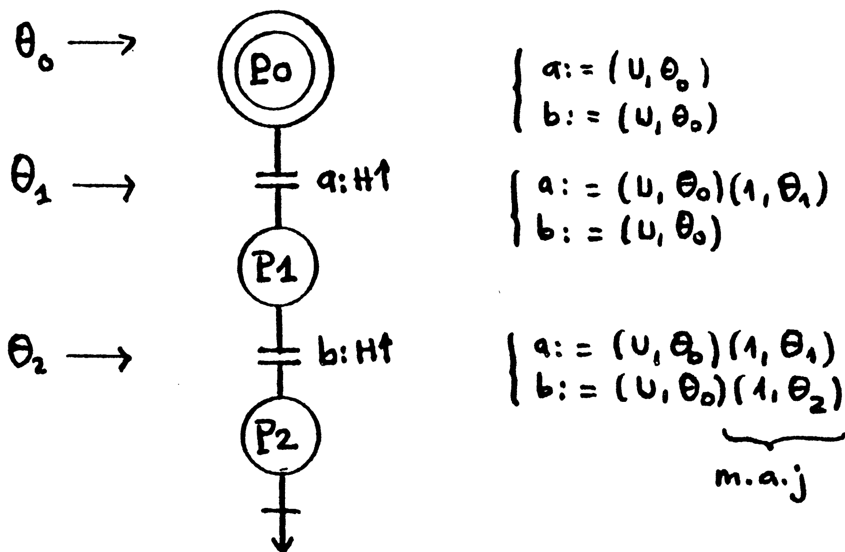


Figure 2-26.

cas 3

Il existe un choix de chemins successeurs du noeud courant. La condition de chemin déjà établie et l'état des variables internes n'impliquent aucun des prédicats : on doit choisir un chemin à suivre. Cette décision doit être suivie de la mise à jour de la condition de chemin.

Exemple

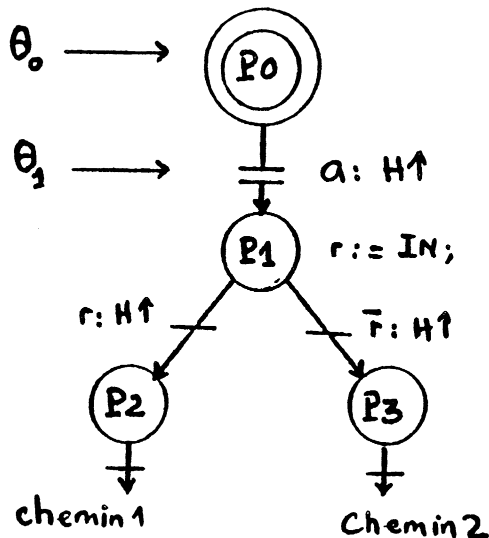


Figure 2-27.

Si on prend le chemin 1 alors r étant égale à IN implique que la condition pour passer en P2 est $IN=1. Nous pouvons exprimer ceci autrement en utilisant l'opérateur "val (<variable>,<date>) ; nous y reviendrons plus tard.

On peut mémoriser l'état de toutes les variables internes et de sorties ainsi que la condition de chemin temporisée en ce point de décision. Ceci permettra de prendre le chemin 2 après l'exécution du chemin 1 en ce point. Dans le cas général, on empilera tous les point de décision ainsi que le contexte correspondant (états des variables internes/sorties/entrées). A chaque fin d'exécution d'un chemin on reprendra l'exécution au dernier point de décision dont le contexte se trouve au sommet de la pile. Quand la pile est vide, on a parcouru tous les chemins et on obtient la condition de chemin temporisée pour chaque chemin.

Définition de l'opérateur "VAL"

Comme nous avons remarqué précedemment, nous pouvons utiliser un opérateur qui nous donne la valeur d'une variable interne en fonction des entrées du circuit. Nous utiliserons alors l'opérateur "VAL" définit comme suit:

Syntaxe : "val" "(" <variable> "," <date> ")"

Sémantique : l'opérateur "val" permet d'accéder à la valeur symbolique de la <variable> à la date symbolique <date>.

remarque

L'utilisation de cet opérateur sera beaucoup plus claire pendant la phase d'unification des règles pour la génération fonctionnelle de spécification de test (cf. chapitre IV).

III.2 - PRISE EN COMPTE DES CONTRAINTES DE STABILITE

D'après la définition d'un échéancier en CADOC.LD, il faut remarquer dans l'exemple donné ci-dessous que la variable a reste à '1' pendant au moins r + r' unités de temps. Après cette durée, sa valeur est indifférente. Ceci montre la puissance et la souplesse d'un échéancier car on peut y inclure les contraintes de stabilité des entrées dans la condition de chemin temporisée.

Les spécifications de stabilité sont données dans la description du circuit sous forme d'assertions. [BEL 84] a donné trois cas de stabilité. Ici, nous nous contenterons de donner une proposition de syntaxe générale pour tous les cas de figure.

Syntaxe :

variable '/' évènement ':' '(S1 , S2 ')'

Sémantique :
(1) variable est une variable d'entrée qui est l'objet de la spécification de stabilité
(2) évènement est une référence temporelle
(3) S1 , S2 sont les deux paramètres de stabilité

La figure ci-dessous montre un exemple

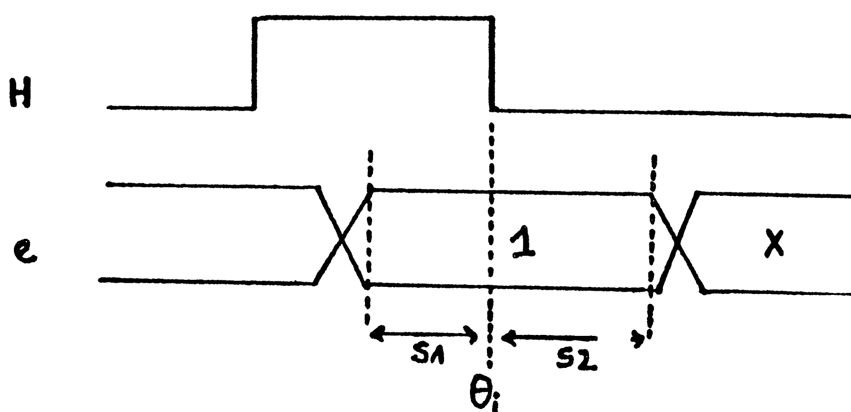


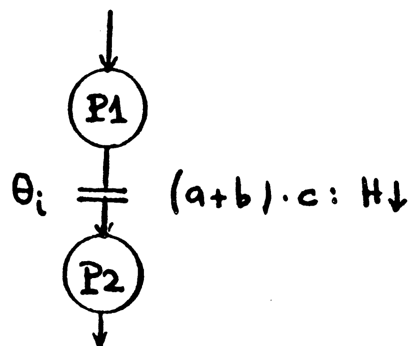
Figure 2-28.

e := (Φ,θ) ... (1,θ_i-S1) (X,θ_i+S2)...

III.3 - PREDICATS COMPLEXES

Dans l'exemple que nous avons donné précédemment, la valeur des entrées a et b ont pu être déterminée parce que les prédicats associés aux transitions étaient simples. Ce n'est pas toujours le cas ; les prédicats peuvent être des expressions booléennes sur plusieurs variables. Il existe alors plusieurs combinaisons de valeurs pour lesquelles les prédicats sont vérifiés.

Exemple :



Pour franchir cette transition, il existe trois combinaisons
(a=1, b=0, c=1)
(a=0, b=1, c=1)
(a=1, b=1, c=1)

Figure 2-29.

Les échéanciers des entrées pourront comporter des couples de la forme : ([P],date) où [P] est un prédicat.

Exemple

Pour l'exemple ci-dessus on aura :
a = (X,θ)........([(a+b).c],θ_i).....
b = (X,θ)........([(a+b).c],θ_i).....
c = (X,θ)........(1,θ_i).............

III.4 - LES BOUCLES

L'exécution symbolique des boucles présente un problème plus complexe que celui des instructions conditionnelles. Nous allons introduire ce problème avec u type de boucle représentée en CADOC.LD par le graphe donné figure 2-30.

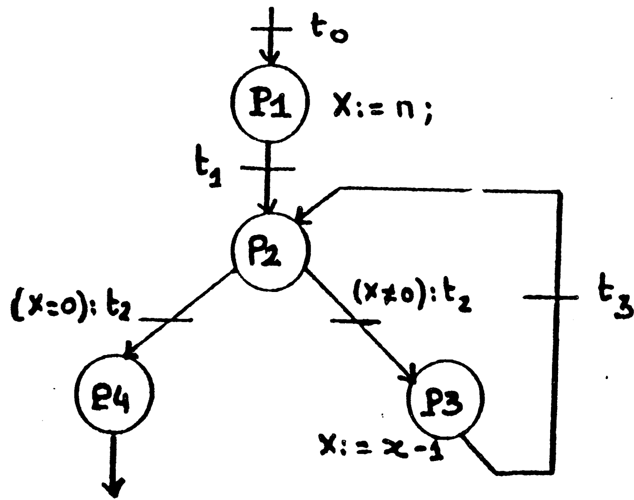


Figure 2-30.

où n peut être une constante, un paramètre ou une variable interne (définie dans la description). L'exécution symbolique temporisée de ce GIT correspond à l'arbre de chemins d'exécution suivant, paramétré par n.

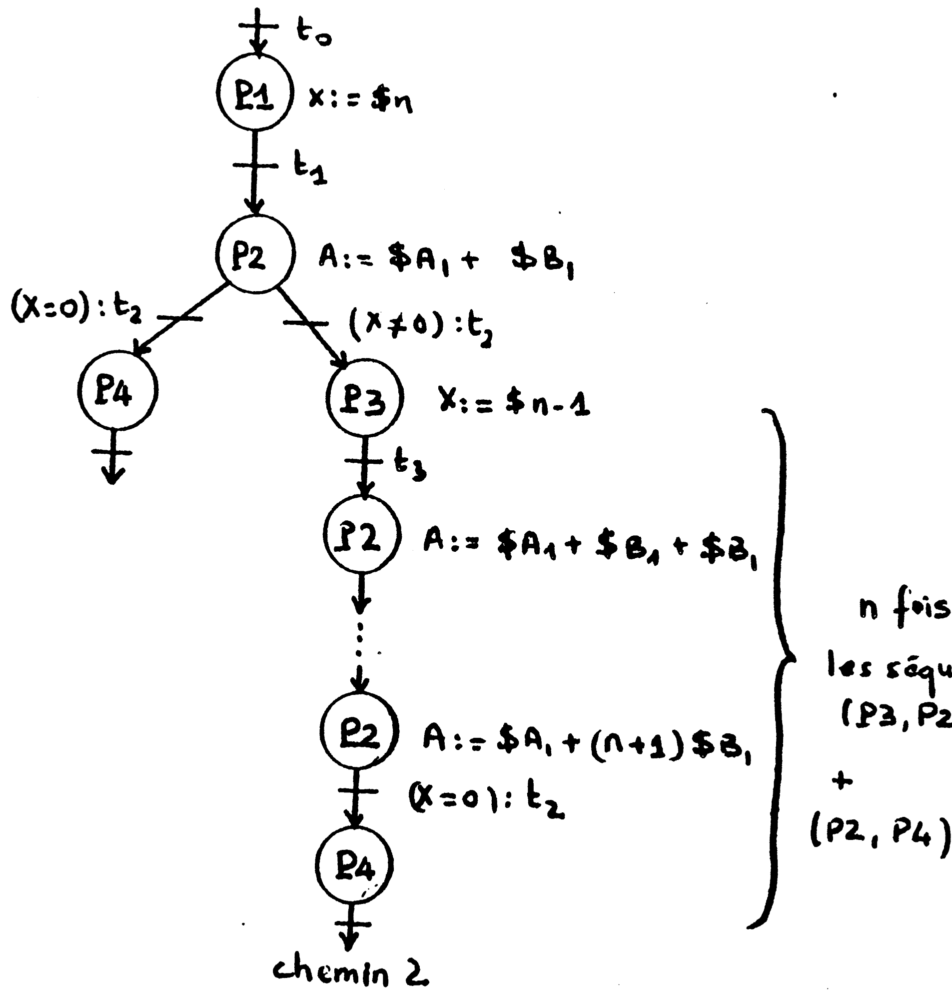


Figure 2-31.

Où $\$A_1$, $\$B_1$ sont les valeurs respectives de A et B avant l'exécution de la boucle.

<u>cas où $n est la valeur d'une constante</u>

Si n est une constante, sa valeur $n est donc définie dans la description (section "CONST"). Par conséquent, la longueur du chemin 2 est définie lors de l'exécution symbolique de la ressource générique fonctionnelle.

si $n = 0 alors le chemin 1 est choisi, et l chemin 2 ne pourra jamais être exécuté : c'est donc un chemin impossible.

si $n ≠ 0 alors le chemin 1 est impossible car on passera nécessairement dans le chemin 2 contenant au moins une boucle.

<u>cas où $n est la valeur d'un paramètre</u>

Rappelons que si n est un paramètre, sa valeur sera fixée lors de l'instantiation de la resource fonctionnelle (cf. CADOC.LD). La longueur du chemin 2 pourrait être déterminée quand la valeur de n sera connue.
Pendant l'exécution symbolique du GIT de la ressource fonctionnelle générique, la valeur de n n'est pas connue. On pourra attendre que sa valeur soit déterminée pour faire l'exécution symbolique. Mais ceci nous conduirait à refaire autant de fois la même exécution symbolique, à quelques paramètres près, que de nombres de ressources générées à partir de la même ressource générique.

Alors on fera l'exécution symbolique temporisée de la ressource générique dont les résultats (conditions de chemin temporisées, sorties symboliques attendues...) seront paramétrés par n.

L'arbre de chemins de l'exécution symbolique de la boucle de l'exemple correspond à :

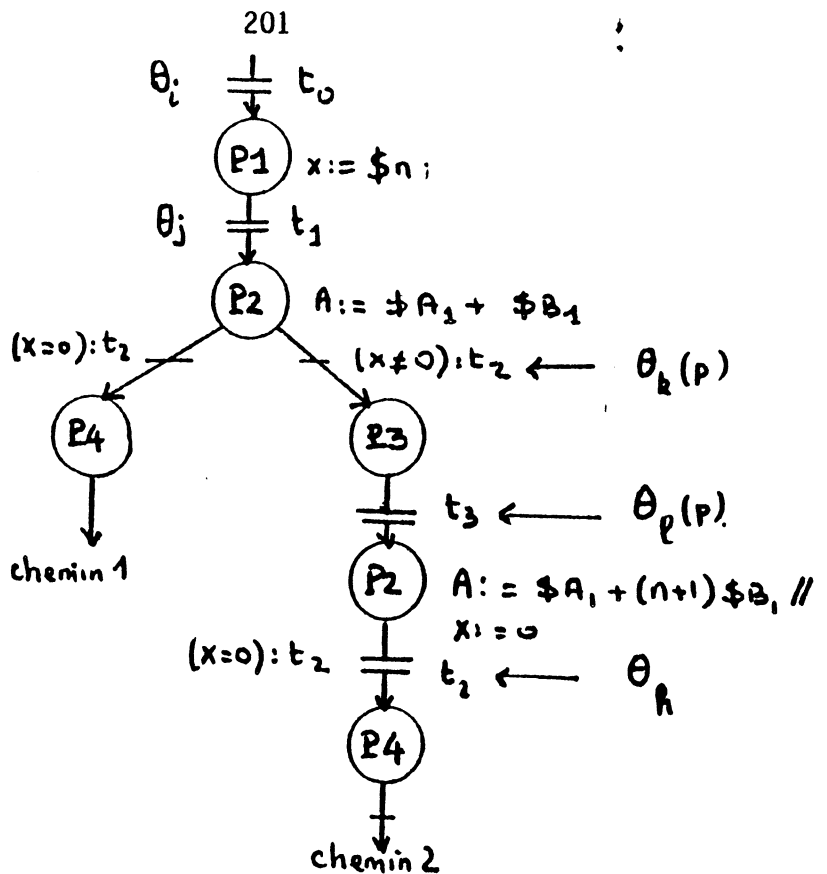


Figure 2-32.

Si $n = 0 alors le chemin sera choisi et le chemin 2 est impossible.

Si $n ≠ 0 et sachant que la boucle est de longueur fixée pour chaque ressource (fille) alors le chemin 2 sera choisi. Ce qui nous intéresse dans ce cas sont les valeurs des autres variables.
Dans cet exemple, ce sera la valeur de A; P2 étant répétée n + 1 fois dans la boucle, donc

$$A := \$A_1 + (n + 1) * \$B_1$$

<u>chemin 1</u>

<u>condition</u>:
$X := \ldots (0, \Theta_i)$
<u>sortie</u>
$A := \ldots (\$A_1, \Theta_i)(\$A_1 + \$B_1, \Theta_j) \ldots$

chemin 2

condition:

X := ... ($n , Θ_i) ...

$n   0

sortie

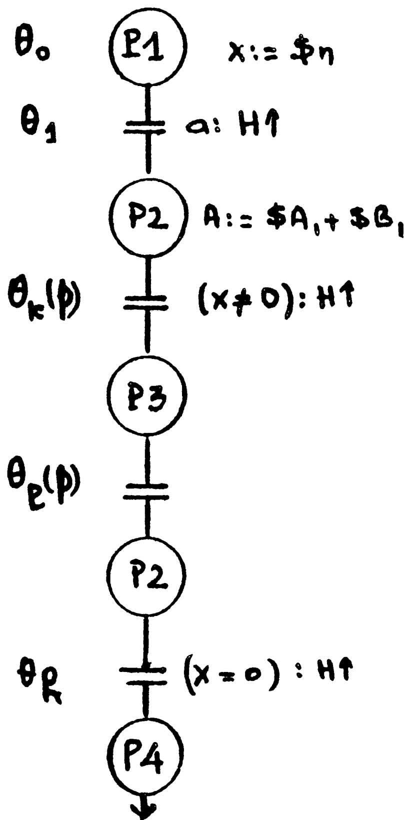
A := ... (A_i$, Θ_i)(A_1$+ B_1$, Θ_j)⌊A_1$+ (p + 1)*B_1$, $\Theta_1(p)$⌋

répecé n fois (sous l'accolade)

avec $1 \leq p \leq n$

$\Theta_1(p)$ est une date définie par rapport à $\Theta_k(p)$

Supposons que le chemin 2 est :



$\theta_k(p) = (x \neq 0) : fmx(H, 1, \theta_\ell(p-1)$

$\theta_\ell(p) = fmx(H, 1, \theta_k(p))$

$\theta_h = (x = 0) : fmx(H, 1, \theta_p(n))$

$\theta_\ell(0) = \theta_1$

Figure 2-33.

<u>cas où le nombre d'exécutions de la boucle est variable</u>

Dans notre exemple, ce serait le cas où n est une variable interne. Il faut s'assurer que la longueur de la boucle soit bornée. Dans le cas le plus général, le test utilise plusieurs variables de valeurs non bornées et que certaines d'entre elles étant modifiées dans la boucle ; ceci conduirait à un arbre d'exécution symbolique infini.

Ce problème est complexe et revient à prouver que la boucle se termine correctement. Il fait appel à des méthodes mathématiques d'induction [BLA 83].

Nous proposons alors une solution interactive où l'utilisateur doit aider le système dès qu'il a detecté une boucle. Il donnera les valeurs paramétrées des sorties et la condition de chemin temporisée paramétrée.

Exemple

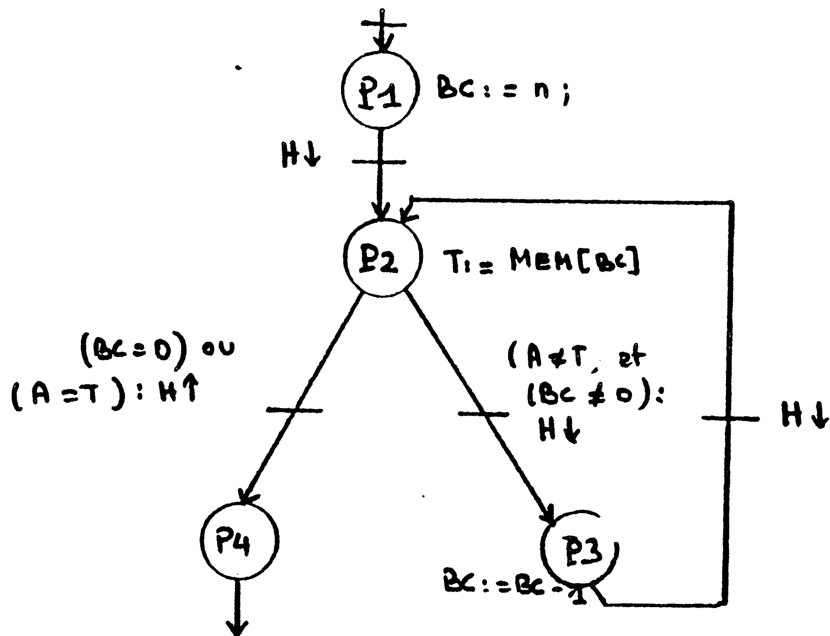


Figure 2-34.

Supposons que c'est le graphe d'un système qui recherche dans une mémoire MEM un caractère donné, mémorisé dans la variable A.
T est un registre tampon et BC un registre d'index qui peut être initialisé à une valeur quelconque m. L'arbre d'exécution symbolique correspondant est :

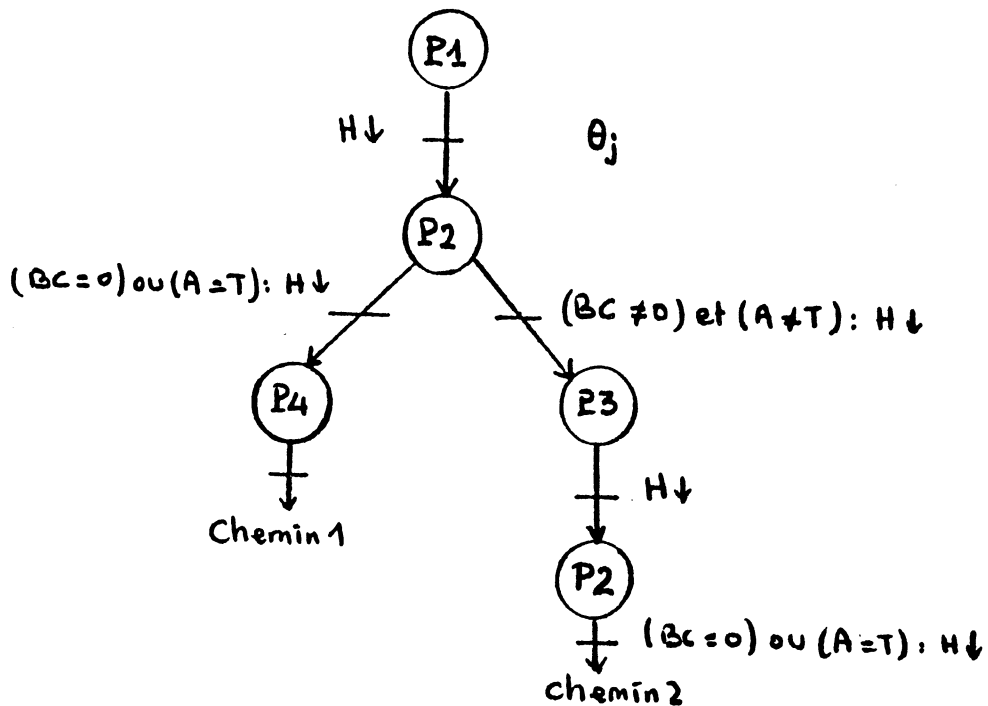


Figure 2-35.

<u>condition de chemin 2</u>

MEM [i] = ($MEM [i], $\theta_j(i)$ - e)(X, $\theta_j(i)$ + e) pour $0 \leq i < m$

et ($MEM [i] ≠ A_1$ et BC_1$ = m)

<u>ou</u>

MEM [i] = ($MEM [i], $\theta_j(i)$ - e)(X, $\theta_j(i)$ + e) pour $0 \leq i < k$

MEM [k] = ($MEM [k], $\theta_j(k)$ - e)(X,$\theta_j(k)$ + e)

avec $MEM [k] = A_1$ et $BC = k

avec $\theta_j(0) = \theta_j$

$\theta_j(i)$ = fdx (H, 2, $\theta_j(i-1)$)

Cette exemple montre bien la complexité d'une automatisation de l'exécution symbolique des boucles.

En conclusion, il existe deux solutions pour ce problème.

Première solution :

Dès que le systèmre détecte une boucle, il demandera à l'utilisateur s'il doit développer la boucle ou non. Dans le cas affirmatif, le nombre de passages dans la boucle doit être limité.

Deuxième solution :

Dès que le système détecte une boucle, l'utilisateur est prié de donner les conditions de chemins temporisées, éventuellement paramétrées, et les éhéanciers des variables internes et de sorties.

Nous penchons vers cette deuxième solution.

III.5 - PRISE EN COMPTE DE L'OPERATEUR "CHANGE"

L'opérateur "CHANGE" est un opérateur temporel qui crée un événement à chaque fois qu'une au moins des variables spécifiées dans la liste de ses opérandes change de valeur. Cet opérateur a été introduit principalement pour pouvoir modéliser les circuits "data-driven" et pour éviter de calculer à chaque unité de temps de simulation les sorties des circuits combinatoires alors que ses entrées n'ont pas changé. Dans ce dernier cas, l'opérateur CHANGE a une utilisation orientée simulation fonctionnelle event-driven.

Remarque :

L'opérateur CHANGE définit à la fois une condition sur les valeurs des ses variables (valeurs précedentes différentes de la valeur courante) et un événement correspondant à ce changement de valeur.

Pour prendre en compte cet opérateur dans les conditions de chemins temporisées, nous pourrions procéder à une transformation de l'opérateur. Cette transformation s'appuierait sur la définition sémantique de l'opérateur La figure ci-dessous présente cette transformation.

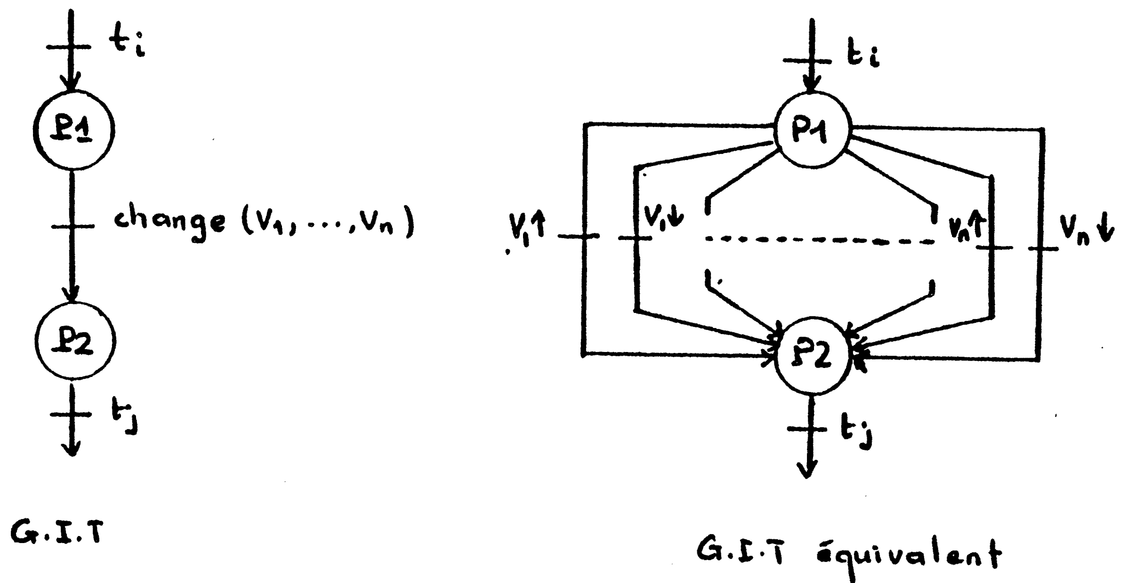


Figure 2-36.

L'implication de cette transformation dans l'expression des conditions de chemins est l'utilisation des opérateurs logiques "ET" et "OU". Les conditions de chemin seront exprimées de la manière suivante :

$cond_1$ et ($cond'_1$ ou $cond'_2$ ou ... ou $cond'_{2n}$) et $cond_2$

Les "$cond'_i$" correspondent aux conditions de passage dans les arcs respectifs résultants d'une transformation de l'arc initial avec une receptivité utilisant "CHANGE" (cf. figure ci-dessus)

Mais cette notation n'est pas homogène avec celles que nous avons déjà introduites. En fait, on peut considérer qu'une expression utilisant l'opérateur CHANGE est un prédicat complexe. Dans ce cas, nous utiliserons la notation déjà introduite (cf. III.2).
Les variables opérandes de CHANGE auront des échéanciers de la forme suivante :

$$V_i = (X, \Theta_0) \ldots (\$V_i, \Theta_j)([change(V_1, V_2, \ldots, V_n], \Theta_k) \ldots$$

D'après la remarque que nous avons faite au début de ce paragraphe, la définition de l'instant de référence Θ_k est très liée au prédicat complexe $\lfloor change(V_1, \ldots, V_n) \rfloor$.
Le prédicat $\lfloor change(V_1, \ldots, V_n) \rfloor$ est vrai si et seulement si il existe au moins une des variables $V_1, \ldots, V_n$ qui change de valeur. Donc pour activer le chemin comportant un arc franchissable sur "CHANGE", il suffit :

- de choisir une variable, au moins, parmi les n variables dans "CHANGE",
- de choisir la valeur de Θ_k, l'instant de changement de cette variable.

La définition de Θ_k est :

$$\Theta_k = change\ (V_1, \ldots, V_n) : e(\Theta_{k-1})$$

où e( ) est l' opérateur défini au pragraphe II.2.

IV - ECHEANCIERS SYMBOLIQUES DES SORTIES

Les échéanciers des variables de sortie sont mis à jour pendant l'exécution symbolique des actions associées au noeud courant du chemin. Les actions définies dans le langage CADOC.LD peuvent être simples ou complexes. Nous imposons dans notre étude que les actions soient simples pour une raison de complexité de l'exécution symbolique.

Une action etant définie comme simple si c'est une affectation de chronogrammes dont chaque élément (valeur,date) est simple : la valeur peut être une valeur constante, une expression arithmétique ou logique, et la date peut être une valeur constante ou un événement simple à un délai près.

Exemple

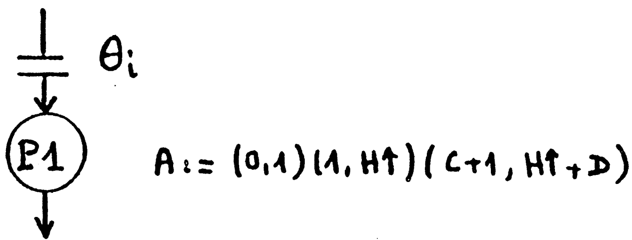


Figure 2-37.

Dans ce cas, l'exécution symbolique de cette action consiste à ajouter ce bout de chronogrammes à son échéancier déjà établi, en modifiant les dates (cf. simulation).

Soient avant P1 les échéanciers de A et de C :

$C = (X, \Theta_0)(\$C_1, \Theta_1)....(\$C_{i-1}, \Theta_{i-1})$

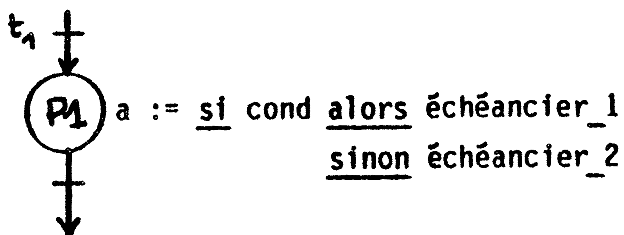
$A = (X, \Theta_0)(0, \Theta_1)....(1, \Theta_{i-1})$

après P1, on a :

$A = (X, \Theta_0)(0, \Theta_1)....(1, \Theta_{i-1})(0, \Theta_i + 1)(1, fmx (H, 1, \Theta_i + 1))$
$(\$C_{i-1} + 1, fmx (H, 2, \Theta_I + 1) + D)$

cas d'une action complexe

Nous voulons seulement montrer la complexité de l'exécution symbolique d'une action complexe du langage CADOC.LD. Prenons par exemple l'action "affectation conditionnelle" de la forme suivante :



Si la partie "condition" n'est pas vérifiée par la condition de chemin temporisée et par les états des variables internes avant l'exécution de l'action alors il faut définir une condition de chemin au niveau des actions ; ce qui rend plus complexe l'exécution symbolique. Dans ce cas, il faut utiliser la description suivante :

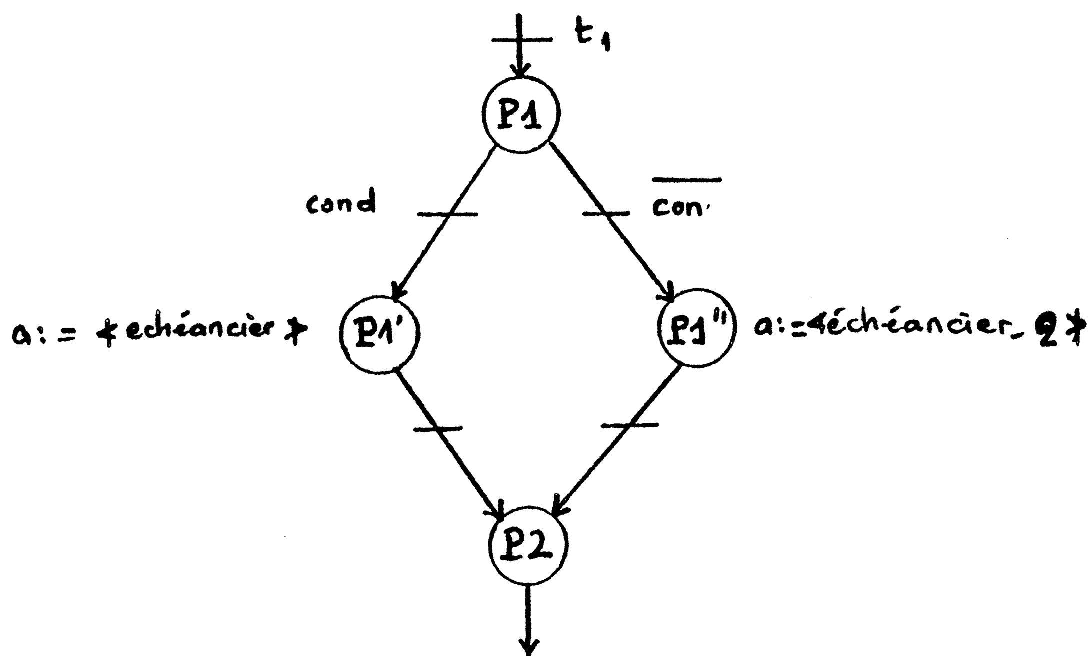


Figure 2-38.

Pour l'action "MUX" de la forme suivante :

a := <u>mux</u> v1 v2 ... vn <u>dans</u>

$c_{01} \cdots c_{0n}$: échéancier_0 ;

$c_{m1} \cdots c_{mn}$: échéancier_m ;

il faut utiliser la description donnée figure 2-39.

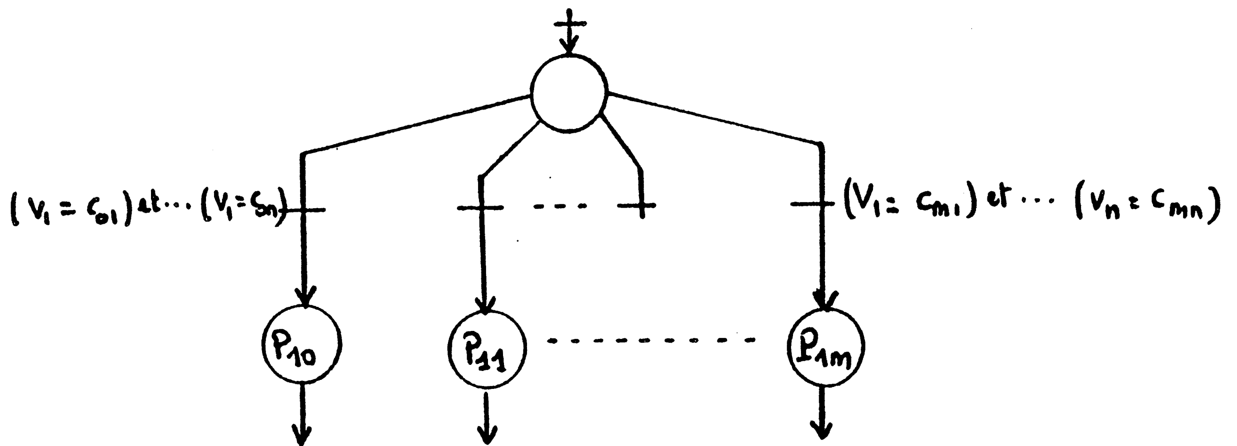


Figure 2-39.

Nous considérons que les "échéanciers_i" sont des échéanciers simples.

V - CONCLUSION

L'outil d'exécution symbolique temporisée que nous avons proposé est un enrichissement de celui défini dans [BEL 84]. Il permet de faire l'exécution symbolique d'une description de tout type de circuits. L'exécution symbolique d'un GIT permet d'obtenir :

- les conditions de chemins temporisées du GIT
- les échéanciers symboliques des sorties attendues.

Elles sont obtenues par le concours de plusieurs tâches dont les principales sont :

- temporisation de chemins définissant les instants de référence.
- définition des conditions des chemins temporisées.
- exécution symbolique des actions déterminant les échéanciers de sorties attendues et des variables internes.

Pour les graphes plus complexes, on appellera une tâche permettant de décomposer le graphe en sous-graphes. On fera l'exécution symbolique des sous-graphes ; à partir de l'arbre des sous-graphes et par opération de composition de chemins, on pourra établir les conditions de chemins temporisées des graphes complexes. Cette t he a été spécifiée dans [BEL 84].

Enfin, notons que cet outil est utilisé pour générer automatiquement les données nécessaires pour la génération fonctionnelle de test qui est présentée au chapitre suivant.

CHAPITRE IV : GENERATION FONCTIONNELLE DE TEST

Les méthodes algorithmiques classiques pour la génération de test des circuits sont appliquées soit pour des circuits de faible taille, soit pour des circuits respectant certaines contraintes de conception (circuits testables). Face aux problèmes de test des circuits intégrés complexes (VLSI), ces méthodes deviennent inefficaces.

Nous proposons dans ce chapitre une méthode utilisant :

- le langage CADOC.LD comme langage de spécification fonctionnelle des circuits (cf. section I, chapitre III)
- l'exécution symbolique temporisée (cf.section II, chapitre III)
- les méthodes de l'intelligence artificielle.

L'intelligence artificielle est utilisée actuellement dans le domaine du diagnostic des circuits intégrés. Ces applications sont plutôt orientées "Système Expert". Dans notre approche, nous faisons appel à l'intelligence artificielle pour résoudre les problèmes de propagation et de consistence de la génération de test. Notre stratégie générale de génération de test est présentée au paragraphe II.2 ; elle fait appel à la construction d'un ensemble de connaissances sur des ressources (paragraphe II) et à des synthèses de connaissances (paragraphe III) à partir desquelles sont déterminés un ensemble d'activations fonctionnelles pour le test du circuit.

I - STRATEGIE DE TEST

Définition

Nous appellerons activation fonctionnelle d'une ressource fonctionnelle l'ensemble de séquences de ses entrées permettant d'activer un chemin donné de la ressource.

Stratégie

Le but principal de notre méthode est de générer un ensemble d'activations fonctionnelles à partir desquelles peut être généré un programme de test pour le circuit étudié.

Nous utilisons comme langage de spécification du circuit le langage CADOC.LD : le circuit à étudier en vue de son test est décrit comme un assemblage de plusieurs ressources fonctionnelles.

D'une manière globale, notre approche consiste à connaître d'abord comment on peut activer une ressource donnée pour accomplir son test. Si cette ressource est une partie d'un circuit, nous chercherons à déterminer comment on peut activer le circuit global et observer les résultats de test de la ressource. La stratégie est la suivante :

- description partitionnée hiérarchisée ou comportementale du circuit en CADOC.LD.
- étude de chaque ressource : extraction des connaissances sur les possibilités d'activer la ressource dans le but d'accomplir son test.
- étude globale du circuit : spécifications des activations fonctionnelles primaires (c.a.d au niveau des entrées primaires) à partir des connaissances sur chacune des ressources constituant le circuit. Cette étape n'existera pas pour une ressource décrite comportementalement (au sens CADOC.LD).
- génération de programme de test à partir des spécifications d'activations fonctionnelles primaires et ,éventuellement, des vecteurs de test préalablement déterminés pour la ressource à tester.

La figure ci-dessous montre notre stratégie.

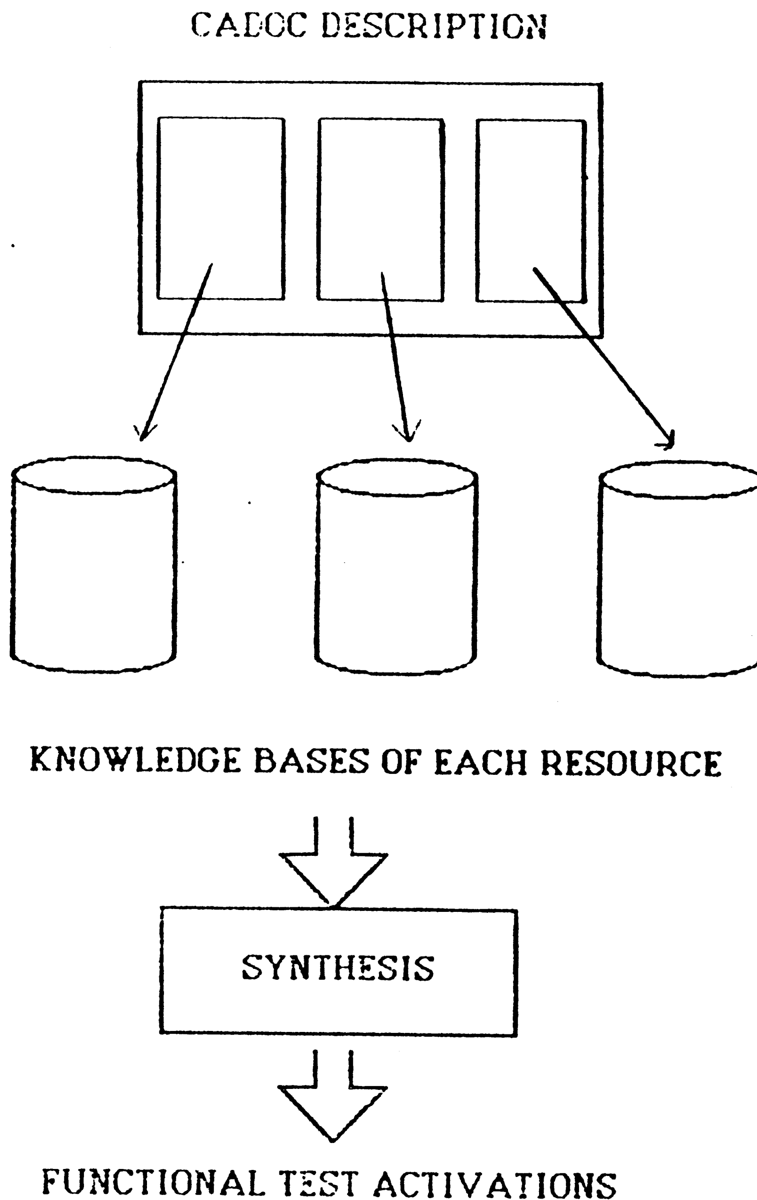


Figure 2-44.

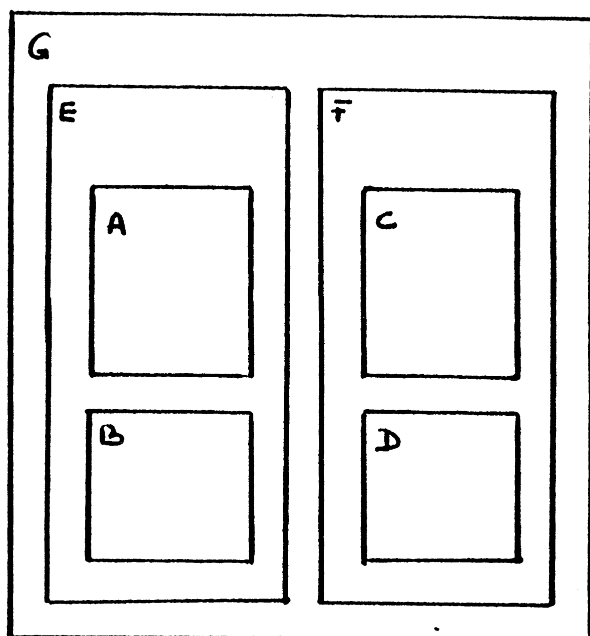
Quelques avantages de cette approche sont donnés ci-dessous. Nous préciserons les autres avantages ainsi que les problèmes au fur-et-à-mesure de notre présentation.

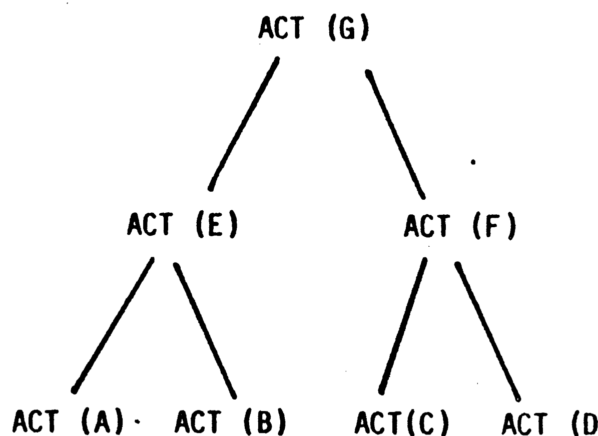
- Cette approche permet d'étudier le test d'un circuit dès ses premières phases de conception : la description fonctionnelle du circuit en CADOC.LD est disponible avant la réalisation du circuit. Ceci n'est pas le cas pour les méthodes classiques telle le D-algorithme où il faut attendre la réalisation logique du circuit pour pouvoir étudier son test. De toute façon, l'étude du test des circuits VLSI n'est pas envisageable au niveau logique.

- Elle est souple car l'utilisateur n'est pas contraint à faire un partitionnement en ressources fonctionnelles satisfaisant à un certain nombre de contraintes ; autrement dit, l'utilisateur est maître de la précision du test qu'il désire. Par exemple, il pourra faire du test purement comportemental lorsqu'il n'est pas en possession de la description de la structure interne réelle du circuit auquel cas il se contentera d'activer les principales fonctions du circuit. Il pourra aussi décrire le circuit partitionné en ressources fonctionnelles pour lesquelles il veut étudier le test individuellement. Connaissant certains vecteurs de test ou certaines séquences de test pour une activation locale donnée d'une ressource quelconque, l'utilisateur obtiendra de notre outil toutes les activations fonctionnelles au niveau des entrées primaires du circuit pour envoyer ces vecteurs de test ou générer ces séquences de test aux entrées de la ressource sous test, d'une part, et d'autre part d'observer les résultats de test aux sorties primaires du circuit.

- Elle est récursive et complètement adaptée à une description partitionnée hiérarchisée de CADOC.LD. En effet, on pourra d'abord étudier le test d'une partie du circuit qui est elle-même composée de plusieurs sous parties. Ensuite, connaissant l'ensemble des activations onctionnelles définies pour cette partie et des autres parties du circuit de même niveau hiérarchique, on pourra en déduire un autre ensemble d'activations forctionnelles. Le processus peut être appliqué de nouveau pour toutes les parties de niveau

hiérarchique supérieur (cf. Figure 2-45).





où ACT (X) sont les activations fonctionnelles de la ressource X.

Figure 2-45.

L'originalité de notre méthode est l'utlisation des techniques qui sont l'exécution symbolique temporisée et de l'intelligence artificielle. L'exécution symbolique temporisée a été étudiée au chapitre III. Celle-ci nous permet d'établir les spécifications des activations fonctionnelles d'une ressource. Les conditions de chemins temporisées et les résultats attendus des sorties constituent les principaux éléments de ces spécifications qui sont considérées comme des connaissances sur la ressource, représentées sous forme de règles ; les activations fonctionnelles de la ressource sont déterminées à partir de ces connaissances. L'intelligence artificielle est utilisée pour déduire un ensemble de spécifications d'activations fonctionnelles primaires considérées comme une synthèse de connaissances dans le but de tester les ressources constituantes d'une ressource globale. Autrement dit, les phases de propagation et de consistence de la génération de tests ou une ressource constituante donnée sont faites en utilisant les techniques d'intelligence artificielle, en considérant particulièrement le problème temporel.
D'autre part, cette approche étant basée sur une description fonctionnelle en CADOC.LD, elle permet de préparer le test du circuit avant la réalisation finale de celui-ci.

II - FICHIER DE CONNAISSANCES D'UNE RESSOURCE

Dans ce paragraphe, nous considérerons une ressource fonctionnelle élémentaire, c'est-à-dire, sans ressource constituante. Nous remarquons ici qu'une ressource peut être aussi bien un circuit très complexe décrit comportementalement qu'un élément matériel fonctionnel primitif tel bascules .... Remarquons aussi qu'il ne sera pas nécessaire de considérer individuellement toutes les ressources fonctionnelles issues de la même ressource fonctionnelle générique. Même si la RGF est paramétrée, il suffit de construire le fichier de connaissances de celle-ci en paramétrant les connaissances. Les fichiers de connaissances des ressources fonctionnelles "filles" seront générées à partir du fichier de connaissances de la RGF. Cette phase est analogue à la phase d'édition de lien du langage CADOC.LD [TIA 85]. Par la suite, nous utiliserons donc le mot "ressource" pour désigner ressource fonctionnelle générique.

II.1 - LES CONNAISSANCES

La connaissance du fonctionnement d'un circuit ou de ses parties composantes est nécessaire pour pouvoir étudier le test du circuit : c'est la première chose qu'un ingénieur de test des circuits complexes fait pour générer le programme de test du circuit. Cette étude ne peut être faite qu'à partir d'une spécification du circuit. En ce qui concerne les circuits intégrés complexes, il est préférable que cette spécification soit au niveau fonctionnelle car cela permet de faire abstraction de la réalisation du circuit et de gagner du temps pour l'étude du fonctionnement du circuit; L'extraction du fonctionnement des parties du circuit complexe à partir des spécifications logiques serait fastidieuse et surtout nécessiterait beaucoup d'heures de travail.

Nous proposons ici une automatisation de la construction du fichier de connaissances sur le fonctionnement d'une ressource. Un circuit ou une partie d'un circuit étant considéré comme une ressource fonctionnelle dont la RGF est décrite en CADOC.LD. A partir du GIT de la description de la ressource, on peut extraire des connaissances. En fait, le GIT constitue déjà une connaissance de la ressource, mais celle-ci se trouve au niveau "évolution

des phases", et par conséquent très orientée vers la simulation fonctionnelle. Ce qui nous intéresse est plutôt une connaissance plus compacte, c'est-à-dire, une connaissance qui lie une certaine séquence d'entrées à une séquence de sorties.

Une connaissance d'une ressource met donc en relation des activations fonctionnelles de la ressource et des réponses attendues. Les activatons fonctionnelles seront définies par les conditions de chemins temporisées des chemins du GIT et les réponses attendues seront définies par les échéanciers symboliques des variables de sorties et des variables internes. L'outil utilisé pour l'extraction de ces connaissances est l'exécution symbolique temporisée que nous avons présentée au chapitre III de cette section. L'exécution symbolique temporisée de la description d'une ressource nous permet de compacter les connaissances représentées initialement par le GIT. Les éléments constituant une connaissance sont :

- une condition de chemin temporisée
- un ensemble d'échéanciers symboliques des variables (sorties, bidirectionnelles et internes).

Le fichier de connaissances d'une ressource est l'ensemble de toutes les connaissances extraites de la spécification fonctionnelle de la ressource.

II.2 - REPRESENTATION DES CONNAISSANCES

Nous trouvons que la représentation des connaissances sous la forme de règles est satisfaisante pour notre problème :

SI <faits> ALORS <actions>

Cette forme est généralement utilisée dans les systèmes à base de règles comme par exemple les Systèmes Experts [LAU 82]. La signification est :
si l'ensemble de faits <faits> existe alors un ensemble d'actions est initialisé.

L'étude du test au niveau fonctionnel n'est intéressant que si celle-ci permet de déterminer d'une manière précise les instants auxquels les données de test doivent être appliquées aux entrées du circuit. Nous devons alors introduire et utiliser la dimension "temps" dans les règles. A notre connaissance, cette approche n'est pas encore utilisée dans le domaine des applications de l'Intelligence Artificielle. Ce qui explique aussi le quasi inexistance de référence bibliographique le long de ce chapitre.
Par conséquant, nous avons étendu cette représentation à une représentation temporisée. La forme d'une règle sera la suivante :

AVEC
<définition-instants-de-référence>
SI
<condition-de-chemin-temporisée>
ALORS
<échéanciers-attendus>

où <définition-instants-de-référence> est constitué d'un ensemble de définitions des instants de références utilisés dans les conditions de chemins temporisées et les échéanciers.

Cette partie est composée de trois sous-parties :
- instant initial θ_0
- les instants θ_i
- l'instant final θ_f, déterminant la fin du chemin correspondant à la règle.

Exemple

Soit un circuit C dont l'interface (ports d'E/S) est décrite dans la figure 2-46a et dont le comportement est décrit par le GIT donné en figure 2-46b. L'arbre des chemins de l'exécution symbolique du GIT est donné figure 2-46c.

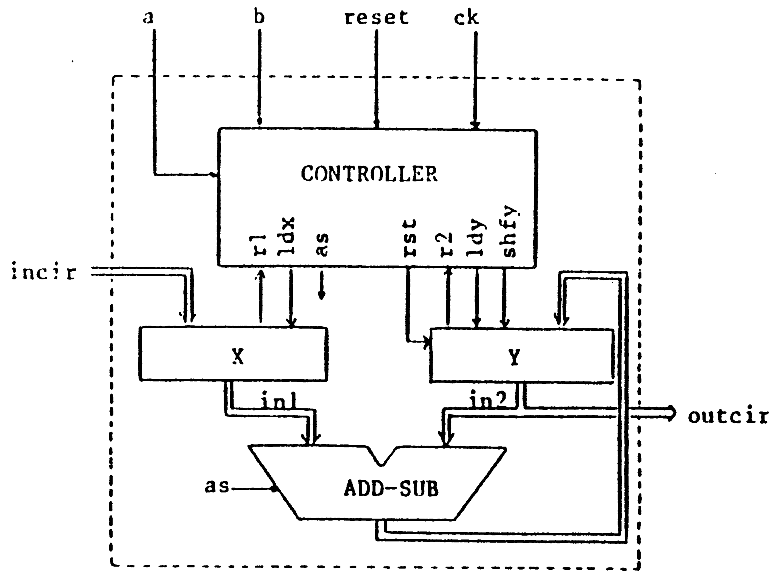


Figure 2-46a.

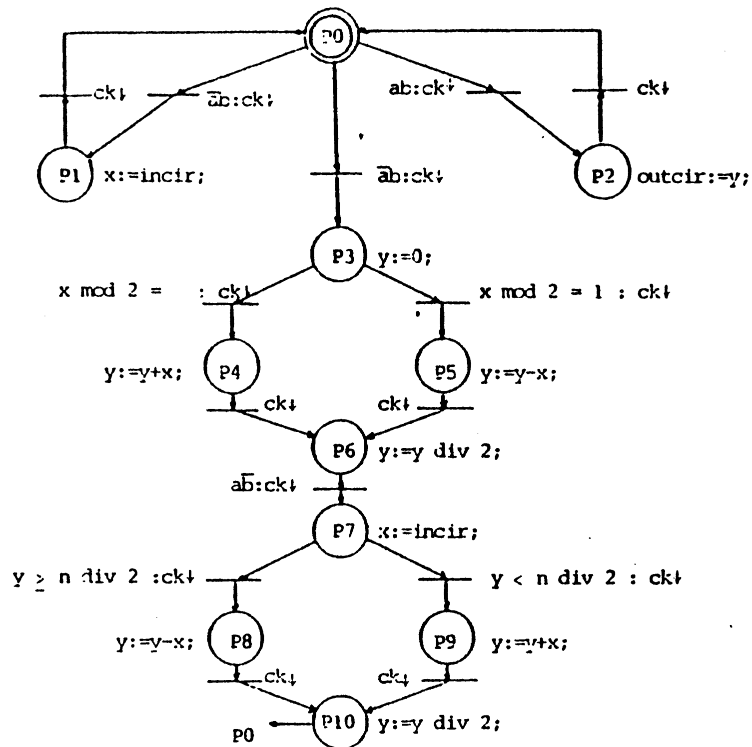


Figure 2-46b

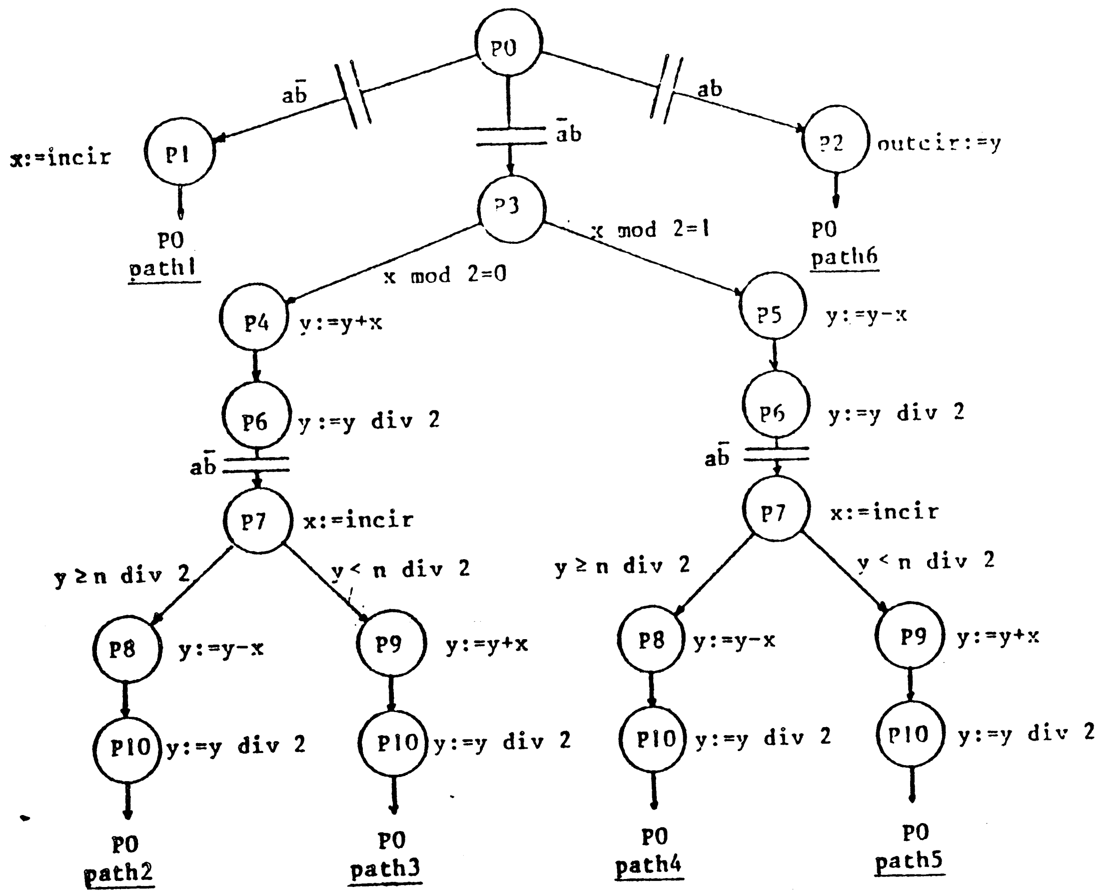


Figure 2-46c.

A chacun des chemins du GIT sera associé une règle :

<u>chemin 1</u> :

<u>AVEC</u>

<u>INSTANT-INITIAL</u> : $\%^{1}_{0}$

$\%^{1}_{1}$ = a et non b : fdx(H,1,$\%^{1}_{0}$)

<u>INSTANT-FINAL</u> : $\%^{1}_{2}$ = fdx(H,3,$\%^{1}_{1}$)

<u>SI</u>

a = (X, $\%^{1}_{0}$)(1, $\%^{1}_{1} - s_a$)(X, $\%^{1}_{1} + s'_a$)
b = (X, $\%^{1}_{0}$)(0, $\%^{1}_{1} - s_b$)(X, $\%^{1}_{1} + s'_b$)
incir = (X, $\%^{1}_{0}$)(%incir, $\%^{1}_{1} - s_{in}$)(X, $\%^{1}_{1} + s'_{in}$)

<u>ALORS</u>

x = (%incir, $\%^{1}_{1}$)

<u>FIN</u>

<u>chemin 2</u> :

<u>AVEC</u>

<u>INSTANT-INITIAL</u> : $\%^{2}_{0}$
$\%^{2}_{1}$ = non a et b : fdx(H,1,$\%^{2}_{2}$)
$\%^{2}_{2}$ = a et non b : fdx(h,3,$\%_{1}$)
<u>INSTANT-FINAL</u> : $\%^{2}_{3}$ = fdx(H,4,$\%^{2}_{2}$)

<u>SI</u>

a = (X, $\%^{2}_{0}$)(0, $\%^{2}_{1} - s_a$)(X, $\%^{2}_{1} + s'_a$)(1, $\%A^{2}_{2} - s_a$)(X, $\%^{2}_{2} + s'_a$)
b = (X, $\%^{2}_{0}$)(1, $\%^{2}_{1} - s_b$)(X, $\%^{2}_{1} + s'_b$)(0, $\%^{2}_{2} - s_b$)(X, $\%^{2}_{2} + s'_b$)
incir = (X, $\%^{2}_{0}$)(%incir, $\%^{2}_{2} - s_{in}$)(X, $\%^{2}_{2} + s'_{in}$)
val(x, fdx(H,1,$\%^{2}_{1}$)))d 2 = 0
(%incir div 2) n div 2

<u>ALORS</u>

y := (0, $\%^{2}_{1}$)(val(x, fdx(H,1,$\%^{2}_{1}$)), fdx(H,1,$\%^{2}_{1}$))

(val(x, fdx(H,1,$\%^{2}_{1}$)) div 2, fdx(H,2,$\%^{2}_{1}$))

$(val(x,\ fdx(H,1,\%\Theta_1^2))\ div\ 2\ -\ \%incir,\ fdx(H,1;,\%\Theta_2^2))$
$((val(x,\ fdx(H,1,\%\Theta_1^2)\ div\ 2\ -\ \%incir)\ div\ 2,\ fdx(H,2,\%\Theta_2^2))$

$x := (\%incir,\ \%\Theta_2^2)$

FIN

Nous ne donnerons pas ici les règles correspondant aux chemins 3,4,5,6. Remarquons seulement que ces chemins (sauf le chemin 6) peuvent partager la même portion de chemin de l'arbre. Par exemple, le chemin 3 partage la portion de chemin (P0, P3, P4, P6, P7) avec le chemin 2 :

les définitions des instants de référence $\%\Theta_0^3$, $\%\Theta_1^3$, $\%\Theta_2^3$ du chemin 3 sont semblables aux définitions des instants de référence $\%\Theta_0^2$, $\%\Theta_1^2$, $\%\Theta_2^3$ du chemin 2. On pourrait donc définir $\%\Theta_0^3$, $\%\Theta_1^3$, $\%\Theta_2^3$ par référence externe à $\%\Theta_0^2$, $\%\Theta_1^2$, $\%\Theta_2^2$ définis dans la règle n° 2. Nous pourrions introduire dans ce langage le mécanisme de références externes utilisés dans les programmes.

Ceci nous conduit à poser la question suivante : quelle est donc l'autonomie d'une règle ? Pour répondre à cette question, nous étudierons au paragraphe suivants les visibilités des objets utilisés dans une règle.

II.3 - VISIBILITE DES OBJETS DEFINIS DANS UNE REGLE

Les règles étant la représentation des connaissances sur le fonctionnement d'une ressource, il est préférable qu'on puisse les définir dans un ordre quelconque indépendamment des manipulations sur celles-ci : aspect non-proceduriel du langage (cf. PROLOG). Un des avantages de cette solution est :

- supposons que dans un premier temps, on n'ait pas encore l'outil d'exécution symbolique temporisée pour la construction automatique du fichier

de connaissances. L'utilisateur doit donc le définir en utilisant un langage du même type que celui défini au paragraphe III.2. L'aspect non procédural permet à l'utilisateur de facilement ajouter ou modifier ce fichier sans se soucier de le redéfinir complètement.

Exemple

- dans les deux règles que nous avons définies au paragraphe III.2, nous avons utilisé deux objets de même nom "%incir" représentant un objet symbolique. Ces deux objets n'ont aucun lien. Par contre, dans une même règle, ils font référence au même objet. Par exemple, dans la règle 1 (chemin 1) le "%incir" défini dans l'échéancier de la variable d'entrée "incir" est le même que celui affecté à la variable "x".

- dans la définition des instants de référence d'une règle nous avons choisi des noms différents pour chaque règle : $\%\theta_i^1$, $\%\theta_j^2$ respectivement pour le chemin 1 et le chemin 2. En fait, on pourait prendre les mêmes identificateurs pour les deux règles.

En ce qui concerne les objets comme les variables a, b, incir, x, y et les constantes s_a, s'_a, s_b, s'_b, ils font référence aux mêmes objets ; ces objets ne sont pas définis dans la règle, ils sont définis dans la spécification de la ressource en CADOC.LD.

Il existe donc deux classes d'objets de visiblité différente :
- les objets définis dans la spécification d'une ressource sont partagés par toutes les règles définies pour la ressource.
- les objets définis par l'exécution symbolique des chemins du GIT d'une ressource sont visibles localement dans une règle.

II.4 - EVALUATION DE LA PARTIE "CONDITION"

La partie "condition" d'une règle est constituée d'égalités et d'expres-

sions booléennes classiques (=, <, >, $\leq$, $\geq$, $\neq$). Nous avons remarqué au chapitre III que les échéanciers dans les conditions de chemin n'expriment pas l'évolution réelle des variables correspondants mais un "squelette" des chronogrammes que les variables en question peuvent avoir.

Exemple

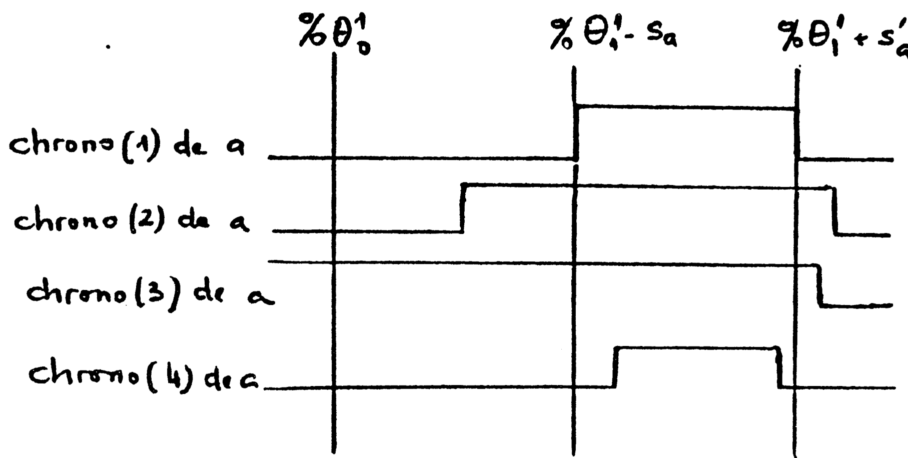


Figure 2-47.

Les trois premiers chronogrammes vérifient l'égalité suivante :

$a = (X, \%\theta_0^1)(1, \%\theta_1^1 - s_a)(X, \%\theta_1^1 + s'_a)$

Par contre, le chronogramme 4 ne vérifie pas celle-ci parce que 'a' doit être à 1 à la date spécifiée par $(\%\theta_1^1 - s_a)$.

Soit l'égalité suivante :

$var = (v_0, t_0)(v_1, t_1) \; .... \; (v_i, t_i)(v_{i+1} \; t_{i+1}) \; .... \; (v_n, t_n)$

Celle-ci est vérifiée si et seulement si :

<u>stable</u> (<u>val</u>(var, $t_{i+1} - t_i$)) <u>et</u> (<u>val</u>(var, t_i) = v_i) <u>pour</u> $0 < i < n - 1$

ou

$\underline{val}(var, t_n) = v_n$;

Rappelons que si v_i = X (don't-care), alors $\underline{val}(var, t_i) = v_i$ est toujours VRAI.

La partie "condition" est vérifiée si toutes les expressions définies dans cette partie sont vérifiées.

II.5 - CONSTRUCTION DES ECHEANCIERS GLOBAUX D'UNE VARIABLE

La partie "action" d'une règle est uniquement constituée d'une liste d'affectations d'échéanciers. L'affectation d'un échéancier à une variable consiste à concaténer l'échéancier donné à l'échéancier global de la variable. Lors de cette concaténation, on doit assurer la cohérence de l'échéancier global, en particulier pour les dates. Les instants définis dans la règle sont en fait des instants non fixés mais peuvent avoir plusieurs valeurs.
L'opération d'affectation ne peut pas être faite tant que les valeurs de tous les objets définis dans la règle ne sont pas fixées ; ces valeurs sont des valeurs symboliques.
Cette opération est analogue à celle effectuée en cours de simulation, sauf qu'en simulation on manipule des valeurs numériques.

Ceci nous amène à définir les variables d'une règle au sens de la logique des prédicats.

II.6 - LES VARIABLES D'UNE REGLE

Il est évident que certains objets utilisés dans une règle ne doivent pas avoir une valeur fixée : ces objets sont classifiés dans la catégorie des variables définies au niveau d'une règle. On peut se référer au langage PROLOG. Comme dans ce langage, nous avons lexicalement précédé du symbole "%" toutes les variables d'une règle qui sont :

- les variables pouvant recevoir une valeur symbolique d'instant de référence
- les variables pouvant recevoir une valeur symbolique. Nous

expliquerons plus en détail l'utilisation de ce type de variables dans le paragraphe III.1.4.

Exemple

$\%t_i$: variable pouvant prendre la valeur symbolique Θ_i

%incir : variable pouvant prendre la valeur symbolique $incir.

Lors de l'utilisation de ces règles, les valeurs de ces variables doivent être déterminées et fixées symboliquement. Cette opération est appélée "unification" (cf. paragraphe IV.1).

II.7 - EXECUTION SYMBOLIQUE ET REGLES

Après l'introduction de ces notions, nous parlerons dans ce paragraphe du lien exact entre les résultats d'exécution symbolique et la définition des règles. Nous avons remarqué plus haut que le langage de définition de règles (introduit informellement) peut constituer un langage d'entrée des règles dans un premier temps. Dans ce paragraphe, nous présenterons les principaux points de la génération automatique d'une règle à partir des résultats d'exécution symbolique de la description d'une ressource en CADOC.LD.

L'exécution symbolique d'une description nous donne tous les éléments de connaissances sur la ressource décrite. Il nous reste donc à le mettre sous la forme de règles. Les valeurs symboliques (Θ_i et les valeurs précédées de $) sont considérées comme fixées pour l'exécution symbolique. Compte tenu des notions que nous avons introduites, nous devons remplacer ces objets par des variables precédées cette fois-ci de %.
Ensuite, la mise sous forme de règles des connaissances est faite (cf. exemple page 220).

Les variables correspondant aux instants de référence définis pendant l'exécution symbolique d'un chemin constitueront la partie 'AVEC' de la règle temporisée.

Exemple

Les instants de référence du chemin 1 de la figure (c) ,page 9 sont :

instant-initial : Θ_0^1

Θ_1^1 = a <u>et</u> <u>non</u> b : fdx(H,1,Θ_0^1)

fin à Θ_2^1 = fdx(H,1, Θ_1^1)

La partie "temporisation" de la règle correspondante est :

<u>AVEC</u>

<u>INSTANT INITIAL</u> : $\%\Theta_0^1$

$\%\Theta_1^1$ = a <u>et</u> <u>non</u> b : <u>fdx</u>(H,1,$\%\Theta_0^1$)

<u>INSTANT FINAL</u> : $\%\Theta_2$ = <u>fdx</u>(H,1,$\%\Theta_1$)

La partie "condition" d'une règle est construite à partir des échéanciers et des expressions booléennes constituant les conditions de chemin temporisées. La construction est évidente : les instants de référence utilisés dans les échéanciers et les expressions booléennes sont remplacés par les variables "instants" correspondants, définis dans la partie "temporisation".

La partie "action" est construite à partir des échéanciers de sortie et de la même manière que pour la construction de la partie gauche.

II.8 - CONCLUSION

Dans ce paragraphe III, nous avons présenté la notion de base de notre

approche qui est la notion de "connaissance". Pour chaque ressource élémentaire constituant un circuit complexe, des connaissances compactes et précises sur leur fonctionnement sont définies. Elles sont représentées sous forme de règles temporisées. Ces règles constituent un "fichier de connaissances" de chaque ressource. Notons que ce fichier est défini précisément pour une ressource générique fonctionnelle quelconque. Les fichiers de connaissances des exemplaires d'une ressource générique fonctionnelle seront générés à partir du fichier de la RGF mère.

Nous avons montré que la construction automatique du fichier de connaissances d'une ressource quelconque peut être faite à partir des résultats d'exécution symbolique de sa description fonctionnelle en CADOC.LD.

III - RECHERCHE DES ACTIVATIONS FONCTIONNELLES GLOBALES

Dans ce paragraphe, le problème qui est considéré est le suivant :

- étant donné un circuit complexe composé d'un ensemble de ressources fonctionnelles interconnectées, on cherche à déterminer les activations fonctionnelles au niveau des entrées primaires pour tester une ressource donnée et les séquences de sorties attendues aux sorties primaires.

Si nous considérons un circuit combinatoire décrit comme une interconnexion de portes logiques, la résolution de notre problème est similaire aux méthodes de sensibilisation de chemin (par exemple le D-algorithme).

Notre idée est de définir une telle méthode au niveau fonctionnel. Nous avons présenté dans le

paragraphe précédent le type de connaissances dont nous avons besoin; ces connaissances sont compactes et précises : elles nous permettent de déduire les valeurs des sorties d'une ressource par rapport à celles de ses entrées. Plus précisement, notre problème est donc :

- étant donnée une ressource fonctionnelle constituante d'un circuit, et ayant des connaissances fonctionnelles de toutes les ressources du circuit, quel est l'ensemble d'activations fonctionnelles permettant de tester cette ressource et d'observer les résultats du test en sorties primaires?

Nous présentons dans les sous-paragraphes suivants les deux phases de la génération fonctionnelle de test qui sont les suivantes :

- phase de consistance
- propagation en avant des sorties de la ressource.

Ces deux étapes sont basées sur des techniques d'intelligence artificielle. Car le problème que nous avons posé ci-dessus s'apparente aux problèmes rencontrés en intelligence artificielle :

- l'objectif est bien fixé
- on utilise des connaissances pour atteindre cet objectif.

Notons que cette étude est seulement une étude préliminaire et par conséquent les détails d'implémentation de l'outil ne pourront pas être donnés ici.

III.1 - PHASE DE CONSISTANCE

La phase de consistance consiste à déterminer les activations fonctionnelles au niveau des entrées primaires qui permettent d'activer et de tester la ressource "cible". L'activation de la ressource "cible" consiste à fournir à ses entrées un ensemble de séquences d'entrées : activation fonctionnelle d'un chemin du graphe du GIT d'une ressource.

La question suivante se pose : est-ce qu'il faut ou non activer toutes les fonctions de la ressource?. Nous pensons que l'utilisateur du sytème doit être maître de ses choix ; dans le cas d'un test d'identification (par exemple le test d'une ressource de type contrôleur), il doit activer toutes les fonctions de la ressource "cible".

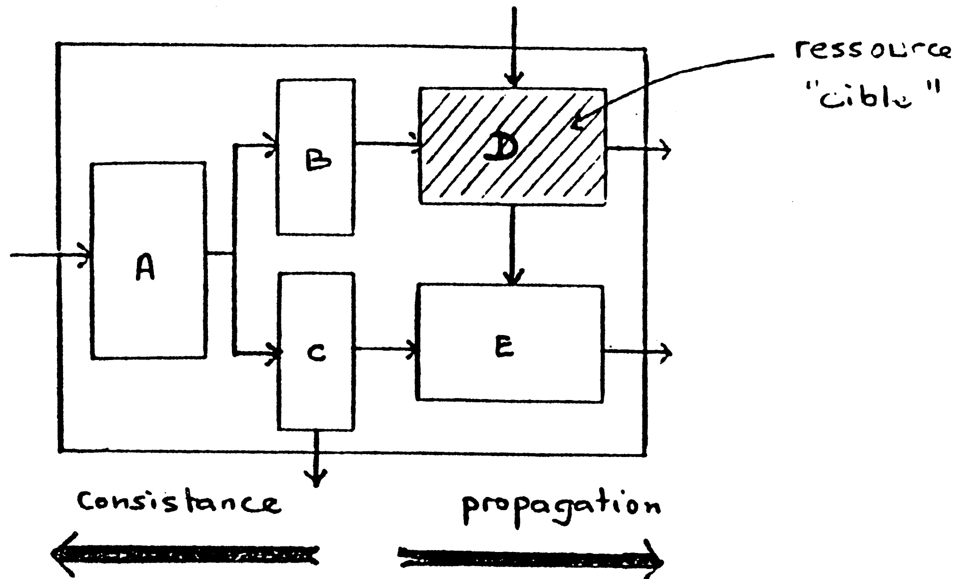


Figure 2-48.

Par la suite, nous allons étudier les points suivants :

- la stratégie utilisée lors de cette phase de consistance
- le problème de choix des règles lors de leur utilisation.

Le problème que nous allons essayer de résoudre est un point important de la conception du système. En effet, l'efficacité du système dépend de la stratégie par laquelle seront faites les choix de connaissances à utiliser. Dans cette phase de consistence nous considérerons trois étapes :

- étape 1 : détermination des ressources qu'il faut "traverser" à partir des entrées primaires jusqu'aux entrées de la ressource à tester (ressource "cible").
- étape 2 : utilisation des résultats de la première étape pour orienter le choix des règles en faisant un chaînage arrière : marquages des règles.

- étape 3 : propagation en avant à partir des entrées primaires jusqu'à

l'activation de la ressource "cible" en utilisant le chaînage en avant et le marquage des règles résultant de l'étape précédente.

Dans le langage de l'intelligence artificielle, la première étape nous donne des méta-connaissances qui serviront à choisir les règles à appliquer;

III.1.1 - Etape 1

L'objectif de cette phase et d'établir un ensemble de connaissances permettant de choisir les règles pendant l'étape 2 que nous développerons après ce paragraphe. Il est évident que l'application directe de la stratégie utilisée dans l'étape 2, à savoir le chaînage en arrière risquerait de nous conduire à des situations d'échec nécessitant une technique de "back-tracking" assez complexe. De plus, il est difficile de déterminer à quel moment on peut considérer que l'on est dans une situation d'échec.

Exemple

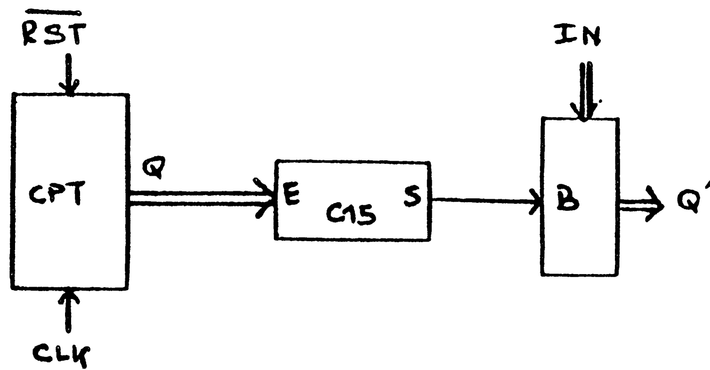


Figure 2-50.

Les règles que l'on peut définir pour la ressource CPT (compteur modulo 16) sont :

- 1° si non RST alors Q := 0 ;

- 2° si CLK alors Q := (%Q + 1) mod 16 ;

pour la ressource C15, qui est comparateur à 15, la connaissance est la suivante :

- 3° si change(E).et (E=15) alors S := V ;

La ressource B (Registre) est chargée à la valeur de IN si S = V(rai), cette connaissance est résumée comme suit :

- 4° si S alors Q' := %Q ;

Pour tester la fonction de chargement de la bascule, il est nécessaire d'appliquer les règles suivantes :

- n°1 : pour initialiser le compteur à 0
- n°2 : pour incrémenter le compteur ⟶ $Q = 1
- n°2 : pour 14 fois ⟶ $Q = 15
- n°3 : pour mettre à V(rai) la sortie S de C15

- n°4 : activation de la fonction "chargement de B".

Cette séquence est plus évidente à trouver pour un être humain (compte tenu de la faible complexité des informations). Mais pour l'automatisation de la recherche d'une telle séquence, il nous faut établir au préalable certaines informations permettant de guider, au cours du chaînage en arrière, le choix des règles et de trouver la séquence dans l'ordre 4, 3, 2, 1.

Alors nous proposons dans cette étape un moyen pour faciliter notre approche : nous construisons pour chaque circuit à étudier un graphe que nous appellerons "graphe de pseudo-précédence".

- **construction du graphe de pseudo-précédence**

A partir de la connectique décrite dans la spécification du circuit en CADOC.LD, nous construirons le graphe de pseudo-précedence. Pourquoi pseudo-précédence ? En effet, le graphe ne donne pas l'ordre d'application des règles au niveau d'un fichier de connaissances mais il donne un ordonnacement des recherches par fichier.

Ce graphe permet de partitionner l'ensemble de toutes les règles (en faisant l'union de tous les fichiers) en deux catégories :

- les règles du premier plan qui sont succeptibles d'être utilisées
- les règles du second plan qui ne seront même pas consultées.

Exemple

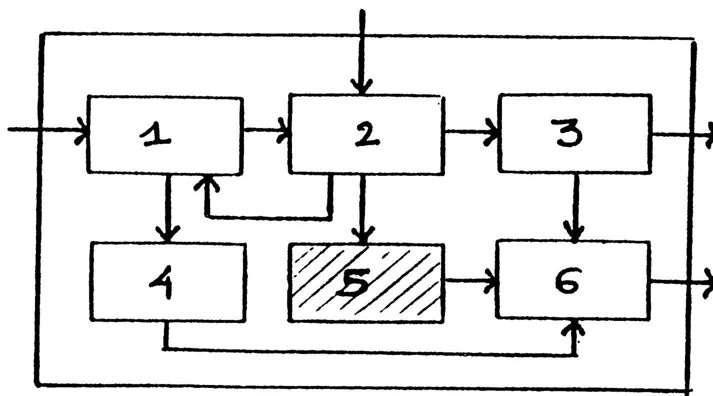


Figure 2-51.

Les règles des ressources 1, 2 seront au premier plan, celles des autres ressources seront au second plan.

Définitions

1) Le graphe de pseudo-précédence est un graphe où les noeuds représentent les numéros de resources (ou de fichier de connaissances) et les arcs orientés représentent la relation suivante :

$n_i \longrightarrow n_j$ si n_i a au moins une sortie vers n_j.

2)Le noeud I est un noeud reservé représentant l'ensemble des entrées primaires du circuit.
Le graphe de pseudo-précédence de notre exemple est :

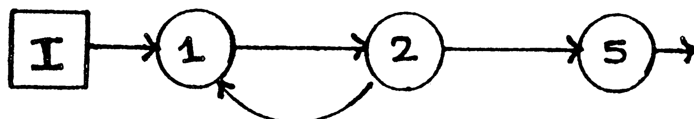


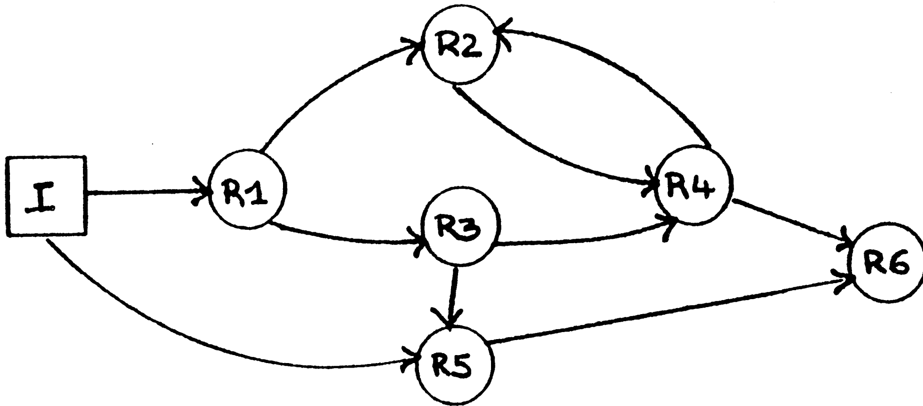
Figure 2-52.

Ce graphe sera consulté pour résoudre le problème. D'après la définition du graphe on peut dire que si n_i est en relation avec n_j, alors n_i contient des règles succeptibles d'être élues car n_i modifie les entrées de n_j. En conséquences le graphe de pseudo-précédence permet déjà de réduire le champs de recherche des règles.

3) On appellera "noeud actif" un noeud du graphe dont au moins une règle a été utilisée dernièrement pour la résolution du problème. A l'étape initial, le noeud actif est celui représentant de la ressource à tester : noeud actif initial.
Dans notre exemple, le noeud actif initial est le noeud 5.

4) On appellera "champs de recherche", l'ensemble des noeuds antécédents des noeuds actifs. La recherche des règles à activer pour la résolution se fait dans le champs de recherche.

Exemple



ETAPE	NOEUDS ACTIFS	CHAMPS DE RECHERCHE
0	R6	{R4, R5}
1	{R4, R5}	{I, R2, R3}
2	{R2, R3, I}	{R1, R4 }
3	{R4, R1}	{R2, R3, I}
4	{R2, R3, I}	{R1, R4}

.
.
.

Figure 2-53.

Remarque

Nous avons une répétition des séquences {R2, R3, I} ----> {R4, R1}. Le problème que nous devons résoudre dans cette situation est de définir le nombre de répétitions à faire. Pour présenter le principe, nous considérons d'abord un graphe sans circuit.

III.1.2 - Etape 2

- **graphe sans circuit**

Le cas le plus simple est celui où le graphe contient uniquement un chemin de I vers n_j (ressource "cible").

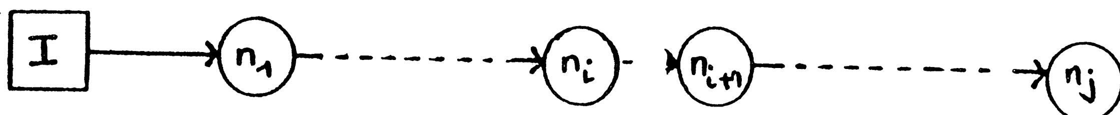


Figure 2-54.

En appliquant la technique de chaînage arrière à partir de n_j on arrivera à déterminer les règles nécessaires en utilisant à chaque étape le champs de recherche.

La stratégie que nous allons appliquer est :

- à l'étape j, $j \in N$.

1- soit n_a le noeud actif à l'étape j, ayant k règles dont la $1^{ième}$ a été choisie.

2- soit $\{n_i\}$ le champs de recherche à l'étape j

3- au niveau de n_i, on fera une partition des règles en deux classes : celles dont les échéanciers de sorties affectent toutes les variables d'entrées de n_a utilisées dans la $1^{ième}$ règle de n_a, et celles qui ne vérifient pas cette condition.

4- on cherchera une règle de la première classe telle que celle-ci, après unification (cf. paragraphe III.1.4)vérifie les conditions et les temporisations spécifiées dans la $1^{ième}$ règle de n_a (Pour la facilité de compréhension, nous avons supposé qu'une seule règle ait été élue).

5- cette règle sera marquée, et sera le point de départ de la nouvelle étape $j + 1$.

6- le noeud actif n_a devient n_i

7- si n_a = I alors le problème est résolu
sinon aller en 1 avec $j := j + 1$.

- **initialisation des variables internes**

Etant donné que les variables internes sont modifiées par la ressource elle-même, les règles susceptibles de modifier leur valeur ne peuvent être choisies que parmi les règles de la ressource.

Le critère de recherche de telles règles est très simple, car toute règle dont la partie "action" affecte un échéancier contenant au moins un couple (valeur, date) avec valeur ≠ X peut être élue. On choisira par exemple, celle qui initialise le plus de variables internes.
Mais si de telles règles n'existent pas, alors le système signalera cette impossibilité d'initialiser les variables internes.

Exemple

Considérons un circuit dont l'une de ses ressources internes est celle décrite dans l'exemple donné figure 2.46a,b,c.

Supposons que cette ressource corresponde au noeud actif, et que la règle du chemin 2 soit élue. Cette règle utilise dans sa partie "condition"

une variable interne à savoir x. Cette variable doit être initialisée.

D'après notre stratégie, la règle du chemin 1 est celle que nous cherchons : la variable x est affectée à (%incir, $\%\theta_1^1$).

Donc les règles 1 et 2 seront marquées pour ce noeud.

Pour l'étape suivante, on considèrera d'abord les règles correspondant aux initialisations des variables internes, puis celles élues normalement.

En résumé, à partir du graphe de pseudo-précédence nous avons utilisé le principe du chaînage arrière pour marquer certaines règles de certaines ressoures. Cs ressources sont celles représentées dans le graphe de pseudo-précédence ; dans le cas simple envisagé, une seule règle est marquée pr ressource, sauf s'il est nécessaire d'initialiser les variables internes.

Exemple

Soit le graphe suivant :

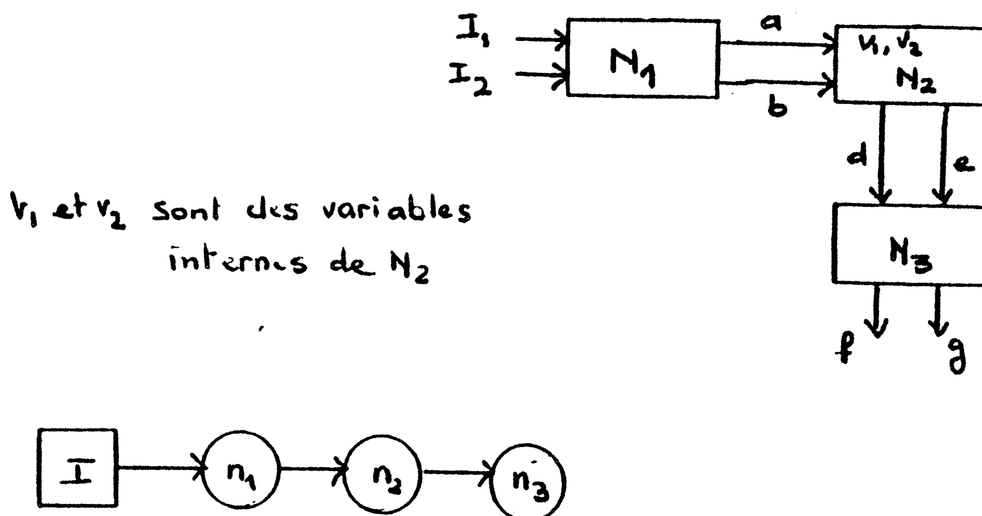


Figure 2-55.

Notations

Nous utiliserons les notations suivantes dans les expressions des règles ci-dessous :

- $N_{i,k}$: numéro absolu de règle, avec i indice de la ressource et k numéro relatif de la règle

- $c_{i,k}$ $(a_1, \ldots, a_n)$: condition temporisée de chemin qui utilise les variables $a_1, \ldots, a_n$, avec i l'indice de la ressource et k le numéro relatif de la règle;

- $s_{i,j}$ $(b_1, \ldots, b_m)$: expressions des échéanciers des variables b_1,

..., b_n (de sorties ou internes).

Les fichiers de connaissances des ressources sont :

- pour N_1 :

n° $N_{1,1}$: si $c_{1,1}(I_1)$ alors $s_{1,1}(a)$

n° $N_{1,2}$: si $c_{1,2}(I_1, I_2)$ alors $s_{1,2}(c)$

n° $N_{1,3}$: si $c_{1,3}(I_1, I_2)$ alors $s_{1,3}(a, c)$

- pour N_2 :

n° $N_{2,1}$: si $c_{2,1}(a, v_2)$ alors $s_{2,1}(e)$

n° $N_{2,2}$: si $c_{2,2}(c)$ alors $s_{2,2}(d, v_1)$

n° $N_{2,3}$: si $c_{2,3}(a)$ alors $s_{2,3}(v_2)$

- pour N_3 :

n° $N_{3,1}$: si $c_{3,1}(e)$ alors $s_{3,1}(f,g)$

n° $N_{3,2}$: si $c_{3,2}(d)$ alors $s_{3,2}(f)$

Soit $v \in \{I_1, I_2, a, b, c, d, e, f, g, v_1, v_2\}$, toutes les variables de cet ensemble étant des variables booléennes,

et - $s_{i,j}(v)$ <==> [v:= (1, %Θ_i)]

- $c_{i,j}(v)$ <==> [v = (1, %Θ_i)]

(cf. évaluation de la partie condition d'une règle au paragraphe III.4)

pas	noeuds actifs	règles marquées	commentaires
0	n_3	$N_{3,1}$	
1	n_2	$N_{2,1}$	v_2 à initialiser
2	n_2	$N_{2,3}$	
3	n_3	$N_{1,1}$ ou $N_{1,3}$	
4	n_3	$N_{1,1}$	
5	I		fin

- **graphe avec circuit**

Considérons le graphe ci-dessous, contenant un circuit tel qu'un seul noeud du circuit soit relié à I.

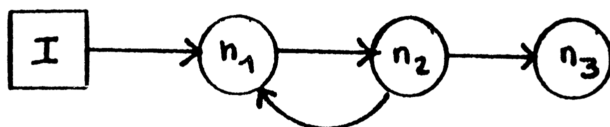


Figure 2-56.

Nous pouvons procéder de la même manière que dans le cas d'un graphe sans circuit jusqu'à ce que le noeud N_1 soit actif pour la première fois. A partir de cette situation, le problème se pose différemment.

En effet, en faisant un chaînage arrière à partir de la règle marquée de N_1, nous devons trouver une ou plusieurs règles de N_2 et I. D'après le graphe

ci-dessus, N_2 n'est pas relié au noeud I. Autrement dit, la ressource N_2 n'est commandable qu'à travers N_1. Le problème qui se pose est alors ici un problème de commandabilité de N_2. Deux cas peuvent être considérés :

- <u>1er cas</u> : en faisant le chaînage arrière à partir de la règle marquée de N_1, aucune règle de N_2 n'est élue ; sauf une ou plusieurs règles de N_1 (dans le cas où il faut initialiser une ou plusieurs variables internes de N_1) sont élues, et seront marquées. Dans ce cas, le chemin réel parcouru est un chemin sans circuit. Cette règle est une règle dont les conséquences permettent l'initialisation de N_2.

- <u>2ième cas</u> : en faisant le chaînage arrière à partir de la règle marquée de N_1, une ou plusieurs règles de N_2 et de I sont élues.
<u>Si</u> après un certain nombre de"boucles" entre N_1 et N_2 on arrive à marquer une règle de N_1 telle qu'on se retrouve dans le cas précédent alors le problème est résolu.

<u>Sinon</u> le problème risque de ne pas avoir une solution auquel cas on est dans une situation d'échec.

Par ailleurs, nous savons que la limite supérieure de ce nombre de "boucles" dépend fortement de la fonctionnalité du circuit, par conséquent la détermination de cette limite est impossible. La solution que nous proposons est alors un système interactif. Le système fonctionnera comme suit dans ce cas :

- dès qu'un circuit du graphe est reconnu, le système se mettra en mode interactif.
Dans notre exemple, le système se mettra en mode interactif après l'activation de N_1 et le marquage d'une ou plusieurs règles de N_2.

- le système fournira les numéros des règles qui peuvent être élues et la règle marquée parmi celles-ci. L'utilisateur pourra changer ce choix.

En fait, la situation d'échec que nous avons mentionnée ci-dessus ne devrait pas arriver si le circuit est bien spécifié : il existe toujours une manière d'initialiser le circuit.

La solution que nous proposons permet deux choses :

- de minimiser le nombre de parcours du circuit.
- de donner la possibilité à l'utilisateur de s'apercevoir d'une éventuelle erreur de conception.

Exemple

Soit le graphe suivant :

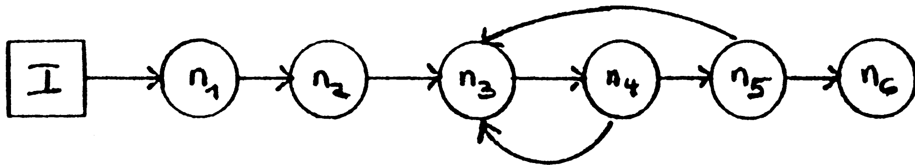


Figure 2-57.

et supposons qu'à chaque étape une seule règle soit marquée.

p

étape	noeuds actifs	mode système
0	N_6	automatique
1	N_5	↓
2	N_4	
3	N_3	
4	N_2,N_4,N5	interactif
5	N_1,N_3,N_4(par exemple)	↓
6	I,N_4,N_2	
7	N_3	
8	N_2	automatique
9	N_1	↓
10	I	

Dans cet exemple, nous pouvons remarquer les changements de mode du système. Nous avons défini quand le système passe en mode "interactif". Nous allons définir quand le système passe en mode "automatique".

A l'initialisation de cette deuxième étape de la phase de la consistance, le système est en mode "automatique". Le passage du mode "interactif" au mode "automatique" se fait quand tous les noeuds actifs sont activés pour la première fois. En effet, ceci assure le fait de quitter un circuit du graphe.

En résumé, nous avons présenté dans ce paragraphe les principes de bases de notre approche. Dans cette partie nous avons supposé que toutes les connaissances nécessaires sont disponibles au niveau de chaque noeud. Ceci est possible par l'utilisation d'un outil d'exécution symbolique temporisée. Mais dans une première phase de réalisation de ce système, l'utilisateur est amené à introduire les règles en utilisant un langage donné (cf. paragraphe II.2). Alors la consistance des connaissances d'une ressource n'est pas assurée. Dans ce cas le système doit tenir compte de cet aspect.

Ce problème est rencontré dans les Systèmes Experts qui sont conçus pour accepter l'introduction de nouvelles connaissances au cours du temps. Nous devons opérer de la même manière pour notre problème, c'est-à-dire que l'utilisateur introduira des règles pour compléter les connaissances sur une ressource fonctionnelle quelconque.

Le système demandera à l'utilisateur de nouvelles connaissances quand il se trouve dans la situation suivante :

- l'ensemble des noeuds actifs ne contient pas I

- et aucune règle du champ de recherche ne pourra être marquée.

Exemple

Reprenons l'exemple du compteur déclenchant le chargement d'un registre donné figure 2-50.

Le graphe correspondant est :

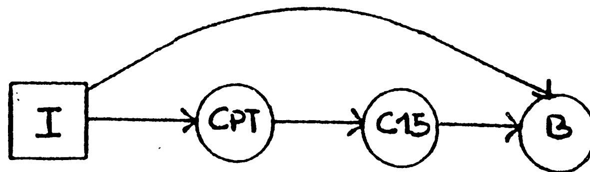


Figure 2-58

où I = {$\overline{RST}$, IN, CLK}

Supposons que le noeud actif soit C_{15} et son unique règle soit marquée à savoir :

si change(E) et (E = 15) alors S := V(rai) ;

Le champ de recherche est composé du noeud CPT. La deuxième règle de CPT est une connaissance sur la fonction "incrémentation du compteur qui est :

si CLK alors Q := (%Q + 1) mod 16 ;

D'après la connectique définie dans la description de ce circuit, la variable E a la même valeur que Q (cf. figure 2-50). L'autre règle de CPT est une règle permettant l'initialisation de CPT. Aucune des deux règles, telles qu'elles sont données, ne peuvent être élues.
Dans ce cas le système demande d'autres informations à l'utilisateur. Ce dernier pourra lui préciser qu'il faut exécuter une fois la règle n°1 est 15 fois la règle n°2. ce qui permet de satisfaire la règle de C_{15}. Après 15 fois la règle n°2 de CPT, Q = 15.

Les informations que l'utilisateur donne, sont de deux types :

- informations sur le fonctionnement du circuit ; dans ce cas l'utilisateur introduit de nouvelles règles dans le fichier de connaissances d'une ressource.

- informations sur la manière d'utiliser les règles, comme le cas de notre exemple.

Remarque

La réalisation et l'utilisation de l'outil d'exécution symbolique temporisée permettra d'éviter cette démarche interactive. Les problèmes de boucles (cf.paragrapheIII.4) devraient être résolus pendant l'exécution symbolique d'une manière complètement automatisée ou interactive.

- **généralisation**

Dans ce qui a précédé, nous avons considéré des cas où le noeud actif est unique. Plus généralement, à un instant donné, l'ensemble des noeuds actifs et le champ de recherche ne sont pas des singletons.

La stratégie généralisée est :

1- soit N_i l'ensemble des noeuds actifs à l'étape t_i dont m règles de l'ensemble ont été marquées

2- soit R_{i+1} le champ de recherche (card(R_{i+1}) > 1)

3- pour toute règle , parmi les m marquées, non vérifiée par aucune règle de R_{i+1} I : changer le mode du système en mode interactif pour résoudre ces règles.

4- marquage des règles de N_{i+1} élues.

5- le nouvel ensemble N_{i+1} estl'ensemble des noeuds de R_{i+1} possédant au moins une règle marquée. Les autres noeuds sont désactivés.

6- si N_{i+1} contient au moins un noeud ayant déjà une règle marquée : changer le mode du système en mode interactif.

7- si N_{i+1} ne contient que des noeuds n'ayant aucune règle marquée : changer le mode du système en automatique.

8- aller à 1 tant que N_{i+1} est différent de {I}.

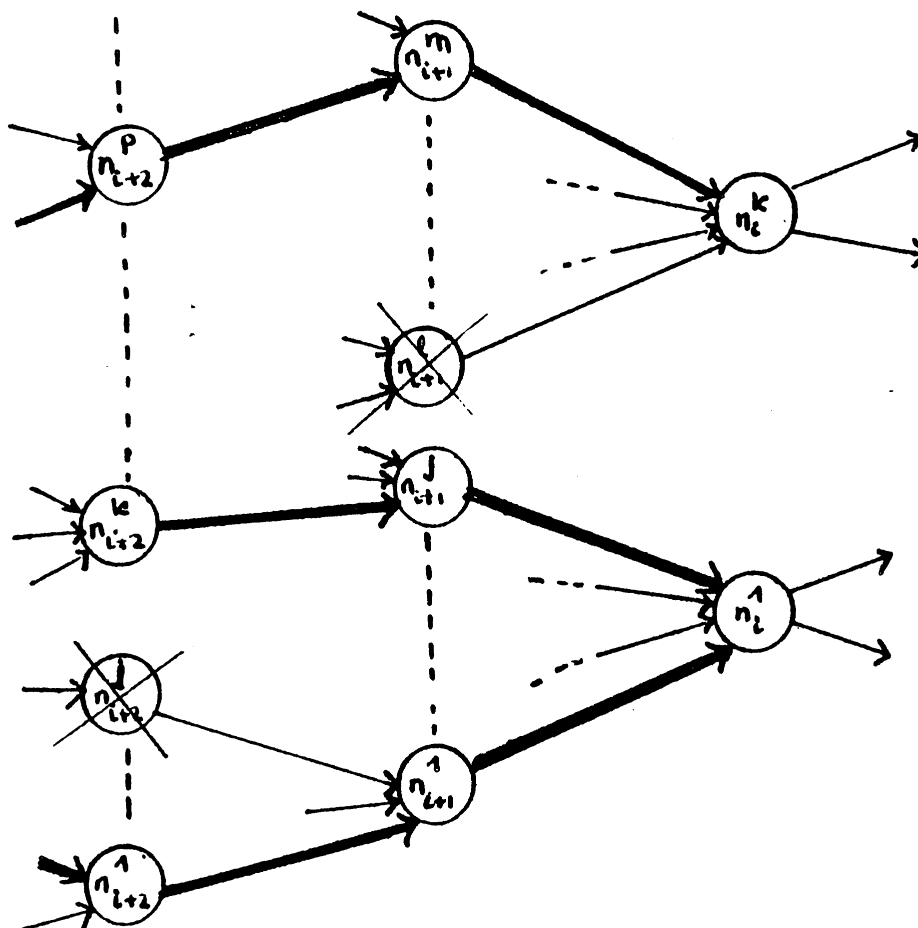


Figure 2-59.

- **Conclusion**

Dans ce paragraphe, nous avons essayé de présenter la méthode utilisée dans la deuxième étape de la première phase qui consiste à faire une propagation en arrière à partir de la ressource fonctionnelle à tester. Nous avons défini un graphe de pseudo-précédence. Ce graphe permet d'orienter le choix des règles à utiliser pendant le chaînage en arrière pour atteindre les

entrées primaires du circuit.

Le système est obligé dans certaines situations de faire appel à la collaboration de l'utilisateur : mode interactif. En effet, dans un premier temps les règles sont supposées déterminées par l'utilisateur. La conséquence d'un oubli est que le système ne peut plus faire aucun raisonnement.

Le système est aussi en mode interactif dans le cas où il "raisonne" selon un circuit du graphe. Le mode interactif peut être évité en utilisant l'outil d'exécution symbolique qui a été présenté au chapitre III.

III.1.3 - Etape 3

A l'issue de la deuxième étape de la phase de consistance, une ou plusieurs règles sont marquées au niveau de chaque fichier de tous les noeuds du graphe de pseudo-précédence. Ces règles ont un attribut chronologique qui est déterminé pendant le marquage des règles. Ces attributs chronologiques seront utilisés lors de cette étape 3.

Exemple

Soit le fichier de connaissances contenant des règles marquées.

- fichier N1 contenant 6 règles dont 4 marquées.

$$\begin{array}{cccccc} r_1, & r_2, & r_3, & r_4, & r_5, & r_6 \\ \downarrow & \downarrow & \downarrow & & \downarrow & \\ (1) & (3) & (2) & & (4) & \end{array}$$

Lors de l'étape 3, les règles de N1 seront appliquées dans l'ordre décroissant des attributs chronologiques.

L'étape 3 termine cette première phase de consistance. Cette étape est nécessaire car il faut maintenant construire 'histoire complète de toutes les variables d'entrées/sorties et internes. En effet, lors de la deuxième

étape, nous avons seulement marqué des règles en leur attribuant des numéros d'ordre chronologique. Nous avons aussi établi des expressions sur les variables de dates lors de l'unification des règles.

Mais nous n'avons pas explicitement déterminé leur valeur symbolique absolue. Pour situer le problème qui doit être résolu par l'existence de cette étape 3, prenons l'exemple suivant :

Exemple

Soit le circuit C composé de six ressources internes :

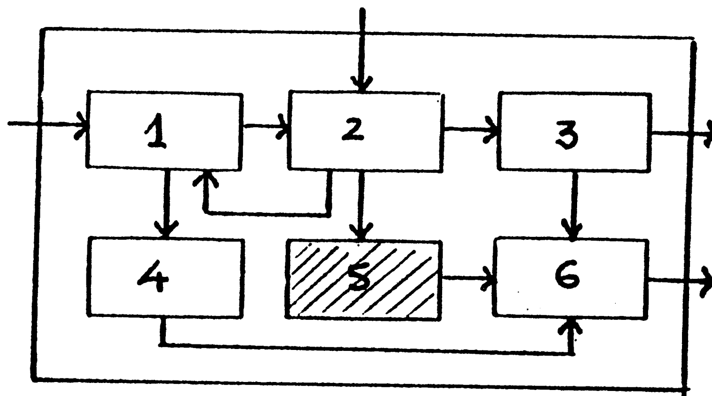


Figure 2-60.

Nous avons marqué des règles des ressources 1 et 2 pendant l'étape 2. Une des règles de la ressource 1 qui est marquée peut activer les ressources 2 et 4. Nous sommes certain que la ressource 2 est activée par cette règle (résultat de l'étape 2). Quant à l'activation de la ressource 4 par cette règle, nous devons vérifier si elle est effectuée ou non.
Nous devons faire de même avec les règles marquées de la ressource 2.

Ceci est une des raisons pour inclure cette étape 3 dans la phase de consistence. Une autre raison est la suivante :

Dans la prochaine phase (phase de propagation), nous devons faire des tests sur les états des variables internes et d'entrées/sorties. Les histoires de ces variables doivent donc être construites complètement.

Dans notre exemple, les sorties de la ressource 5 doivent être propagées vers les sorties en traversant la ressource 6. Cette dernière peut être activée par les ressources 4 et 3. Tous ces noeuds n'appartiennent ni au graphe de pseudo-précédence ni à la liste des ressources traversées par les valeurs des sorties de la ressource 5. Il est donc nécessaires d'avoir les échéanciers des variables de sorties des ressources 4 et 3.

- propagation en avant

La technique utilisée est le chaînage en avant. Cette étape est plus simple car le choix de certaines règles et l'ordre d'application de ces règles sont déjà faits. Ce sont les règles nécessaires pour activer la fonction de la ressource à tester.

A l'initialisation de cette étape, on définira un repère temporel symbolique : soit $\$\theta_0$ la valeur de l'instant initial.
A chaque fois qu'on utilisera une règle, on déterminera les valeurs des instants de référence utilisés dans la règle par rapport au repère temporel global. On remplacera dans les échéanciers les variables de dates ($\%\theta_i$) par leur valeur symbolique.

En ce qui concerne les variables de données, on donnera des valeurs symboliques ($incir1, $incir2, ...) à chaque fois que celles-ci sont nécessaires.

- construction des histoires des variables

L'histoire d'une variable est représentée par son échéancier global. Les échéanciers des variables internes et d'entrées/sorties sont construites en utilisant les valeurs symboliques des variables des règles ; les variables de données et de dates (précédées de %) seront affectées par leur valeurs symboliques globales.

Dans les termes des systèmes à base de règles, les échéanciers de toutes les variables internes et d'entrées/sorties constituent l'ensemble des faits générés à partir des faits initiaux ; ces derniers étant les échéanciers des entrées/sorties et des variables internes à l'instant initial ($\%\Theta_0$).

- **détermination des règles du second plan**

Rappelons que les règles du second plan sont celles appartenant aux ressources non représentées sur le graphe de pseudo-précédence. Dans l'exemple donné ci-dessus, les règles du second plan sont celles des ressources 4,3, 6.

Les échéanciers des variables qui activent les ressources du second plan, sont définis avec des valeurs bien déterminées. Dans ce cas, il suffit de chercher les règles du second plan qui sont satisfaites par ces échéanciers.

Pour rendre plus efficace la recherche, on utilisera la connectique du circuit. Pour notre exemple, si une règle marquée de la ressource 1 est considérée, il est inutile de chercher dans le fichier de connaissances des ressources 3 et 6.

Par ailleurs, rappelons qu'une règle correspond à un chemin de l'arbre d'exécution symbolique du GIT d'une ressource : deux chemins distincts n'ont pas les mêmes conditions de chemins temporisées. Ainsi, nous pouvons conclure qu'une et une seule règle d'un fichier de connaissances sera élue. Par conséquent, nous n'aurons pas à poser le problème de "retour en arrière". Car le choix résultant de l'étape 2 implique la construction de ces échéanciers. Cette étape correspond à la phase d'implication décrite dans les méthodes classiques de test (cf. chapitre II).

En conclusion, cette étape permet de construire les échéanciers effectifs correspondant aux choix faits lors de l'étape 2. Un repère global d'instant symbolique est utilisé.

III.1.4 - Unification et marquage des règles

Pendant la deuxième étape, nous utilisons la stratégie de chaînage en arrière. A chaque évolution, nous devons déterminer les règles à marquer telles que leurs conséquences vérifient les conditions temporisées des règles courantes (c'est-à-dire, dernièrement marquées). Le problème que nous allons résoudre dans ce paragraphe est la détermination des règles à marquer.

Nous allons considérer dans un premier temps que l'ensemble des règles courantes soit un singleton. Soit n_i le noeud actif et $r_{i,1}$ la règle de n_i courante. D'après notre stratégie, soit $C_R(n_i)$ le champs de recherche derivé du noeud n_i. Nous devons déterminer une ou plusieurs règles de $C_R(n_i)$ telles que leurs conséquences vérifient la condition de $r_{i,1}$.

Rappelons que les règles sont temporisées. D'autre part, nous avons utilisé des variables dans la définition de ces règles. Lors de l'utilisation de ces règles, ces variables doivent être instanciées, c'est-à-dire des valeurs doivent être affectées à ces variables.

Avant de marquer une ou plusieurs règles de C_R (n_i), nous devons donc déterminer les valeurs des variables.

Les variables définies dans une règle sont de deux types :
- les variables de données (exemple %incir, ...)
- les variables de dates (exemple $\%\theta_1$, ...)

Ce procédé d'instantiation est appelé unification. La méthode d'unification que nous proposons pour notre problème est précédée d'une réduction du champ de recherche $C_R(n_i)$.

Soient V(I) l'ensemble des variables d'entrées et $V_c(r_{i,1})$ l'ensemble des variables de description utilisée dans la règle, plus précisemment dans la partie "condition" de la règle $r_{i,1}$; Nous pouvons réduire l'ensemble des

règles de $C_R(n_i)$ en le partitionnant en deux classes :

- les règles qui affectent au moins une variable de $V_c(r_{i,1}) - V(I)$: cet ensemble sera noté $R(r_{i,1})$.
- les règles qui n'affectent aucune variable de $V_c(r_{i,1}) - V(I)$: on notera cet ensemble $\bar{R}(r_{i,1})$. En effet, les conditions sur des variables d'entrées primaires sont toujours satisfaisables. Il suffit de leur donner des valeurs symboliques telles que la condition soit satisfaite.

Exemple

Soit la règle suivante :

AVEC

INSTANT-INITIAL : $\%\theta_0^1$

$\%\theta_1^1$ = a et non b : fdx(H,1,$\%\theta_0^1$)

INSTANT-FINAL : $\%\theta_2^1$ = fdx(H,3,$\%\theta_1^1$)

<u>SI</u>

$a = (X, \%\Theta_0^1)(1, \%\Theta_1^1 - s_a)(X, \%\Theta_1^1 + s'_a)$
$b = (X, \%\Theta_0^1)(0, \%\Theta_1^1 - s_b)(X, \%\Theta_1^1 + s'_b)$
$incir = (X, \%\Theta_0^1)(\%incir, \%\Theta_1^1 - s_{in})(X, \%\Theta_1^1 + s'_{in})$

<u>ALORS</u>

$x = (\%incir, \%\Theta_1^1)$

<u>FIN</u>

Supposons que la ressource dont une des règles est celle donnée ci-dessus, soit connectée avec une autre ressource partageant toutes les deux la même horloge.

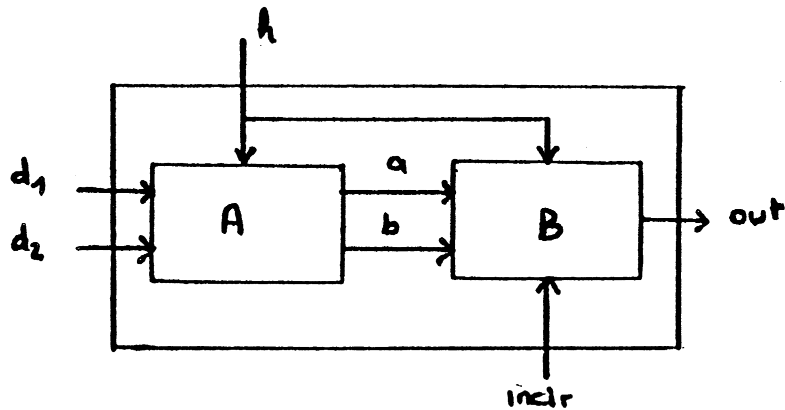


Figure 2-61.

Soit le GIT de la ressource A :

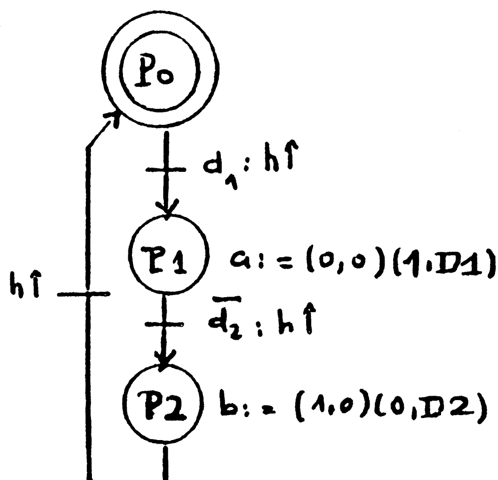


Figure 2-62.

La règle que l'on définit pour A est la suivante :

règle j :

AVEC

INSTANT-INITIAL : $\%\Theta'_0$

$\%\Theta'_1 = d_1$: fmx $(H,1,\%\Theta'_0)$

$\%\Theta'_2 = \overline{d_2}$: fmx $(H,1,\%\Theta'_1)$

INSTANT-FINAL : $\%\Theta'_3$ = fmx $(H,1,\%\Theta'_2)$

SI

$d_1 = (X, \%\Theta'_0)(1, \%\Theta'_1 - \varepsilon)(X, \%\Theta'_1 + \varepsilon')$

$d_2 = (X, \%\Theta'_0)(0, \%\Theta'_2 - \varepsilon)(X, \%\Theta'_2 + \varepsilon')$

ALORS

a := $(0, \%\Theta'_1)(1, \%\Theta'_1 + D1)$

b := $(1, \%\Theta'_2)(0, \%\Theta'_2 + D2)$

FIN

Le graphe de pseudo-précédence est :

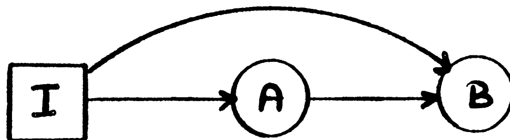


Figure 2-63.

$V_c(i) = \{h, incir, a, b\}$ $V(I) = \{h, incir\}$

$j \in R(i)$, car la règle j affecte les variables a, b $\in V_c(i) - V(I)$

En ce point nous savons donc que les conséquences de la règle j peuvent vérifier les conditions de la règle i. Supposons que j soit l'unique règle appartenant à R(i).

Ayant déterminé les éléments de $R(r_{i,1})$, nous devons choisir une seule règle s'il y en a plusieurs.

La méthode d'unification, dans notre problème, consiste à déterminer des équations et/ou inéquations à partir des données suivantes :

- des échéanciers des variables d'un sous-ensemble de règles de $R(r_{i,1})$
- des définitions des instants de référence utilisés dans la règle $r_{i,1}$;

On appliquera cette méthode pour chaque règle courante $r_{i,1}$. Nous voulons illustrer le principe sur l'exemple ci-dessus.

Exemple

Les échéanciers de deux variables "a" et "b" de la règle j sont représentés par les chronogrammes suivants :

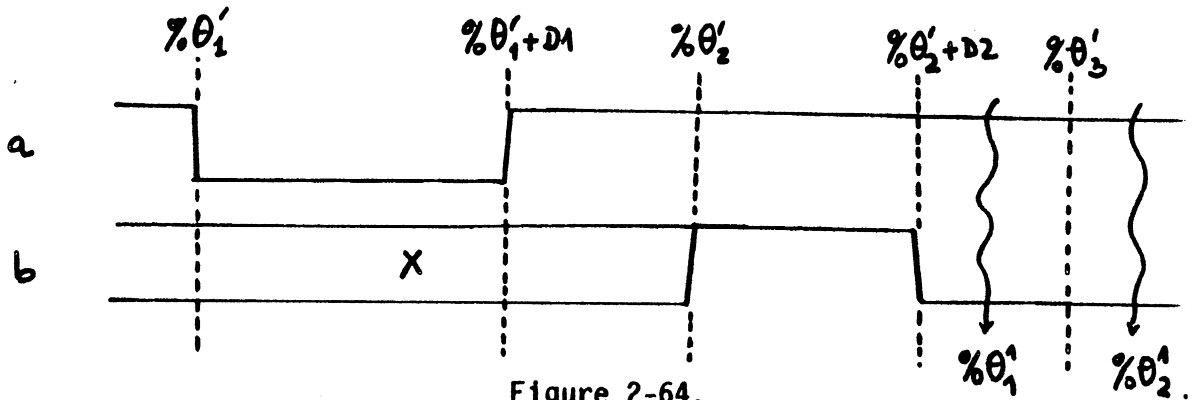


Figure 2-64.

A partir des définitions des instants de référence utilisés dans la règle i, nous pouvons déduire que $\%\theta_1^1$, $\%\theta_2^1$ peuvent avoir comme valeur celle définie par les expressions suivantes :

$\%\theta_1^1$ = fdx(H,1,$\%\theta_2'$ + 2)

$\%\theta_2^1$ = fdx(H,2,$\%\theta_2'$ + 2)

Malheureusement, il existe plusieurs possibilités de construire le chronogramme donné figure 2-64. Par exemple on pourra avoir :

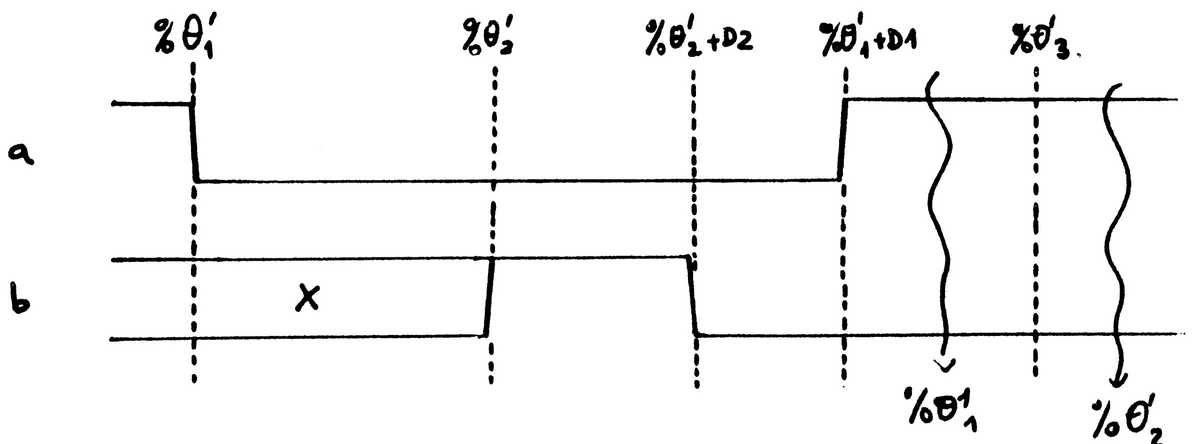


Figure 2-65.

Dans ce cas, les équations sont les suivantes :

$\%\Theta_1^1 = fdx(H,1,\%\Theta_1' + 1)$

$\%\Theta_1^1 = fdx(H,2,\%\Theta'_1 + 1)$

L'objet de ce chapitre IV étant une réflexion sur la possibilté de l'utilisation des techniques de l'intelligence artificielle, ce point particulier sur l'unification des règles temporisées est un important sujet de recherche future.

Dans cet exemple nous avons tous les échéanciers nécessaires pour appliquer la méthode. Dans le cas général, nous devons organiser l'ensemble $R(r_{i,l})$ en sous-ensembles tels que chacun d'eux couvre les variables de $V_c(r_{i,l})$: l'ensemble des variables d'entrées ou internes utilisées dans la partie "condition" de la règle $r_{i,l}$. La notion de couverture signifie ici que chaque variable de $V_c(r_{i,l})$ est au moins affectée une fois.

Parmi ces sous-ensembles de $R(r_{i,l})$, on en déterminera un qui puisse satisfaire la partie "condition" de $r_{i,l}$. Là aussi, nous devons déterminer quel est l'ordre de consultation des sous-ensembles. La stratégie que l'on peut adopter est de commencer à partir du sous-ensemble ayant le plus petit nombre d'éléments.

Exemple

variables de $V_i(r_i,1)$ / règles de $R(r_i,1)$	a_1	a_2	a_3	a_4	a_5	a_6
r_1	X	X		X		
r_2		X	X		X	
r_3		X		X	X	X
r_4	X					X

Figure 2-66.

On pourra utiliser un tableau de couverture classique ; en colonnes , nous aurons les variables de $V_c(r_{i,1})$ et en lignes, les règles de $R(r_{i,1})$.

Dans l'exemple donné figure 80, on aura les sous-ensembles suivants ;

- $\{r_1, r_2, r_4\}$ --> 3
- $\{r_1, r_2, r_3\}$ --> 3
- $\{r_1, r_3\}$ --> 2
- $\{r_4, r_3\}$ --> 2

On choisira, par exemple, $\{r_4, r_3\}$,puis $\{r_1, r_3\}$ si $\{r_4, r_3\}$ ne satisfait pas la règle $r_{i,1}$, etc ...

Jusqu'ici, nous avons présenté la complexité du problème d'unification en ce qui concerne les variables de dates. Quant aux variables de données, l'unification n'est pas faite explicitement. En effet pour les variables

d'entrées primaires (appartenant à $V_c(r_{i,l}) \cap V(I)$) le problème est toujours résolu en leur attribuant des valeurs symboliques. Et pour les variables internes, on applique la méthode d'évaluation de la partie "condition" d'une règle (cf. paragraphe III.4).

Nous avons essayé de présenter les divers problèmes de l'unification des règles pendant l'étape 2 de la phase de consistance. Le problème important se situe au niveau des variables de dates. Car les chronogrammes générés par les règles "candidates" de $R(r_{i,l})$ doivent d'abord satisfaire temporellement les conditions dans $r_{i,l}$ et ensuite les variables de données.

III.2 - Phase de propagation

Après avoir déterminer les activations fonctionnelles du circuit pour le test d'une ressource "cible" (cf. paragraphe précédent), nous devons étudier l'observation des résultats de test. Dans le cas général, ces résultats doivent être acheminés vers des points d'observation où l'analyse des résultats peuvent être faite. Une analyse préalable des dépendances des variables peut être faite avant l'étude de l'observation de celles-ci : on déterminera si des valeurs, observées en ces points d'observation, sont fortement ou faiblement dépendantes des valeurs de sorties de la ressource "cible". Cette étude est appelée : étude de testabilité fonctionnelle. Nous n'en parlerons pas dans ce paragraphe. Les lecteurs peuvent consulter [BEL84] pour plus de détail.

Les points d'observation doivent être déf is avant cette phase de propagation. Ces points sont généralement les sorties primaires du circuit. Mais actuellement, des recherches sur la possibilité d'observer et de mesurer directement les sorties des parties sous test du circuit sont faites. Ces recherches sont orientées vers l'utilisation des microscopes électroniques à balayage [LAU 84] [BAL 85]. Dans le cas où l'utilisation des MEB est possible, cette phase de propagation n'est plus un problème car le MEB peut

observer directement les sorties de la ressource sous test. Par contre, le problème se pose si nous considérons les sorties primaires comme points d'observation.

Cette phase de propagation est beaucoup plus simple que la phase de consistence. La phase est initialisée avec :

- la règle de la ressource sous test qui correspond à la fonction activée pour le test,
- un ensemble d'échéanciers avec des valeurs symboliques fixées,
- la description de la connectique du circuit.

Contrairement à la phase de consistance, cette phase démarre avec beaucoup d'informations déjà établies, en particulier la règle de départ est connue. La connectique est nécessaire car elle aide le système à faire le choix des règles. A partir de ces remarques, nous pouvons définir les étapes dans cete phase de propagation.

Cette phase comporte deux étapes que nous noterons :

- étape 4 : détermination des ressources qui doivent être obligatoirement activées pour propager les résultats de tet vers des sorties primaires.

- étape 5 : propagation en avant à partir de l'état final de la phase précédente en utilisant le chainage avant.

III.2.1 - Etape 4 : graphe de propagation

Utilisant la connectique du circuit définie dans la description en CADOC.LD, nous pouvons obtenir un autre graphe de pseudo-précédence dont la construction est la suivante :

- un noeud du graphe représente une ressource.
- un noeud $n_i \rightarrow n_j$ si au moins une sortie de n_i connectée à une entrée de n_j.

Le graphe que nous appelons graphe de propagation est construit séquentiellement, c'est-à-dire :

1 - à l'instant initial, on considère le noeud représentant la ressource sous test. Soit n_0 ce noeud.

2 - $n_i = n_0$

3 - on ajoute au graphe les noeuds représentant les ressources connectées à n_i

4 - pour les noeuds définis en 3, établir les arcs $n_i \dashrightarrow n_j$.

5 - pour tous les noeuds définies en 3, aller en 3. (développement des noeuds)

6 - ajouter le noeud I et les arcs de I vers les noeuds du graphe , ainsi que ceux constituant le chemin de I vers un des noeuds du graphe.

Exemple

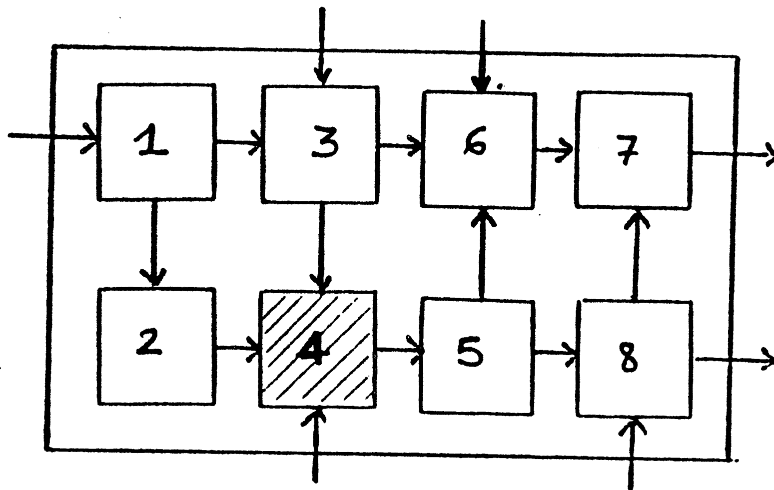


Figure 2-67.

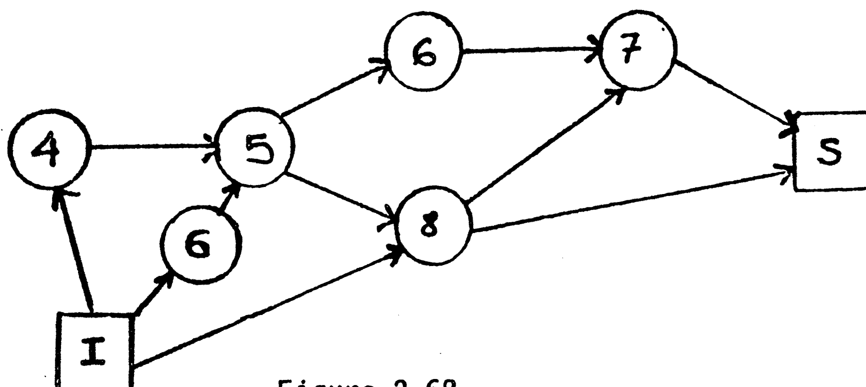


Figure 2-68.

Supposons que la ressource 4 est celle sous test ; le graphe obtenu est donné figure 2-68.

- **Définition**

Le noeud S est un noeud reservé représentant l'ensemble des sorties primaires du circuit.
D'après le graphe de notre exemple, nous avons au moins un chemin de noeud 4 au noeud S. Mais ceci n'est pas suffisant pour dire que les sorties sont réellement observables.

Par contre, si le grahe ne comporte aucun chemin du noeud de la ressource sous test vers le noeud S, alors les résultats de test ne sont pas observables aux points d'observation fixés (ici ce sont les sorties primaires).

A l'issue de cette quatrième étape, nous pouvons extraire une information de testabilité du circuit. Si le graphe obtenu comporte un chemin d'observabilité, il nous faut procéder à la 5ème étape pour conclure.

III.2.2 - Etape 5 : propagation en avant

Cette étape consiste en la propagation effective des résultats de test. Cette étape est presque semblable à l'étape 3 (dans la phase de consistence). A l'initialisation de l'étape 5, les faits initiaux sont composés des échéanciers de toutes les variables obtenus à la fin de l'étape 3. La règle initiale à développer sera celle de la fonction à tester.

L'autre différence est qu'il n'existe pas de règle préalablement marquées. En effet, ceci n'est pas nécessaire car il suffit de consulter le graphe obtenu à l'issue de l'étape 4. Etant donné que deux règles distinctes n'ont pas les mêmes conditions d'activations, à chaque pas de la propagation, nous rencontrons deux situations.

- soit une règle des fichiers de connaissance du noeud considéré est vérifié, auquel cas on avancera.

- soit aucune règle n'est vérifiée, dans ce cas il faut voir globalement la situation. Si aucun développement n'est possible et que le noeud S n'est pas atteint alors nous détectons que les résultats de test ne sont pas observables aux points de S.

Pendant la propagation, le graphe détermine les fichiers des connaissances, dans lesquels il faut choisir en premier les règles. Si aucune règle de l'union de ces fichiers n'est élue, la propagation au niveau des autres règles n'est pas nécessaire.

Par contre, si il y a eu des règles élues au niveau de ces fichiers, il faut effectuer l'implication de ces nouvelles règles sur les règles des autre fichiers correspondant aux ressouces non représentées dans le graphe.
Cette implication est necéssaire pour mettre à jour les échéanciers de toutes les variables du circuit.

Le noeud S est atteint quand l'ensemble des variables affectées par toutes les règles dernièrement élues inclut au moins une variable de sortie. Evidemment, l'observation au niveau d'une seule variable de sortie peut être insuffisante. Nous ne discuterons pas ici du nombre de variables de sorties à observer, mais on peut envisager un système qui passe en mode interactif : il demande à l'utilisateur s'il doit continuer ou non, à chaque fois qu'une valeur est affectée à une sortie.

Les étapes 4 et 5 constituant la phase de propagation sont plus simples, car beaucoup d'informations sont déjà établies au cours de la phase de propagation.

IV - CONCLUSION

Nous avons montré dans ca chapitre IV que le problème de génération fonctionnelle de test peut être résolu par des techniques d'intelligence artificielle. Plusieurs problèmes ne sont pas étudiés, notamment sur les problèmes d'implémentation de notre approche.

Nous avons retenu que cette approche permet de résoudre les problèmes de genération de test des circuits complexes au niveau fonctionnel. En effet, nous avons concludans le chapitre II qu'en ce qui concerne le test de VLSI, une approche mulitniveau (cf. Chapitre II, paragraphe III) ou fonctionnelle est souhaitée pour faire face à la complexité des VLSI.

Notre méthode ne vise pas à générer directement des programmes de test, mais à déterminer un ensemble d'activations fonctionnelles permettant le test des parties d'un circuit. La méthode que nous avons proposée consiste à manipuler des connaissances qui sont représentées sous formes de règles temporisées.

Dans une première phase de réalisation, on pourra définir ces règles par l'intermédiaire d'un langage du type de celui que nous avons utilisé dans ce chapitre.

Dans une deuxième phase un outil d'exécution symbolique (cf. chapitre III) peut être utilisé pour générer automatiquement ces règles.

Notre approche est composée de cinq étapes. Les trois premières constituent la phase de consistence de la détermination des activations fonctionnelles, et les deux dernières étapes forment la phase de propagation des résultats des tests vers les sorties.

Dans la phase de consistance, le point le plus important est l'étape 2. Le but de cette étape est de déterminer les règles qui doivent être nécessairement activées lors de la propagation en avant afin d'assurer l'activation de la fonction à tester de la ressource "cible". L'unification des variables de données ne semble pas être un problème. Mais celle des variables de dates est très importante car nous devons trouver un contexte temporel cohérent lors de l'instantiation des variables de dates. Nous ne prétendons pas avoir résolu tous les problèmes à ce niveau. Mais nous avons essayé de mettre en évidence les points essentiels qui nécessiteraient beaucoup plus d'investigations.

BIBLIOGRAPHIE

[ACK 78] **S.H ACKERS.**
"Binary decision diagnosis"
IEEE Trans. on Comp., vol c-27, pp 509-516, 1978.

[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.

[BAL 85] **P. BALMEY, O. WILLM**
"Etude de l'utilisation systématique d'un microscope électronique à balayage dans le test de fin de fabrication des circuits intégrés complexes"
Rapport de stage de fin d'études, ENSERG, Institut National Polytechnique de Grenoble, Juin 85.

[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 84] **C. BELLON**
"Test fonctionnel des circuits complexes"
Thèse de Docteur d'état en Informatique, Institut National Polytechnique de Grenoble, Octobre 1984.

[BEL 85] **C. BELLON, J. RARIVOMANANA et al.**
"CADOC system : a tool for multilevel description and test generation for VLSI circuits"
7th Int. Symp. on CHDL and their Applications (CHDL 85), pp. 364-380, Tokyo, Japon, Août 1985.

[BEN 84] **D.G. BENNETS**
"Design of testable logic circuits"
Addison-Wesley Publ. Comp., 1984.

[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.

[BLA 83] **J.P. BLANQUART**
"Introduction à la vérification formelle de programmes par exécution symbolique"
Rapport de recherche n° 83.081, Novembre 1983.

[BOR 81] **D. BORRIONE**
"Langages de description de systèmes logiques"
Thèse d'état, INPG, Grenoble, France, Juillet 1981.

[BRE 80] **M.A. BREUER, A.D. FRIEDMAN**
"Functional level primitives in test generation"
IEEE Trans. on Comp., vol c-29, pp 223-234, Mars 1980.

[CHU 74] Y. CHU
"Introducing CDL"
IEEE Computer, pp. 42-44, Décembre 1974.

[CIS 84] G. CISNEROS
"Programmation parallèle et programmation fonctionnelle : Proposition pour un langage"
Thèse de Docteur de 3ième cycle Informatique, INPG, Octobre 1984.

[COR 81] W.E. CORY, W.M. van CLEEMPUT
"Symbolic simulation for functional verification with ADLIB and SDL"
18th DAC, pp 82-89, Nashville, USA, Juin 1981.

[CRA 85] M. CRASTES DE PAULET
"Spécification et simulation fonctionnelles de circuits complexes : le système CADOC"
Thèse de Docteur-Ingénieur en Microéléctronique, Lab. Circuits et systèmes, INPG, Novembre 1985.

[CLU 84] Mc. CLUSKEY, E.C. ARCHAMBEAU
"Fault-coverage of pseudo-exhaustive testing"
FTCS-14, Kissimee, Floride, Juin 1984.

[DAN 82] R.B. DANNENBERG, G.W. ERNST
"Formal program Verification using symbolic execution"
IEEE Trans. on Comp., vol 8E-8, n° 1, Janvier 1982.

[DAV 76] R. DAVID, G.BLANCHET
"About random fault detection of combinatorial networks"
IEEE Trans. on Comp., pp 659-664, Juin 1976.

[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.

[DIO 83] **J. DION, P. ROBINSON**
"Progamming languages for hardware description"
20th DAC, pp 12-16, Miami Beach, USA, Juin 1983.

[DIJ 75] **E.W. DIJKSTRA**
"Guarded commands, nondeterminacy and formal derivation of programs"
CACM pp. 453-457, Août 1975.

[HAN 84] **S. HANRIAT**
"Le langage CADOC.LD : mise en oeuvre de son compilateur"
DEA d'Informatique, INPG, Juin 1984.

[HAN 85] **S. HANRIAT, J. IDT**
"Compilateur de fonctions booléennes et de contrôleur sur réseaux prédiffusés"
Colloque National "Conception de circuits à la demande sur réseaux prédiffusés", pp 493-517, Grenoble, Mai 1985.

[HIL 74] **D.D. HILL**
"Introducing AHPL"
IEEE Computer, pp. 28-30, Décembre 1974.

[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.

[HOA 78] C.A.R. HOARE
"Communicating sequential processes"
Comm. of ACM, vol. 21, n°8, Août 1978.

[FELD 83] S.I. FELDMAN
"The circuit language Xi"
Int. Conf. on Comp. Design (ICCD), pp 652-655, New-York, Octobre 1983.

[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.

[HAL 84] N. HALBWACHS
"Modélisation et analyse du comportement des systèmes informatiques temporisés"
Thèse d'état, USMG/INPG, Grenoble, France, Juin 1984.

[JOH 79] W. JOHNSON
"Behavioral-level test development"
Proc. 16th DAC, pp 171-179, 1979.

[KIN 76] J.C. KING
"Symbolic execution and program testing"
Comm. of ACM, vol. 19 n° 7, pp 385-394, Juillet 1976.

[KOH 70] S. KOHAVI
"Switching automata theory"
Mc Graw Hill Ed., Computer Science Series, Chap. 13, 1970.

[LAI 83] K.W. LAI
"Functional testing of digital systems"
Proc. of the 20th DAC, 1983.

[LAU 82] J.L. LAURIERE
"Représentation et utilisation des connaissances"
Techniques et Science Informatique, n° 1 et 2, 1982.

[LAU 84] J. LAURENT
"Analyse des cirucits intégrés par microscopie électronique"
Thèse de Docteur de l'INPG, ,Institut National Polytechnique de Grenoble, Octobre 1984.

[LEV 82] Y.H. LEVENDEL, MENON
"Test generation algorithms for non procedural CHDL"
IEEE Trans. on. Comp., vol C-31, n°7, Juillet 1982.

[LIN 84] T. LIN, Y. SU
"Functional test generation of digital LSI/VLSI systems using machine symbolic execution technique"
Int. Test Conf., Philadelphia, 1984.

[MAY 83] **D. MAY**
"OCCAM"
SIGPLAN Notices, Vol.18, n°4, Avril 1983.

[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 de matériel : communication personnelle"
Lab. Genie Informatique, IMAG, Grenoble, 1985.

[NIC 80] **V. NICKEL**
"VLSI - The inadequacy of the stuck-at fault model"
IEEE Test Conf., 1980.

[OCC 83] **OCCAM Programming Manual**
INMOS Limited, 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 830 **R. PILOTY, et al.**
"CONLAN report"
Ed. Springer-Verlag, 1983.

[PIL 85] **R. PILOTY et al.**
"The CONLAN project, concepts, implementations and applications"
IEEE Computer, pp. 81-92, Février 1985.

[QUI 83] **P. QUINTON**
"Algorithmes systoliques : de la théorie à la pratique"
Publication interne n° 196, IRISA, Rennes, France, Mars 1983.

[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 83] **F.J. RAMMIG**
"Hierarchical modular description of VLSI system"
pp. 112-116, 1983.

[RAR 83] **J.RARIVOMANANA et al.**
"CADOC : a functional specification tool"
Int. Conf. on Computer Aided Design (ICCAD 83), pp. 65-66,
Santa Clara, USA, Septembre 1983.

[RAR 84a] **J. RARIVOMANANA, et al.**
"Dynamic test specifications for VLSI : CADOC system"
4th Int. Conf. on Reliabilty and maintainability, pp 525-531,
PERROS-GUIREC, France, Mai 1984.

[RAR 84b] **J. RARIVOMANANA 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.

[RAR 85] **J. RARIVOMANANA, et al.**
"CADOC System : a tool for multilevel description and test
generation for VLSI circuits"
Proc. of 7th Int. Conf. on CHDL, Tokyo, Japon, Août 1985.

[ROB 83] **C. ROBACH, P. MALECHA**
"A Computer Aided Test analysis systems : CATA"
IEEE Fault Tolerant Computing Symp., MILAN (Italie), Juin 1983.

[ROB 85] **C. ROBACH, G. SAUCIER**
"Le test et la testabilité des circuits intégrés et systèmes"
Fascicule II : Les études de testabilité - Modules B, Formation
continue, INPG-ENSIMAG, Avril 1985.

[ROI 84] **C. ROISIN**
"La description des protocoles de communication avec le langage
parallèle CSP".
Thèse de Docteur Ingénieur, INPG, Grenoble, France,
Octobre 1984.

[SAU 72] **C. SAUCIER**
"Recherche d'une séquence de test d'une machine séquentielle"
Revue RAIRO n° J-1, 1972.

[SAU 80] **C. SAUCIER, G. ROBACH**
"Microprocessor functional testing"
Int. Test Conf., 1980.

[SAU 81] **G. SAUCIER**
"Les perspectives dans le domaine du test et de la testabilité des circuits à très haute integration"
Rapport de recherche n)268, IMAG, Grenoble, Octobre 1981.

[SEL 68] **F.F. SELLERO, M.Y. HSIAO, L.W. BEARNSON**
"Analysis errors with the boolean difference"
IEEE Trans. on Comp., vol C-17, pp 676-683, 1968.

[SHA 85] **M. SHAHDAD et al.**
"VHSIC hardware description language"
IEEE Computer, pp. 94-102, Février 1985.

[SHE 74] **J.J. SHEDLETSKY**
"A probabilistic treatment of sequential circuits"
Technical Note, Stanford Univ., Fevrier 1974.

[SHE 76] **J.J. SHEDLETSKY, E.J. Mc CLUSKEY**
"The error latency of a fault in a sequential digital circuit"
IEEE Trans. on Comp., pp 655-659, Juin 1976.

[SHE 77] **J.J SHEDLETSKY**
"Random testing : practicality vs. effectiveness"
Fault Tolerant Computing Symp., FTCS-7, LOS ANGELES, Californie, pp 175-179, Juin 1977.

[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.

[SON 82] **K. SON, J. FONG**
"Automatic behavioral test generation"
Digest of papers, Int. Test Conf., 1982.

[SU 82] **Y. SU, HIESH**
"Testing functional faults in Digital Systems"
Journal of Digital Systems, vol 6, n°2, pp 161-183, 1982.

[SOU 83] **J.R. SOUTHARD**
"Mac Pitts : an approach to silicon compilation"
Computer, pp. 74-82, decembre 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.

<u>**A U T O R I S A T I O N de S O U T E N A N C E**</u>

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974

VU les rapports de présentation de

. Madame G. SAUCIER, Professeur
. Monsieur G. MICHEL, Chef de département AMS, CNET

Monsieur RARIVOMANANA Jens Arivelo

est autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité "Informatique".

Fait à Grenoble, le 6 novembre 1985

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble