Génération de test fonctionnel de circuits digitaux décrits avec un langage déclaratif: Lustre

Mazen Al Mahrous

THESE

présentée par

Mazen Al MAHROUS

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

GENERATION DE TEST FONCTIONNEL DE CIRCUITS DIGITAUX DECRITS AVEC UN LANGAGE DECLARATIF : LUSTRE

date de soutenance : 2 Juillet 1990

Composition du jury :

Guy MAZARE	(Président)
Dominique BORRIONE	(Rapporteur)
Christian LANDRAULT	(Rapporteur)
Catherine BELLON	(Directeur de thèse)

Thèse préparée au sein du Laboratoire de Génie Informatique (Unité Génie Matériel)

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

46 avenue Felix Viallet
38031 GRENOBLE cedex

Tél. : 76.57.45.00

Année universitaire 1989

Président de l'Institut :
Monsieur Georges LESPINARD

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	INPG
BLIMAN Samuel	ENSERG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BONNET Guy	ENSPG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHERUY Arlette	ENSIEG
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COLINET Catherine	ENSEEG
CORNUT Bruno	ENSIEG
COULOMB Jean-Louis	ENSIEG
COUMES André	ENSERG
CROWLEY James	ENSIMAG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPEY Maurice	ENSERG
DEPORTES Jacques	ENSPG
DEROO Daniel	ENSEEG
DESRE Pierre	ENSEEG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSPG
FAUTRELLE Yves	ENSHMG
FOGGIA Albert	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GENTIL Sylviane	ENSIEG
GREVEN Hélène	IUFA
GUEGUEN Claude	ENSIEG
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOST Rémy	ENSPG
JOUBERT Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LADET Pierre	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LORET Benjamin	ENSHMG
LOUCHET François	ENSEEG
LUCAZEAU Guy	ENSEEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOHR Roger	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PA ULEAU Yves	ENSEEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
PIC Etienne	ENSERG
PLATEAU Brigitte	ENSIMAG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
REINISCH Raymond	ENSPG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrièle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SERMET Pierre	ENSERG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VINCENT Henri	ENSPG
ZADWORNY François	ENSERG

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
COURNIL M.
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LACHENAL D.
LADET Pierre
LATOMBE Claudine
LE HUY H.
LE GORREC Bernard
MADAR Roland
MEUNIER G.
MULLER Jean
NGUYEN TRONG Bernadette
NIEZ J.J.
PASTUREL Alain
PLA Fernand
ROGNON J.P.
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri
YAVARI A.R.

Chercheurs du C.N.R.S

DIRECTEURS DE RECHERCHE CLASSE 0

LANDEAU	Ioan
NAYROLLES	Bernard

Directeurs de recherche 1ère Classe

ANSARA Ibrahim
CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
KRAKOWIAK Sacha
LEPROVOST Christian
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ARMAND Michel
AUDIER Marc
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
CHATILLON Chritiant
CLERMONT Jean-Robert
COURTOIS Bernard
DAVID René
DION Jean-Michel
DRIOLE Jean
DURAND Robert
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GARNIER Marcel
GUELIN Pierre
JOUD Jean-Charles
KAMARINOS Georges
KLEITZ Michel
KOFMAN Walter
LEJEUNE Gérard
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MEUNIER Jacques
PEUZIN Jean-Claude
PIAU Monique
RENOUARD Dominique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
VENNEREAU Pierre
WACK Bernard
YONNET Jean-Paul

Personnalités agrées à titre permanent à diriger des travaux de recherche
(décision du conseil scientifique)

E.N.S.E.E.G
HAMMOU Abdelkader
MARTIN-GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G
BOREL Joseph

E.N.S.I.E.G
DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.M.G
ROWE Alain

E.N.S.I.M.A.G
COURTIN Jacques

C.E.N.G
CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIEB Maurice
VINCENDON Marc

Laboratoires extérieurs :

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Situation particulière

PROFESSEURS D'UNIVERSITE

DETACHEMENT

ENSIMAG	LATOMBE	J..Claude	Détachement	21/10/1989
ENSHMG	PIERRARD	J.Marie	Détachement	30/04/1989
ENSIMAG	VEILLON	Gérard	Détachement	30/09/1990
ENSIMAG	VERJUS	J.Pierre	Détachement	30/09/1989
ENSPG	BLOCH	Daniel	Recteur à c/	21/12/1988

SURNOMBRE

INPG	CHIAVERINA	Jean	30/09/1989
ENSHMG	BOUVARD	Maurice	30/09/1991
ENSEEG	PARIAUD	J.Charles	30/09/1991

à la mémoire de ma mère et à mon père

à ma femme et mes sœurs

à mes collègues et amis

Remerciements

Je tiens à exprimer toute ma reconnaissance à Madame **Catherine BELLON**, Maître de conférence à l'ENSIMAG, pour avoir assuré l'encadrement scientifique de mon travail et pour m'avoir fait partager son enthousiasme pour la recherche et m'avoir aidé pendant toutes ces années.

Je tiens à remercier Monsieur **Guy MAZARE**, Professeur à l'ENSIMAG, de m'avoir fait l'honneur de présider le jury de cette thèse.

Je tiens également à remercier Madame **Dominique BORRIONE**, Professeur à l'UJF, et Monsieur **Christian LANDRAULT**, directeur de recherche au CNRS, d'avoir accepté d'être rapporteurs de cette thèse.

Je témoigne tout particulièrement ma gratitude à Monsieur Paul CASPI, chargé de recherche au CNRS et responsable de l'Unité Génie Matériel du Laboratoire de Génie Informatique, pour ses conseils constructifs qui me furent très précieux et pour les nombreuse discussions, aussi sympathiques qu'enrichissantes, que j'ai eues avec lui.

Je voudrais également remercier :

- Monsieur Nicolas HALBWACHS pour sa lecture attentive de ma thèse et pour ses conseils, Monsieur Raoul VELAZCO et Madame Chantal ROBACH, responsables au LGI-UGM.

- Messieurs Antoine PROVOST, Pierre WODEY et Imad SABOUNI, collègues du LGI, pour les discussions scientifiques.

- Madame Solange ROCHE pour ses corrections de mes erreurs de français ainsi que pour son extrême sympathie.

- tous mes collègues et amis de l'équipe "Unité Génie Matériel du LGI pour l'ambiance agréable qu'ils y ont fait régner.

Résumé

La complexité croissante des circuits intégrés rend difficile la génération du test de tels circuits. Une des solutions est la génération du test à partir d'une description de haut niveau.

Une approche fondée sur un modèle du circuit à tester décrit dans le langage déclaratif de haut niveau LUSTRE est proposée. Cette approche est basée sur la traduction des différents opérateurs primitifs de ce langage en graphe SATAN pour l'évaluation de la testabilité, l'analyse de test ainsi que la détermination des séquences de test d'un circuit.

Une méthode de test par identification de la partie contrôle à travers la partie opérative des circuits complexes est ensuite définie. Cette méthode utilise les automates générés à partir d'une description d'un circuit composé d'une partie opérative et d'une partie contrôle en langage LUSTRE.

Mots clés

Test fonctionnel, Langages de description de matériel, Langage déclaratif, Testabilité, Ecoulements, Test par identification d'automates.

Abstract

The increasing complexity of integrated circuits have resulted in making their test a difficult task. Numerous methods propose the use of circuit high-level description as a basis for test generation.

We propose a functional approach to the test generation problem starting from a high-level description. The circuit under test is modeled using the LUSTRE high-level data-flow description language. The different LUSTRE primitives are then translated to a SATAN-format graph in order to evaluate the testability of the circuit and to generate the test sequences.

Another method for the test of the complex circuits composed of an operative part and a control part is also defined. It consists of checking experiments for the control part observed through the operative part. It has been applied to the automata generated from a LUSTRE description of the circuit.

Key words

Functional test, Hardware description languages, Data-flow languages, Testability, Information path, Checking experiments.

TABLE DES MATIERES

DEUXIEME CHAPITRE : ANALYSE DE TESTABILITE A PARTIR D'UNE DESCRIPTION EN LUSTRE 81

AVANT PROPOS

Les récents progrès dans le domaine de la technologie des circuits intégrés ont permis de développer des circuits LSI/VLSI très complexes réalisant souvent des fonctions sophistiquées. La complexité croissante de tels circuits rend difficile leur conception et leur description ainsi que la génération des séquences qui permettent de les tester.

Pour aborder ce problème, de nombreuses recherches se sont orientées vers la description et génération de test à haut niveau. Nous proposons l'utilisation du langage LUSTRE, qui permet de décrire le matériel d'une façon plus proche de son fonctionnement réel, pour la description des circuits et la génération de leur test.

Cette thèse se divise en trois chapitres, tous trois sont basés sur l'utilisation du langage de haut niveau, déclaratif : LUSTRE .

Le premier chapitre est une étude de l'utilisation du langage LUSTRE en tant que langage de description de matériel. Les caractéristiques de ce langage seront présentées ; ses capacités pour la description de matériel seront étudiées tout en mettant en évidence les extensions à effectuer.

Le deuxième chapitre suggère une extension du système d'évaluation de testabilité SATAN. Nous proposons d'effectuer une traduction automatique et directe des différentes primitives LUSTRE en graphe de testabilité SATAN pour l'évaluation de la testabilité du circuit. Une extension du graphe SATAN obtenu est introduite pour prendre en compte la notion de temps. On obtient un graphe de testabilité temporisé qui permet de déterminer les différents instants d'activation du circuit nécessaires pour l'établissement des séquences de test.

Le troisième chapitre présente une extension des méthodes de test par identification d'automates au test de circuits complexes : test de la partie contrôle du circuit à travers sa partie opérative. Les définitions des différentes séquences de test élémentaires nécessaires à la construction d'un test par

identification seront étendues, et des algorithmes de recherche de ces séquences seront définies. Cette méthode s'applique aux automates générés à partir d'une description LUSTRE d'un circuit ; un logiciel de détermination des séquences de test élémentaires a été réalisé.

CHAPITRE 1

DESCRIPTION DE MATERIEL PAR UN LANGAGE

HAUT NIVEAU DECLARATIF : LUSTRE

INTRODUCTION

Ce chapitre a pour but d'étudier l'utilisation du langage LUSTRE en tant que langage de description de circuits ou systèmes digitaux.

LUSTRE est un langage de programmation à flots de données (data-flow), déclaratif, synchrone. Il a été conçu pour programmer des systèmes parallèles temps réel. Il possède une sémantique mathématique simple et formelle qui permet de minimiser le risque de mauvaise compréhension et qui permet de prouver des programmes écrits en LUSTRE. Dans [PIL 86], il est montré que tout programme LUSTRE manipulant des booléens peut être traduit en une formule de logique temporelle équivalente et donc que l'équivalence de deux programmes LUSTRE (donc éventuellement de deux niveaux de description de circuits) peut être prouvée.

Bien que ce langage ait été conçu pour la programmation de systèmes temps réel, LUSTRE peut être utilisé pour la description de circuits. La description de circuits se heurte à deux principaux problèmes : d'abord, il est souvent nécessaire d'utiliser plusieurs langages suivant le niveau d'abstraction pour décrire un circuit, ce qui va créer des problèmes de preuves lors du passage d'un niveau à un autre. Ensuite, le choix d'un langage de description se heurte à un dilemme entre des langages "confortables" (mais rendant difficiles les manipulations formelles de programmes) et des langages plus formels (mais dont l'usage demande un savoir faire certain). Il semble que LUSTRE permette de résoudre ces deux problèmes.

On discutera d'abord le problème de description de matériel en définissant certains termes utilisés dans ce domaine, puis les langages de description de matériel (HDL "Hardware Description Language") en mettant l'accent sur le langage VHDL. VHDL, ayant été défini pour constituer un standard, présente des caractéristiques très intéressantes dans le domaine de description de matériel. Dans les années qui vont suivre, il jouera un rôle crucial pour le développement des systèmes électroniques [MEN 89].

On présentera le langage LUSTRE en définissant les variables, les équations, les types, les nœuds et les assertions du langage. On étudiera

ensuite les capacités du langage LUSTRE : étant un langage de type déclaratif, il permet d'exprimer le parallélisme intrinsèque aux circuits et de modéliser les aspects temporels du fonctionnement des circuits. On présentera ensuite les caractéristiques du langage tout en mettant en évidence les extensions à effectuer pour que LUSTRE soit d'une utilisation aisée et s'adapte mieux au problème de description de circuits.

I - LES LANGAGES DE DESCRIPTION DE MATERIEL

I. 1 - Description de matériel - Définitions

Avant d'aborder les langages de description de matériel, il faut définir quelques termes particuliers utilisés dans ce texte.

Un ***matériel*** est un ensemble de composants électroniques digitaux interconnectés.

Les circuits digitaux d'une façon générale peuvent être classés en circuits combinatoires et circuits séquentiels.

Les valeurs de sortie d'un ***circuit combinatoire*** dépendent des valeurs présentes des entrées (bien qu'un changement de valeurs des entrées ne soit pas reflété instantanément sur les sorties, en raison du retard introduit par les éléments du circuit).

Les valeurs de sorties d'un ***circuit séquentiel*** dépendent des valeurs d'entrées présentes et passées.

Les circuits séquentiels peuvent être :

* des circuits synchrones dont l'évolution est synchronisée par horloges ;
* des circuits asynchrones dont l'évolution est dirigée par les changements de valeurs des entrées.

Une ***description*** d'un circuit est un modèle représentant le circuit différent suivant le domaine et le niveau de la description.

I. 1. 1 - Domaines de description

Le matériel peut être décrit selon trois domaines : comportemental, structurel et physique.

- Le domaine comportemental (fonctionnel) dans lequel le circuit est modélisé comme une "boite noire" en décrivant ses sorties comme une fonction de ses entrées et du temps. Cette fonction est représentée de manière abstraite.

- Le domaine structurel dans lequel le circuit est représenté en terme d'un ensemble de composants interconnectés.

- Le domaine physique dans lequel le circuit est représenté comme un assemblage de figures et de formes géométriques sans comportement associé.

Nous nous intéressons aux domaines comportemental et structurel seulement. Pour chacun de ces domaines, on peut distinguer un certain nombre de niveaux. Par ordre d'abstraction croissant, on peut définir les niveaux suivants : électrique, logique, transfert de registre, sous-système et système.

I. 1. 2 - Niveaux de description

Dans la littérature, la définition de niveau de description est différente suivant les auteurs. Cette diversité reflète le fait que chacun s'est attaché à résoudre une classe de problèmes bien précise. On remarque que plusieurs auteurs font un mélange entre le niveau et le type de description. Barbacci [BAR 75] distingue trois types de description : comportemental, fonctionnel et structurel ; parallèlement à cette classification, l'auteur distingue entre ce qu'il appelle une description procédurale et une description non-procédurale.

A une ***description procédurale***, composée d'un ensemble d'instructions, correspond l'exécution en série de ces instructions suivant leur ordonnancement. Une ***description non-procédurale*** n'implique pas de séquentialité implicite.

Thomas [THO 81] considère qu'un circuit est décomposé en une partie opérative (PO) et une partie contrôle (PC) et peut être représenté aux niveaux suivants : logique, fonctionnel et comportemental tout en distinguant la partie opérative et la partie contrôle. Thomas conclut que pour une représentation au niveau comportemental, il peut exister plusieurs représentations au niveau fonctionnel et pour chacune de ces dernières il peut exister plusieurs réalisations au niveau logique. Borrione constate qu'il existe un nombre à priori aussi grand que l'on veut de niveaux de description [BOR 81]. Cette idée a influencée la réalisation du simulateur multi-niveaux du projet CASCADE [BOR 85].

En ce qui concerne les niveaux de description, nous les définirons comme suit :

a) Niveau électrique : une description comportementale au niveau électrique est représenté par une fonction continue du temps qui est définie au travers d'un ensemble d'équations différentielles. Une description structurelle est définie par assemblage d'éléments du type transistor, résistance, capacité.

b) Niveau logique : une description comportementale au niveau logique est une fonction discrète du temps et peut s'exprimer sous forme d'un ensemble d'équations booléennes temporisées. Une description structurelle représente un circuit comme assemblage de portes logiques et d'interrupteurs ("switch", AND, OR, NAND, . . .).

c) Niveau transfert de registre : une description comportementale d'un circuit à ce niveau s'exprime à partir des opérations de manipulations sur un ensemble de variables qui sont associées aux points de mémorisation du circuit (un transfert est décrit par une affectation). Une description structurelle est définie à partir d'éléments du type registre, multiplexeur, unité arithmétique et logique (UAL), . . .

d) Niveau sous-système : une description comportementale à ce niveau est décrit par un algorithme qui est exprimé en termes d'opérations de manipulation de données. Une description structurelle se ferait en termes des modules de matériel.

e) Niveau système : une description comportementale à ce niveau est spécifiée par les fonctionnalités et les performances du système (pour les microprocesseurs, par exemple, un jeu d'instructions). Une description structurelle se ferait en termes d'éléments complexes du type mémoire, CPU "Central Processing Unit", contrôleur, . . .

Les trois domaines sont représentés en fonction de leurs niveaux respectifs dans la figure I-1 [MAR 83].

domaine / niveau	domaine comportemental	domaine structurel	domaine physique
niveau système	performances jeu d'instructions	CPU, mémoires, contrôleur, . . .	partitions physiques
niveau sous-système	algorithmes (manipulation de données)	modules de matériel	groupes
niveau transfert de registre	opérations transfert de registre	ALU, MUX, registres	plan de masse
niveau logique	équations booléennes	portes logiques	topologie masque cellule
niveau électrique	équations différentielles	transistor, résistance, capacité	

Figure I-1 : Les niveaux de descriptions suivant le domaine.

En fait, les frontières entre les différents niveaux ne sont pas bien définies, d'autant plus que les concepteurs de langages de description de

matériel ont pour but de les rendre les plus souples possible, donc les plus multi-niveaux possibles (par exemple le langage VHDL peut s'utiliser du niveau logique au niveau système).

Les niveaux de description des circuits : système, sous-système, RTL, logique, électrique et silicium, peuvent être représentés par une pyramide descriptive [ARM 88]. Le sommet de cette pyramide correspond au niveau système.

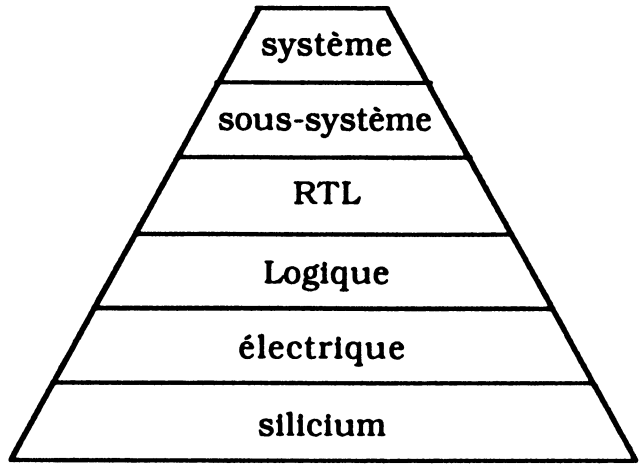


Figure I-2 : Niveaux de description des systèmes digitaux.

I. 1. 3 - Description hiérarchisée

Une ***description structurelle*** à un certain niveau établit la connectique entre des éléments matériels. Un matériel peut être décrit d'une façon structurelle à différents niveaux suivant la complexité de ses composants.

Une ***description fonctionnelle ou comportementale*** établit une relation fonctionnelle entre les valeurs des signaux de sorties obtenues et les valeurs des signaux d'entrées présentes et passées. Cette description considère un ensemble matériel comme une "boite noire" réalisant un ensemble d'actions à des instants précis. Cette description n'implique pas une architecture spécifique du circuit et elle est basée sur une abstraction. Une description comportementale peut être impérative ou déclarative.

Une ***description hiérarchisée*** est définie en décrivant un circuit comme un assemblage de sous-descriptions, elles-mêmes étant décrites hiérarchiquement ou non. Si les sous-descriptions correspondent à des niveaux différents d'abstraction, nous parlerons de composition hiérarchisée multi-niveau.

Il existe trois types de hiérarchies :

* hiérarchie structurelle
* hiérarchie comportementale
* hiérarchie mixte (structurelle et comportementale)

a) hiérarchie structurelle :

Dans cette hiérarchie, la description interne d'un élément matériel est constituée par l'interconnexion de sous-éléments matériels, eux-mêmes étant constitués par l'interconnexion de sous-éléments qui peuvent être les primitives du langage suivant le niveau d'abstraction. La "mise à plat" d'une hiérarchie structurelle permet de représenter des réalisations très complexes en utilisant des primitives d'un très bas niveau.

Par exemple, un circuit peut être décrit par un assemblage de modules matériels de haut niveau, comme des registres, des additionneurs, des multiplieurs et des multiplexeurs, chaque module étant décrit par un assemblage de porte logiques constituantes.

b) hiérarchie comportementale :

Une hiérarchie comportementale repose sur la notion d'action réalisées par le matériel à des instants donnés. Dès lors, la décomposition comportementale n'a pas la même sémantique qu'une décomposition hiérarchique structurelle .

Deux problèmes principaux doivent notamment être résolus :

- les conditions d'activité ou d'inactivité d'un bloc
- les protocoles régissant les relations entre les blocs.

La communication et la synchronisation entre blocs se fait par des approches différentes. Une hiérarchie comportementale est spécifiée par exemple par l'utilisation de processus concurrents, qui consiste à considérer les blocs de description comme un ensemble de ressources concurrentes. Comme langages représentatifs de ce type de hiérarchie, on peut citer OCCAM [MUN 83] et ADLIB [HIL 80] qui est une extension du langage PASCAL.

c) hiérarchie mixte :

On a défini une description structurelle et une description comportementale. Entre ces deux extrêmes, il y a toutes les phases intermédiaires mixtes, où des informations structurelles et comportementales peuvent coexister dans une même description.

En effet, deux types de représentations mixtes de structure et de comportement peuvent exister :

1) une hiérarchie mixte de description peut contenir des blocs comportementaux et des blocs structurels. Un bloc structurel peut faire appel à des blocs de comportement et/ou de structure. La figure I-3 montre un exemple de cette hiérarchie. On constatera qu'un bloc comportemental constitue une feuille.

Plusieurs langages utilisent ce type de hiérarchie comme le langage MODLAN. Les blocs comportementaux, dans ce type de hiérarchie, représentent des parties qu'on n'a pas encore conçu dans les détails, ou qu'on ne veut pas décrire de façon plus détaillée.

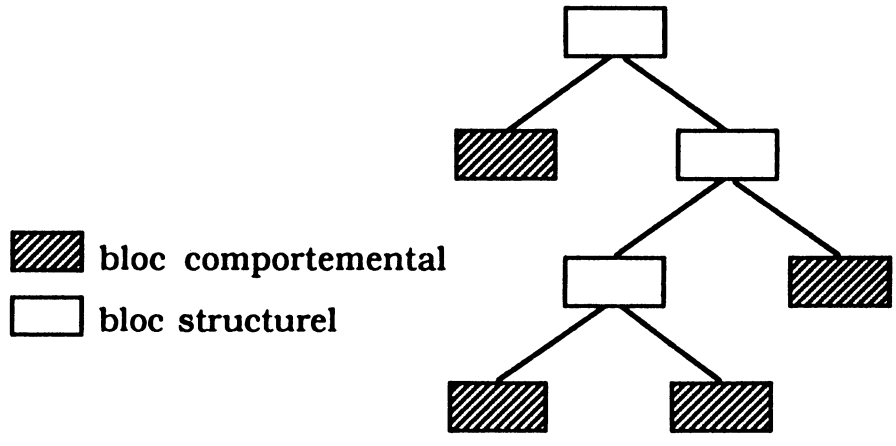


Figure I-3 : Hiérarchie mixte du type 1.

2) une hiérarchie mixte où à chaque bloc structurel est associé un autre bloc comportemental, figure I-4. Ce type de hiérarchie est utilisé par le système SABLE [HIL 79] avec les deux langages ADLIB (pour décrire le comportement) et SDL (pour décrire la structure).

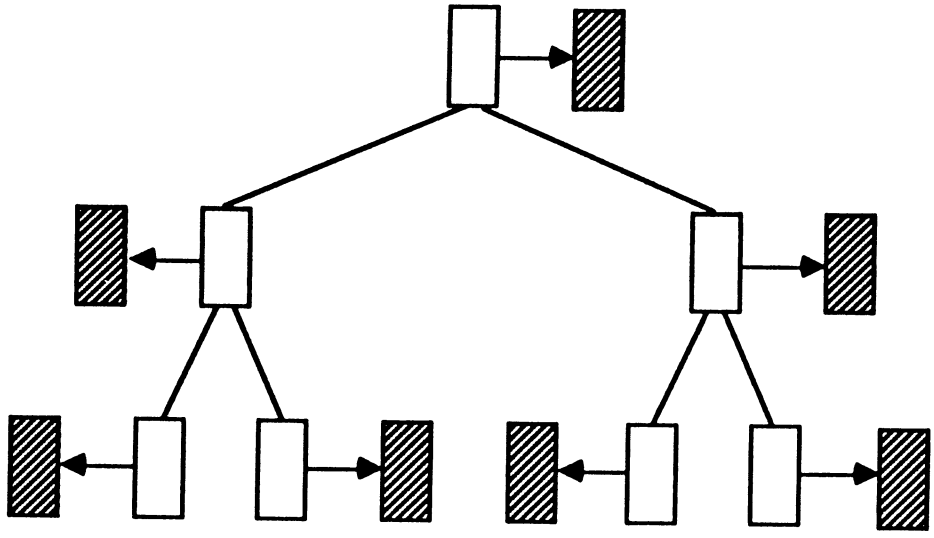


Figure I-4 : Hiérarchie mixte du type 2.

L'utilisation de ce type de hiérarchie est fondamentale pour une conception de type "top-down" en utilisant des raffinements progressifs pour les différents étapes de conception.

Le langage VHDL a généralisé cette idée par la possibilité d'avoir plusieurs ***alternatives de conception*** (différentes versions de description) comportementales et/ou structurelles d'un bloc, c'est à dire plusieurs descriptions d'un même bloc matériel qui coexistent dans la description d'un ensemble.

I. 2 - Les langages de description de matériel

Les premières tentatives de définition d'un langage de description du matériel (ou HDL "Hardware Description Language") ont commencé vers le milieu des années soixante [CHU 65]. Pendant la vingtaine d'années suivantes plusieurs propositions ont été formulées.

Pour offrir le maximum de souplesse d'utilisation, un langage de description doit permettre une hiérarchie mixte aux différents niveaux d'une conception, spécifiant la structure des composants et leurs comportements. Les primitives de haut niveau de ce langage peuvent être construites sans être reliées aux primitives de bas niveau. Ce langage doit permettre une simulation de ***niveau mixte*** qui permet de simuler un mélange d'éléments aux différents niveaux d'abstraction et de simuler des éléments combinatoires et séquentiels dans le même modèle.

Certains langages de description du matériel offrent un jeu de primitives couvrant plusieurs niveaux, permettant le passage d'un niveau à un autre. Ceci a pour conséquence, d'avoir plusieurs niveaux d'abstraction du matériel décrit par des décompositions hiérarchiques de plus en plus fines.

Parmi les langages de description de matériel, on peut citer les langages impératifs comme SDL et ADLIB [COR 81].

Une extension des langages parallèles comme OCCAM [MUN 83], ADA [BAR 84] est proposé aussi afin de les utiliser dans le domaine de description de matériel. L'utilisation des langages déclaratifs dans ce domaine est complètement justifiée par la nature parallèle de matériel en mettant l'accent sur la propriété data-flow de ces langages ; par exemple le langage LTS [MIL 84] (LTS est basé sur les mêmes idées que le langage LUSTRE, notion d'histoire d'une variable, sémantique formelle, . . .).

Des langages spécifiques à la description de matériels sont proposés, on peut citer les langages CADOC [CRA 85], [BEL 84], IRENE [MAR 86] et CONLAN [PLT 85].

CADOC est un langage destiné à la description de circuits et à leur simulation. Il permet d'effectuer une description hiérarchisée multi-niveaux, une description temporelle très précise même à un haut niveau d'abstraction (en utilisant un graphe interprété et temporisé GIT et en utilisant des échéanciers) et une description modulaire pour pouvoir partitionner le circuit en blocs dont la description sera de plus en plus détaillé à chaque évolution dans la conception. Le langage CADOC ne possède aucun élément prédéfini ; les concepteurs peuvent définir librement tous les éléments qu'ils utilisent. Les descriptions peuvent être compilées séparément ce qui permet la création

d'une bibliothèque. Les descriptions peuvent aussi être paramétrées ; lors de l'utilisation d'une telle description, on doit fixer les valeurs de ses paramètres.

IRENE est un langage pour la description, la simulation et la synthèse de matériel. Il est l'un des outils constituant le système KARENE de CAO. Il permet d'effectuer une description du comportement et/ou de la structure d'un circuit. Une description comportementale IRENE-C et une description structurelle IRENE-S peuvent être regroupées en une seule description dite mixte [MAR 83].

CONLAN définit un ensemble de langages, chaque langage est utilisé pour un niveau d'abstraction donné ce qui permet de décrire le matériel à plusieurs niveaux d'abstraction (comportemental et structurel) et de vérifier la cohérence entre ces descriptions. Chaque nouveau langage est défini à partir d'un langage déjà existant. Le langage racine de CONLAN nommé BASE CONLAN (ou BCL) a été formellement défini à partir d'un ensemble de primitives appelé PRIMITIVE SET CONLAN (PSCL). Ce noyau commun permet le développement de différents langages de description de matériels ayant tous la même syntaxe.

La prolifération des propositions de langages de description a mené vers une tentative de normalisation de ces langages. En 1986, le comité de l'IEEE propose l'adoption de VHDL comme un langage standard. VHDL est un langage puissant dans le domaine de description de matériel à l'instar de ADA dans le domaine de la programmation.

I. 3 - Le langage VHDL

Le langage VHDL a été développé sur la demande du département de la défense américaine. Des efforts ont été effectués récemment par le comité de l'IEEE pour l'adoption de VHDL comme un langage standard pour la description de matériels. VHDL est le premier langage de description de matériels qui utilise le concept de package proposé pour le langage ADA. L'élément de base d'une description VHDL est l'entité de conception (Design Entity).

Le langage VHDL permet de décrire les circuits digitaux du niveau système au niveau logique (mais pas au niveau électrique). Il permet d'effectuer toute description intermédiaire entre la description comportementale et la description structurelle. Cette description permet de spécifier la structure de chemin de données ainsi que le comportement de flux de contrôle (en utilisant des processus) [WAX 86].

VHDL permet la description de circuits combinatoires et séquentiels (synchrones ou asynchrones). Il permet d'effectuer quatre types de description [LIS 89] : description des circuits combinatoires, description fonctionnelle, description au niveau transfert de registre (ensemble d'instructions de contrôle ou de chemin de données) et description algorithmique (un processeur par exemple).

Il permet d'effectuer une conception modulaire et un partitionnement en entités de conception et leurs assemblages, ce qui permet la création d'une bibliothèque et la réutilisation de ces entités dans tous les niveaux d'abstraction. En plus, il possède des possibilités d'abstraction de données et permet la définition, par l'utilisateur, de nouveaux types de données.

Le langage VHDL permet la conversion de type des variables et le changement de niveau de description contrôlé par l'utilisateur. Il est le premier langage de description de matériel qui permet de créer des alternatives de conception (différentes versions de description) pour une interface unique explicite [AYL 86].

I. 3. 1 - L'architecture d'un système VHDL

Le système VHDL proposé en [LOU 87] est un système ouvert. Il est organisé de façon à permettre l'ajout d'autres outils de CAO. La figure I-5 représente son environnement. Il est constitué de quatre unités principales : le compilateur ou analyseur, le simulateur, l'analyseur inverse et le simplificateur.

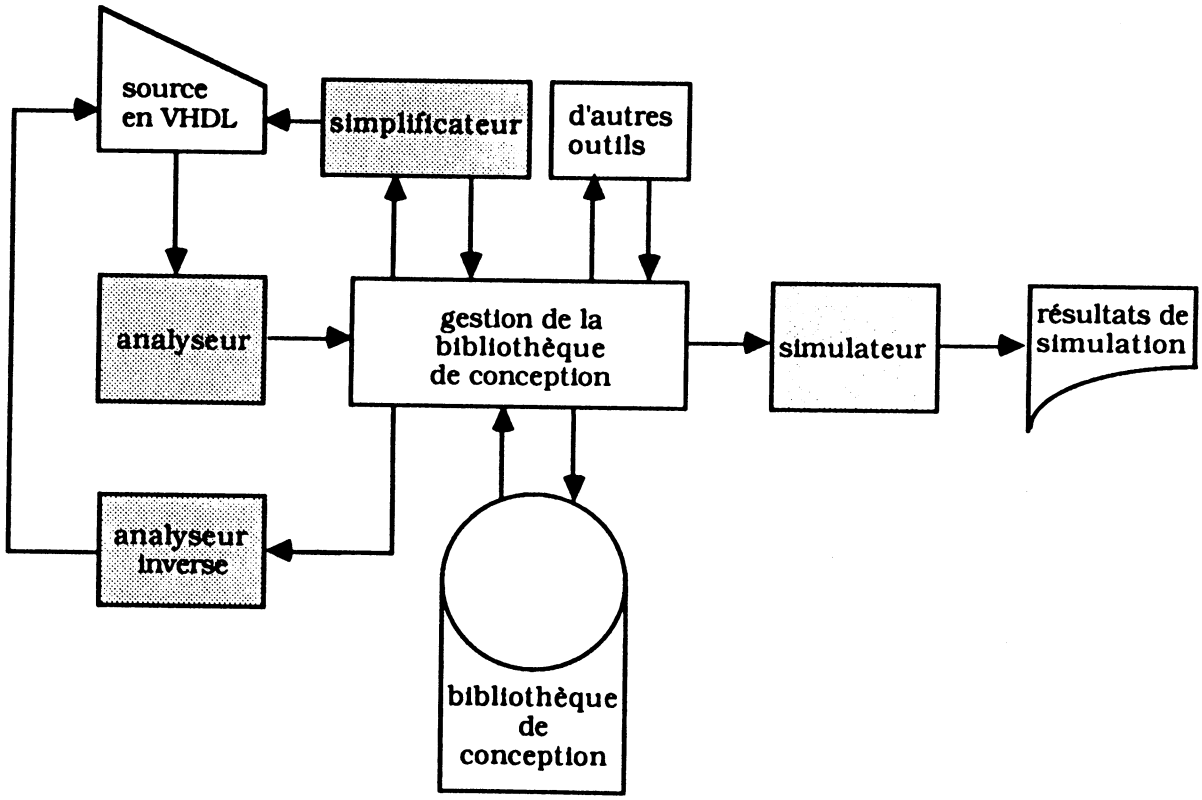


Figure I-5 : L'environnement de VHDL.

L'analyseur a comme rôle la compilation des programmes sources VHDL en format intermédiaire. L'analyseur inverse établit les sources VHDL équivalents à partir de données en format intermédiaire. Le simplificateur a le rôle de fusionner les modules de descriptions de matériels en une description unique, en mettant à plat une description hiérarchisée, figure I-6.

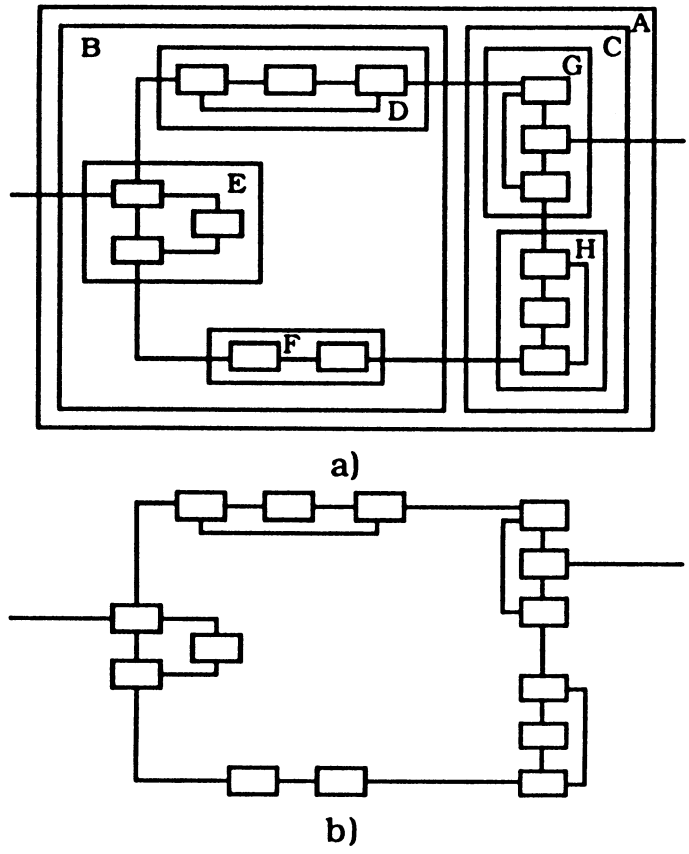


Figure I-6 : a) modules hiérarchisés de l'entité A ;
b) description simplifiée de A.

En [SHA 85], il a été proposé de gérer la bibliothèque de description d'entité de conception en utilisant un "profiler". Les entités de conceptions sont stockées dans la bibliothèque en forme code intermédiaire. Le "profiler" permet à l'utilisateur de créer un assemblage d'entités (à partir d'entités compilées et disponibles dans la bibliothèque) pour décrire une application précise.

En [SMI 89] est présenté l'outil développée à MCC (Microelectronics and Computer Corporation). Dans le but d'améliorer la rapidité de simulation les descriptions sont compilées en langage "C". Cette démarche ressemble à celle du système LUSTRE (Cf. paragraphe II. 6).

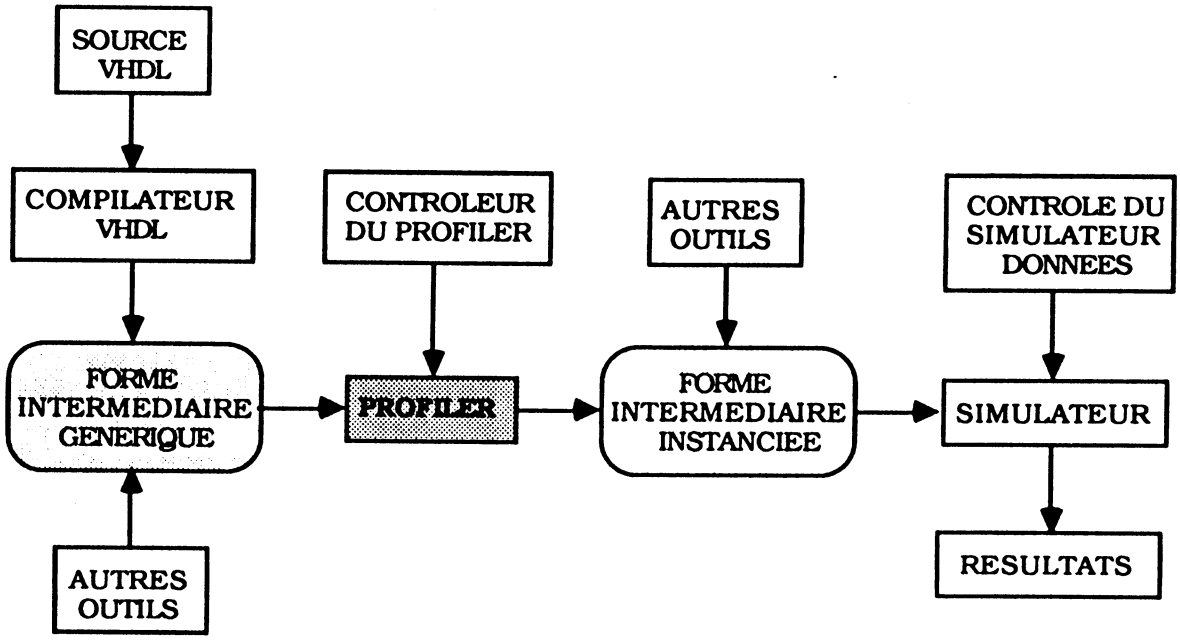


Figure I-7 : Gestion de la bibliothèque VHDL par le "profiler".

I. 3. 2 - Entité de conception

La représentation des alternatives de conception en VHDL (comme d'ailleurs en CONLAN) se fait par un mécanisme spécifique. Le fait qu'une entité puisse posséder différentes descriptions offre un outil intéressant pour une conception descendante aussi bien qu'ascendante.

Une entité de conception est constituée :

- d'un en-tête
- d'un ou plusieurs corps architecturaux.

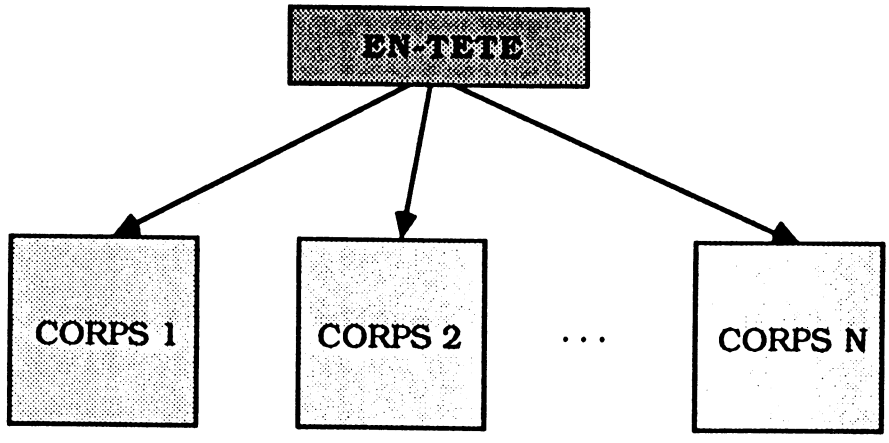


Figure I-8 : Entité de conception.

L'en-tête décrit les ports d'entrée/sortie et peut spécifier des valeurs génériques et des propriétés commmunes à tous les corps. Chaque corps alternatif décrit le module matériel d'une façon différente. Par exemple, un corps peut décrire le comportement du module, un autre décrit sa structure en terme d'interconnexion de ses composants et un troisième donne un modèle des opérations de l'entité en terme de micro-opérations au niveau transfert de registre.

Trois types généraux de description sont possibles dans un corps architectural :

i) Description structurelle

Une description structurelle en VHDL est définie par l'interconnexion d'un ensemble de "sous-composants", dont le comportement est défini à l'extérieur (par d'autre entités), et peut être appliquée à tous les niveaux d'abstraction. Cette description peut utiliser la notion d'attribut. Ces attributs associés à une entité (ou à un ensemble d'entités) peuvent être analysés par le système VHDL. Une description structurelle nous permet donc de décomposer une conception d'une façon hiérarchisée en un ensemble de sous-composants.

Dans une description structurelle, après déclaration des composants utilisés et des signaux internes, il y a "instantiation" des composants et définitions de leurs signaux d'entrée et de sortie.

exemple :

```
architecture STUCTURE of processor is

component ALU_box
    port map(left_side : in bit_vector; bus_out : out bit_vector; . . . );

component C_design
    port map(port_1 : inout bit_vector; port_2 : inout bit_vector; . . . );
```

```
signal data_in_1, data_bus : bus;
.
.
.
begin
ALU : ALU_box port map(data_in_1, data_bus, . . .);
C : C_design port map(data_in_1, data_bus, . . .);
end STUCTURE;
```

ii) description data-flow

Une description data-flow est bien adaptée pour la description de transformations de données en terme d'instructions RTL exécutées en parallèle. Une telle description permet d'exprimer le parallélisme et le comportement simultanément. Les instructions d'affectation des signaux seront prises en compte en parallèle. Une affectation exprime une relation entre les variables sources et la variable destination, les changements de valeurs des variables sources étant éventuellement pris en compte avec un retard. Si on se place à un instant t, les valeurs correspondantes à des instants $t_1 < t$ sont accessibles mais non modifiables, et les valeurs correspondantes à des instants $t_2 > t$ sont uniquement modifiables.

Exemple :

-l'interface de FULL_ADDER

```
entity FULL_ADDER
  (X, Y : in BIT; CIN : in BIT := '0';
   COUT, SUM : out BIT) is
end FULL_ADDER
```

-le corps de la description data-flow de FULL_ADDER

```
architecture DATA_FLOW of FULL_ADDER is
    signal C : BIT;
begin
    SUM<= X xor Y xor CIN after 5 ns;
    C<= (Y and CIN) or (X and CIN) or (X and Y);
    COUT<= C after 6 ns;
end DATA_FLOW;
```

iii) description comportementale

Une description comportementale décrit le matériel d'une façon purement algorithmique (la plus abstraite). Cette description définit la transformation de données en terme d'algorithme pour calculer les réponses en sortie en fonction de changement des valeurs des entrées. Comme une description data-flow, une description comportementale permet la description des relations entre entrées et sorties pour chaque sous composant d'une description structurelle.

Une description comportementale peut être décrite par des blocs (exécutés en parallèle) ou par des processus (exécutés en série) à tous les niveaux (du niveau algorithmique jusqu'au niveau circuit). VHDL peut donc représenter facilement le parallélisme demandé par un circuit en utilisant une description comportementale. Un processus comporte une partie déclaration et une partie instruction. Ce processus est exécuté en série en utilisant les instructions séquentielles **if**, **case** et **loop**. Dans une description comportementale toutes les variables utilisées sont déclarées et typées.

Exemple :

En supposant que MR et M sont déclarés comme suit :

```
signal MR:BIT;
signal M:BIT_VECTOR(1 to 4);
```

un processus qui décrit une RAM est le suivant :

```
begin
RAM : process(MR,M(i))
      type Matrix is array (Natural range <>) of integer;
      static variable Mem : Matrix (0 to 511);
   begin
      if (MR = '1') then enable M(i);
      else disable M(i);
      end if;
      if (MR and M(i)) = '1' then
      case RW is
```

```
        when '0' =>
          DBus<= ReadOut(Mem(IntVal(Addr))) after 70ns;
        when '1' =>
          Mem(IntVal(Addr)):= ReadIn(DBus);
      end case;
      else DBus<= "ZZZZZZZZ" after 55ns;
      end if;
    end process;
end block;
end ONE_IMPLEMENTATION;
```

Cet exemple décrit une RAM par un processus. La fonction IntVal convertit un vecteur de bits non signé en un nombre naturel. On remarque que l'opération d'écriture demande 70 ns, par contre l'opération de lecture est immédiate. Le bus de données doit être en haute-impédance quand la RAM est non sélectionnée. Ceci est réalisé par un vecteur 'ZZZZZZZZ' où Z est une valeur représentant la haute-impédance.

Remarque :

On peut décrire en VHDL des structures régulières en utilisant par exemple un processus plusieurs fois. Ceci se fait en utilisant la clause "generate". Par exemple :

```
for i in 2 to 4 generate
RAM : process(MR, M(i))
.
.
end generate;
```

I. 3. 3 - Caractéristiques temporelles

VHDL permet de décrire les activités synchronisées par une horloge ou par un événement. Il permet la spécification d'un temps de retard où une fonction retard peut être appliquée à un signal. Donc il est possible d'effectuer une description composée par un mélange de circuits indépendants

synchronisés par une horloge ou non. Il satisfaisait toutes les caractéristiques temporelles demandées à tous les niveaux d'abstraction.

Toutes les unités du temps, en VHDL, sont définies comme des multiples d'une horloge de base fs. Avec cette unité du temps l'utilisateur peut spécifier les caractéristiques et les contraintes temporelles de son circuit.

VHDL peut simuler un retard de propagation ainsi qu'un retard inertiel et un retard de transport. La sémantique d'un ***retard inertiel*** implique que la sortie d'une fonction logique ne répondrait pas au signal d'entrée stimulant le modèle si cette dernière a une durée inférieure à une durée spécifique. Par contre, la sémantique d'un ***retard de transport*** implique que la sortie d'une fonction logique répond avec fidélité au signal d'entrée pour n'importe quel signal présent à l'entrée. La possibilité de modéliser un retard inertiel est très importante pour une description au niveau logique. Un retard inertiel donne à l'utilisateur la possibilité d'éliminer certains impulsions qui ont une largeur plus petite qu'une largeur définie (Ceci est très utile pour éliminer les aléas de l'horloge). Un retard de transport assure que tous les changements quelles que soient leurs durées se propagent en amont dans la conception (Ceci est très utile pour tenir en compte toutes les impulsions des données).

I. 3. 4 - Extension du langage

Le langage VHDL ayant été défini pour constituer un standard, il présente les caractéristiques suivantes :

- il ne dépend, ni d'une technologie particulière, ni d'une méthodologie particulière de conception.
- aucune extension syntaxique ou sémantique du langage n'est possible.

Cependant, l'utilisateur peut définir des types de données et rajouter des informations temporelles ou physiques dans une description d'entité.

II - PRESENTATION DU LANGAGE LUSTRE

Avant d'étudier l'utilisation du langage LUSTRE pour la description de circuits, nous présentons ici les caractéristiques de ce langage.

LUSTRE est un langage de programmation déclaratif, synchrone [CAS 87], [BER 86-a]. LUSTRE a été basé sur une interprétation synchrone du langage à flots de données LUCID [ASH 85]. Il peut être utilisé pour la programmation de systèmes intrinsèquement parallèles : son parallélisme implicite permettra une expression facile de spécifications parallèles [BER 86-b].

II. 1 - Variables, équations, expressions

Chaque variable ou expression dans le langage LUSTRE représente une suite infinie de valeurs $X = (x_0, x_1, \ldots, x_n, \ldots)$, appelée flux, où x_n représente la valeur de X à l'instant n de son horloge.

Les variables sont définies par des équations. Si X est une variable et E une expression, l'équation $X = E$ a pour interprétation que, pour tout indice, $x_n = e_n$ $(x_0=e_0, x_1=e_1, \ldots x_n=e_n, \ldots)$, (X a la même horloge que 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.

Chaque variable qui n'est pas une entrée est définie par une équation. L'ordre des équations est sans importance. Les expressions sont construites à l'aide de variables, de constantes (considérées comme des suites de constantes) et d'opérateurs du langage [PLA 87].

II. 2 - Opérateurs du langage

On peut distinguer deux types d'opérateurs en LUSTRE :

II. 2. 1 - Opérateurs sur les valeurs

Les opérateurs sur les valeurs sont les opérateurs usuels (arithmétiques, booléens, conditionnels) qui sont étendus pour opérer sur les suites de valeurs en supposant que les entrées de ces opérateurs évoluent exactement au même rythme, par exemple :

$$Z = \text{if } X > Y \text{ then } X - Y \text{ else } Y - X$$

Cette expression définit la suite Z dont le n-ième terme est la valeur absolue de la différence des n-ièmes termes des suites définissant X et Y. Donc, les opérandes de tout opérateur sur les valeurs doivent être sur la même horloge.

II. 2. 2 - Opérateurs sur les suites

En plus des opérateurs sur les valeurs, LUSTRE possède quatre opérateurs sur les suites : les opérateurs "pre" (précédent) et "->" (suivie de) qui manipulent explicitement des suites d'une façon cyclique, mais sans introduire de nouvelles horloges, l'opérateur "when" qui permet de définir un sous-système qui n'évolue effectivement qu'à certains instants et l'opérateur "current" qui permet de mettre les opérandes sur la même horloge.

a) L'opérateur "pre"

Cet opérateur retourne la valeur de son argument à l'instant précédent. Si $X = (x_0, x_1, \ldots, x_n, \ldots)$, alors $\text{pre}(X) = (\text{nil}, x_0, x_1, \ldots, x_{n-1}, \ldots)$ où nil est une valeur indéfinie, semblable à la valeur d'une variable non initialisée dans les langages impératifs.

b) L'opérateur "->" (suivi de)

Cet opérateur sert à initialiser les variables. Si $X = (x_0, x_1, \ldots, x_n, \ldots)$ et $Y = (y_0, y_1, \ldots, y_n, \ldots)$ sont deux variables de même type, et de même horloge :

$$X \text{ -> } Y = (x_0, y_1, \ldots, y_n, \ldots).$$

Donc X->Y est toujours égal à Y sauf à l'instant initial.

c) L'opérateur "when"

L'expression X = (EXP when COND) définit la suite des valeurs de EXP aux instants où COND (variable booléenne) vaut "true". Ce filtrage est utilisé pour des variables qui n'évoluent effectivement qu'à certains instants. X est caractérisé par la suite des valeurs de EXP et la suite des valeurs de COND, on dira que X est sur l'horloge COND et COND est une horloge d'échantillonnage.

d) L'opérateur "current"

Pour pouvoir opérer sur des variables d'horloges différentes, current "projette" une expression d'horloge C sur une expression dont l'horloge sera l'horloge de C.

Si X est une expression d'horloge C, et si C est l'horloge d'échantillonnage, alors (current X) est une expression dont l'horloge est celle de C et dont la valeur à chaque instant est celle prise par E au dernier instant où C valait vrai.

exemple :

	E =(	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	...
	C =(| true	false	true	true	false	true	false	false	...
X=E when	C =(	e_0		e_2	e_3		e_5			...
Y=current(X)	=(	e_0	e_0	e_2	e_3	e_3	e_5	e_5	e_5	...

Tableau I-1 : Filtrage et projection

II. 3 - Types

LUSTRE possède les types prédéfinis suivants : booléen, entier et réel en ajoutant les types définis dans le langage hôte c'est à dire le langage cible de compilateur. L'utilisateur peut lui-même définir d'autre types, avec leurs opérations de base. De plus, il y a un autre type en LUSTRE c'est les tuples :

Si $E_1, E_2, \ldots, E_n$ sont n expressions de types $\tau_1, \tau_2, \ldots, \tau_n$, respectivement alors $(E_1, E_2, \ldots, E_n)$ est un tuple de type $(\tau_1, \tau_2, \ldots, \tau_n)$. L'horloge d'un tuple est le tuple formé des horloges de ses éléments. Un tuple peut donc avoir des éléments sur des horloges différentes. Il sera parfois commode de considérer que l'horloge d'un tuple est une horloge simple si tous les éléments ont la même horloge.

II. 4 - Noeuds hiérarchisés

L'utilisateur peut définir ses propres opérateurs, appelés nœuds. Un noeud est un sous-programme. Il reçoit des variables d'entrée, il calcule des variables de sortie et éventuellement des variables locales au moyen d'un système d'équations. Donc, un programme LUSTRE est une description d'un ensemble de nœuds et de leurs interconnexions d'entrées/sorties. Chaque nœud est constitué par une interface et un corps. Une interface définit les variables de communication avec d'autre nœuds ou avec le monde extérieur. Les variables d'entrées, de sorties et éventuellement les variables internes ont des déclarations fortement typées en vue d'une vérification lors de la compilation.

L'horloge de base d'un nœud est déterminée par l'horloge de ses paramètres d'entrée : plus précisément, si les paramètres d'entrée ont des horloges différentes, l'horloge du nœud est l'horloge du premier paramètre d'entrée. Il faut que les horloges des autres paramètres soient dérivées de cette horloge de base pour le nœud. Un noeud peut contenir d'autres noeuds internes. On peut ainsi décrire entièrement un système par une composition hiérarchisée de noeuds : on obtient ainsi un réseau de noeuds. Par exemple, on peut définir un compteur généralisé comme suit:

```
node COUNTE(INIT, INCR : int; RAZ : bool) returns (N : int)
let
N = INIT-> if RAZ then INIT
           else pre(N)+INCR;
tel.
```

On peut écrire :

```
EVEN = COUNTE(0,2,false)
MOD4 = COUNTE(0,1,pre(MOD4=3)).
```

Ce qui définit "EVEN" comme la suite des nombres pairs et "MOD4" comme la suite cyclique des entiers modulo 4. COUNTE((0,1,false) when C) exécute un cycle et calcule le noeud COUNTE chaque fois que C vaut true.

L'horloge de base d'un nœud est la racine d'un arbre qui est constitué par l'ensemble des horloges filles. Une horloge Y est "l'enfant" d'une horloge X si Y est crée à partir de X en utilisant l'échantillonnage "when". Notons que, puisque l'opérateur when ne distribue pas sur les opérateurs de suite, il n'est généralement pas équivalent de synchroniser l'exécution d'un noeud en échantillonnant ses entrées (comme précédemment), et d'échantillonner ses sorties. Dans ce dernier cas on calcule le noeud COUNTE chaque cycle, mais on ne "montre" le résultat que quand C vaut true.

COUNTE((0,1,false) when C) ≠ (COUNTE(0,1,false)) when C.

Un nœud peut recevoir des entrées d'horloges différentes, mais lorsque l'horloge d'un paramètre n'est pas l'horloge de base, cette horloge doit elle-même être passée en paramètre. Par exemple, l'entête de déclaration d'un nœud pourrait être :

```
node N(E : bool; (X : bool) when E) returns . . .
```

L'horloge de X est E c'est à dire X n'existe que quand E est vraie.

On peut aussi retourner des résultats d'horloges différentes, à condition que toutes les horloges soient visibles de l'extérieur du nœud. Par exemple, le nœud UNE_SUR_DEUX suivant supprime une entrée sur deux [PLA 88] :

```
node UNE_SUR_DEUX(X : int) returns(horloge : bool; Y : int);
let
horloge = true-> not pre(horloge);
Y = X when horloge;
tel.
```

II. 5 - Assertions

Le compilateur LUSTRE synthétise le contrôle en évaluant de manière symbolique les expressions booléennes, ce qui lui permet de générer des automates. Afin de limiter le risque d'explosion combinatoire de ces automates, tant en nombre d'états qu'en nombre de transitions, le programmeur peut écrire des assertions c'est à dire des expressions qui sont vraies à tout instant. Le compilateur en générant les états et les transitions, peut utiliser les assertions fournies par le programmeur pour élaguer des chemins, permettant ainsi de réduire la taille de l'automate engendré (les assertions sont implémentées par le compilateur LUSTRE version 2).

Exemple :

Un registre ayant deux signaux de "set" et "reset" peut être décrit en supposant au niveau du nœud LUSTRE qu'un seul signal peut arriver au même moment. L'assertion qui réalise cette hypothèse est la suivante :

```
assert # (set, reset);
```

Cette assertion permet de minimiser les transitions effectuées en testant le signal reset par exemple (resp. set). Si ce signal est vrai, ce n'est pas la peine de tester le deuxième signal set (resp. reset).

II. 6 - L'environnement du système LUSTRE

Le système LUSTRE a été défini d'une façon très flexible permettant l'ajout d'autres outils. La figure I-9 représente l'environnement du système LUSTRE. Il est constitué d'un compilateur, d'un minimiseur et d'un traducteur.

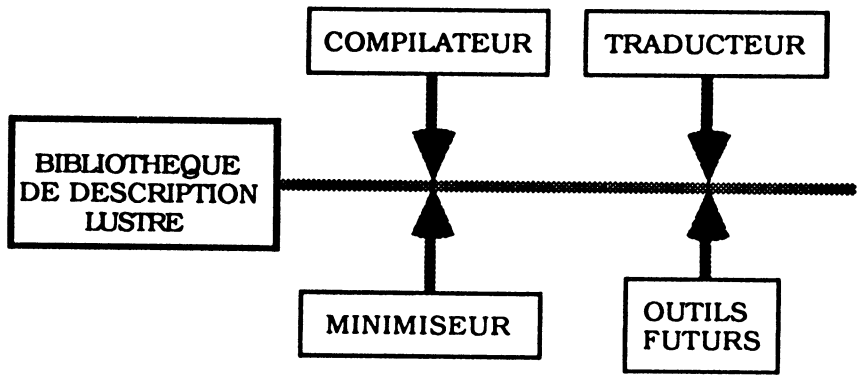


Figure I-9 : L'environnement de LUSTRE.

L'organisation générale du système LUSTRE est montrée dans la figure I-10. Le compilateur LUSTRE synthétise le contrôle des programmes sous forme d'automate d'états fini [CAS 87].

La compilation d'une source LUSTRE se déroule en deux phases. A la fin de la première phase, qui consiste en une analyse syntaxique et sémantique, on obtient un réseau d'opérateurs et à la fin de la deuxième phase, qui se fait par le générateur du code, on obtient le code intermédiaire. Ce code, qui spécifie toutes les actions produites dans tous les états, est présenté en formats portable, nommé OC (Objet Code), commun LUSTRE/ESTEREL [COU 87].

Le rôle du minimiseur est d'éliminer les états redondants équivalents qui peuvent exister dans l'automate obtenu après la compilation d'un programme LUSTRE. Ce code intermédiaire portable minimisé peut être traduit dans un langage impératif (par exemple le langage PASCAL ou C). On obtient donc un modèle (un fichier exécutable) qui simule le circuit.

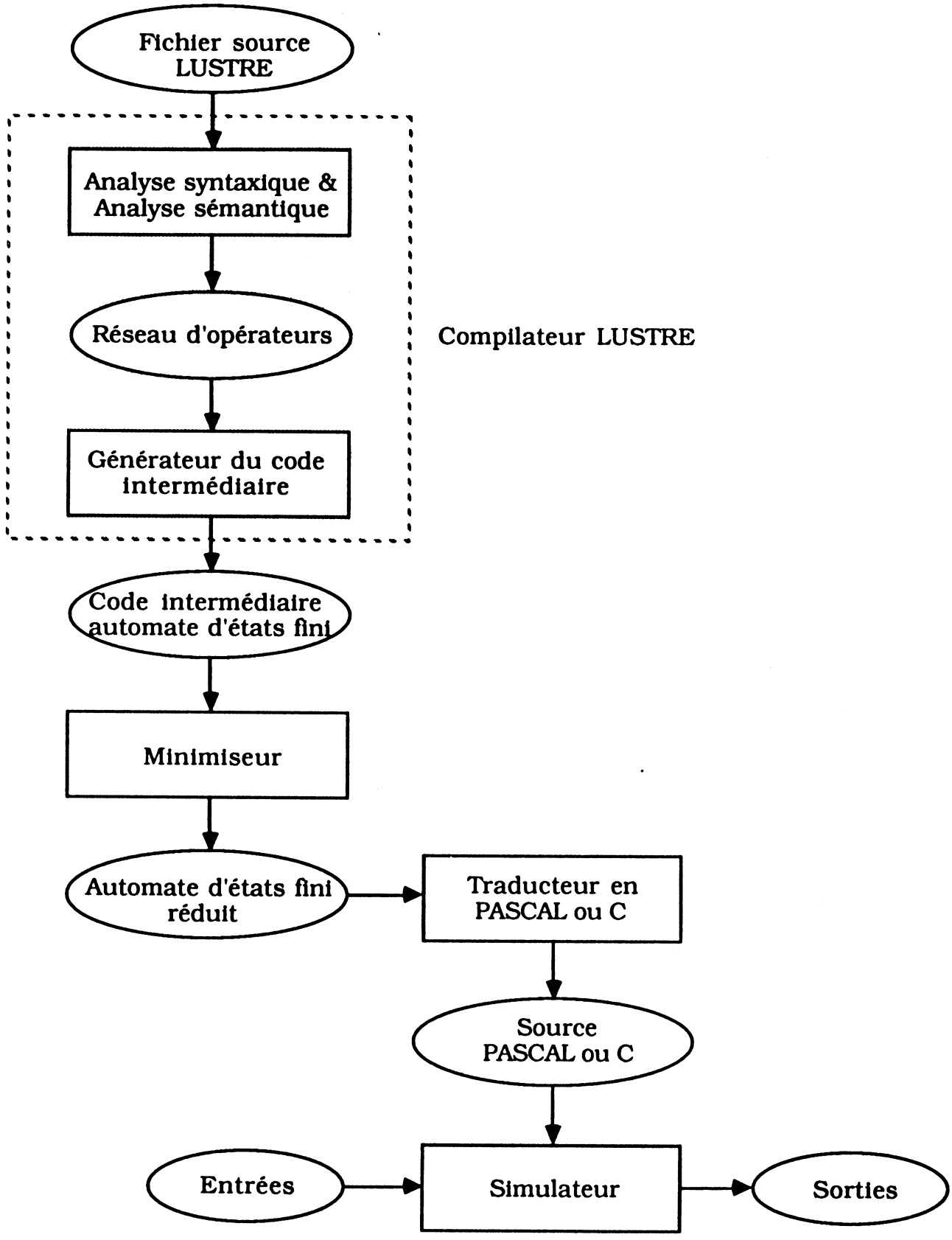


Figure I-10 : Organisation générale du système LUSTRE.

Le compilateur LUSTRE génère un automate d'états fini toujours connexe. On a un état initial (état 0) dans lequel les variables sont affectées avec leurs valeurs initiales. Une ***variable d'état*** est une variable de type booléen utilisée dans le programme LUSTRE avec l'opérateur "pre" (qui renvoie la valeur précédente de la variable en question).

Le nombre d'états d'un programme LUSTRE augmente en général exponentiellement avec le nombre de variables d'états. Si on a m variables d'états dans un programme LUSTRE, on obtient jusqu'à 2^m états après la compilation (toutes les combinaisons possibles des variables d'états) mais généralement le maximum n'est pas atteint. Par exemple, si les variables d'états sont toutes des variables d'entrées et qu'on n'utilise que leur valeur précédente (et non pre(pre(. . .))) le graphe des états est un graphe complet (en excluant l'état initial 0). Ce qui signifie que l'automate peut passer d'un état à n'importe quel autre à l'instant suivant.

III - LUSTRE ET LA DESCRIPTION DE CIRCUITS

On présentera par la suite notre étude sur les différentes caractéristiques du langage LUSTRE en tant que langage de description de circuits tout en mettant en évidence les extensions à effectuer pour qu'il soit d'une utilisation aisée et s'adapte mieux au problème de description de circuits.

III. 1 Analyse des caractéristiques du langage pour la description de circuits

Le fonctionnement du matériel étant naturellement parallèle, sa description est facilitée par un langage de type déclaratif. LUSTRE permet d'exprimer le parallélisme intrinsèque aux circuits et offre un moyen très adéquat pour modéliser les aspects temporels du fonctionnement des circuits. Son parallélisme implicite, dû à sa nature "data-flow", est parfaitement adapté au problème de description de circuits. Sa sémantique simple et sa syntaxe réduite permettent de faciliter encore cette description.

Une première tentative pour utiliser LUSTRE dans le domaine de la description de circuits est présentée dans [LON 86], [HAL 86]. Dans la suite on va donner quelques exemples d'utilisation du langage LUSTRE. D'autres circuits complexes se trouvent dans [MAH 88].

Exemple

Considérons un circuit composé d'un registre simple 4 bits et d'un registre à décalage 4 bits, indiqué dans le schéma figure I-11.

Rout représente la sortie (4 bits) du registre reg et clk_1 est son signal d'horloge. Cette sortie est connectée à l'entrée parallèle du registre à décalage shreg. D représente l'entrée série. L'entrée p/s permet de choisir entre deux modes de fonctionnement : série ou parallèle, clk_2 étant l'horloge du registre à décalage.

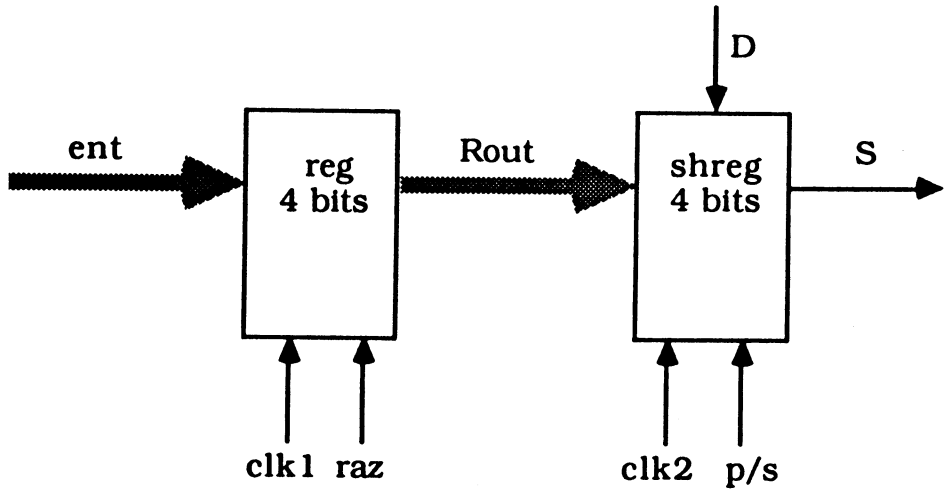


Figure I-11 : Le schéma de ce circuit.

Un programme LUSTRE décrivant ce circuit est le suivant :

```
node front(clk : bool)                          - définition d'un front sur un
        returns(trans : bool);                  - signal d'horloge du circuit.
let
trans= clk and (false ->not pre(clk));
tel.

node registre(clk, raz : bool; entree : int) - description registre sensible au
        returns(sortie : int);                  - front montant.
let
sortie= 0-> if raz  then 0
                else pre( if front(clk) and not raz  then entree
                                                     else sortie );
tel.

node shiftreg(ps, clk : bool; E, D : int)    - description registre série/parallèle
        returns(Q4 : int);                   - sensible au front montant.
var reg : int;
let
reg= 0-> pre( if front(clk) then if ps then E
                                 else ((reg mod 8) * 2) + (D mod 2)
              else reg );
Q4= (reg div 8) mod 2;
tel.
```

```
node circuit(clk1, clk2, ps, raz : bool; ent, D int)  - description du circuit.
            returns(S : int);
var Rout : int;
let
Rout= registre(clk1,raz,ent);
S= shiftreg(ps,clk2,Rout,D);
tel.
```

Le nœud "front", dans ce programme, détecte les fronts montants sur un signal d'horloge "clk" : sa sortie "trans" est initialement fausse, elle sera vraie si "clk" passe de zéro à un. (On peut, si on veut, détecter les fronts descendants en utilisant le noeud précédent appelé par front(not clk)).

Le nœud "registre" décrit un registre simple de 4 bits ayant un signal de remise à zéro raz réalisé avec des bascules sensibles au front montant du signal d'horloge clk. "entree" est un entier compris entre 0 et 15.

Le nœud "shiftreg" décrit un registre à décalage 4 bits réalisé avec des bascules sensibles au front montant de l'horloge. Le chargement parallèle (resp. série) de ce registre s'effectue sur le front montant du signal d'horloge "clk" si ps est vrai (resp. faux).

Le nœud "circuit" décrit la connectique entre ces éléments.

Pour les deux registres dans cette description, on a exprimé le retard dû au temps de propagation vers la sortie de la valeur de l'entrée au front d'horloge : si il y a un front montant à l'instant t (et pas de signal raz pour reg), le changement de valeur de la sortie sera effectif à l'instant t+1. Par contre on remarque que la valeur initiale du registre est fixée à 0, alors que dans la réalité cette valeur est inconnue.

On a 2 variables d'états dans cet exemple clk_1, clk_2 ce qui nous donne 4 états. Le graphe d'états obtenu après la compilation du programme LUSTRE précédent est le suivant, figure I-12. On remarque que c'est un graphe complet.

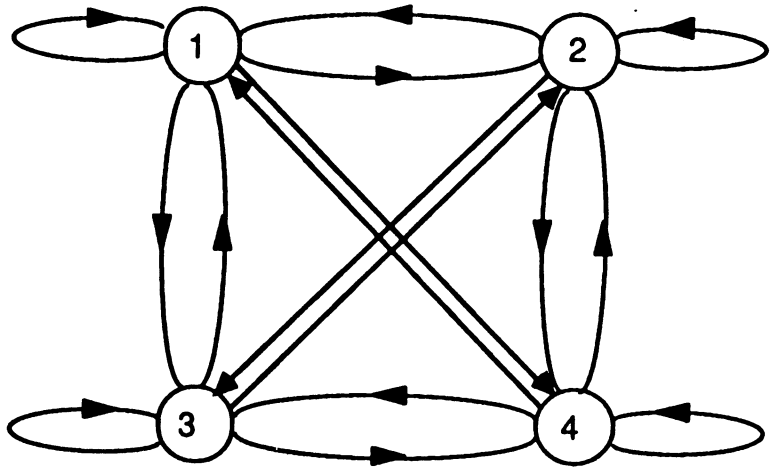


Figure I-12 : Graphe d'états obtenu (sans l'état initial).

Cette description montre qu'il est possible d'utiliser le langage LUSTRE en tant que langage de description de circuits. Il semble donc qu' un langage déclaratif est adapté à la description des circuits. Nous étudions par la suite les types de descriptions possibles par ce langage et ses caractéristiques temporelles.

III. 1. 1 - niveau et type de description

LUSTRE est un langage multi-niveaux. Il est capable de décrire les circuits du niveau sous-système (description fonctionnelle avec une abstraction haut niveau) au niveau logique (description au niveau porte). Comme les autres langages de description de circuits HDL, LUSTRE ne permet pas de décrire les circuits au niveau électrique.

La nature déclarative du langage LUSTRE implique que tout description LUSTRE est de type structurel explicitement (interconnexion de nœuds) ou implicitement. Cela n'empêche pas la description à un haut niveau d'abstraction.

Exemple :

Cet exemple illustre les différents niveaux de description en décrivant globalement et au niveau logique un additionneur 4 bits.

```
node GLOBAL_ADDER(A, B : int) returns(SUM, C : int);
assert ((A < 16) and (B <16));
let
SUM= (A + B) mod 16;
C= (A + B) div 16;
tel.
```

Une description structurelle, faisant appel à un nœud additionneur 1 bit (ADDER_BIT) est la suivante :

```
node XOR(A, B : bool) returns(C : bool);
let
C= (A and not B) or (not A and B);
tel.

node ADDER_BIT(X, Y, Cin : bool) returns(SUM, Cout : bool);
var S : bool;
let
S= XOR(X, Y);
SUM= XOR(S, Cin);
Cout= (X and Y) or (S and Cin);
tel.

node STRUCTURAL_ADDER(X1, X2, X3, X4, Y1, Y2, Y3, Y4 : bool)
        returns(Z1, Z2, Z3, Z4, C :  bool);
var C1, C2, C3 : bool;
let
(Z1, C1)= ADDER_BIT(X1, Y1, false);
(Z2, C2)= ADDER_BIT(X2, Y2, C1);
(Z3, C3)= ADDER_BIT(X3, Y3, C2);
(Z4, C)= ADDER_BIT(X4, Y4, C3);
tel.
```

LUSTRE permet de spécifier des structures de type transfert de données (les manipulations, traitements et calculs à effectuer et les affectations à exécuter) ainsi que le comportement de type flux de contrôle (opérateur if-then-else) d'une façon parallèle. La distinction de ces deux types est facile.

LUSTRE est capable d'effectuer une simulation au niveau mixte qui permet de simuler un ensemble d'éléments aux différents niveaux d'abstraction. Il permet encore de simuler des éléments combinatoires et séquentiels dans le même modèle. L'abstraction de données offerte en LUSTRE est très utile pour compacter la taille de la description. Parmi les autres avantages du langage LUSTRE, on peut citer le fait qu'il est facile à utiliser (la syntaxe de LUSTRE est réduite et sa sémantique est parfaitement définie ce qui permet d'utiliser des opérateurs mathématiques puissants).

Remarque :

Le partitionnement en unités et l'assemblage de ces unités pour avoir une conception complète est tout à fait possible du fait de l'aspect modulaire du langage LUSTRE. Chaque unité de conception en LUSTRE décrivant un sous-composant spécifique peut être mise dans un fichier. L'environnement LUSTRE permet la réutilisation de ces unités et la création d'une pseudo-bibliothèque mais le compilateur recompile toutes ces unités en tenant compte de leur intégralité. La compilation séparée n'est pas possible en LUSTRE (contrairement à VHDL, où ce mécanisme de base du langage ADA est repris).

III. 1. 2 - Caractéristiques temporelles

LUSTRE permet la description des circuits combinatoires et des circuits séquentiels. Les circuits séquentiels décrits en LUSTRE peuvent être :

* des circuits synchrones dans lesquels l'horloge de synchronisation peut être : soit l'horloge de base, soit une de ses dérivées ;
* des circuits asynchrones dont l'évolution est dirigée par les changements de valeurs des entrées.

Chaque instant en LUSTRE est défini par rapport à l'horloge de base ou une de ses dérivées. Le comportement temporel du matériel est une partie très importante de sa description, et la notion du temps de LUSTRE est très proche du matériel. Chaque flux a son horloge qui est elle-même un flux. L'opérateur "when" permet de filtrer des événements, pour des modules qui n'évoluent effectivement qu'à certains instants, ce qui est bien adapté aux circuits. Les deux opérateurs (when et current) permettent la transformation de flux.

L'horloge de base d'un nœud est fixée par les entrées englobantes. C'est l'horloge de la première entrée qui spécifie le rythme de calcul de ce nœud et donc toutes les autres horloges doivent être dérivées de cette horloge. Dans l'exemple suivant, c'est clk dans le nœud "registre" de cette description qui fixe le temps. Les calculs de ce nœud sont effectués suivant le rythme de clk.

```
node registre(clk, raz : bool; entree : int) returns(sortie : int);
let
sortie= 0-> if raz  then 0
                    else pre( if front(clk) and not raz  then entree
                                                         else sortie );
tel.
```

Les sorties d'un nœud peuvent être sur des horloges différentes à condition de déclarer en sortie l'horloge de calcul de chaque variable qui évolue sur une horloge différente de l'horloge de base, avant de mettre la variable elle-même.

Nous allons analyser les aspects temporels de la description de circuits en LUSTRE : d'abord, nous discutons de l'horloge de synchronisation d'un circuit et de la modélisation du retard de propagation pour les circuits combinatoires. Ensuite, on présenterons les différentes possibilités offertes par le langage LUSTRE pour décrire la synchronisation des circuits séquentiels.

III. 1. 2. 1 - Horloge de base/horloge du circuit

Il faut bien distinguer l'horloge de base LUSTRE (qui est définie par l'horloge du premier paramètre d'entrée du nœud le plus englobant) et l'horloge du circuit décrit. L'horloge de base peut représenter un temps "réel", c'est à dire une ou plusieurs ns par exemple ou être plus ou moins similaire à l'horloge du circuit (c'est à dire correspondre à une période ou à une demi-période d'horloge du circuit).

Si l'horloge de base représente un temps "réel", cela permet de simuler les circuits au niveau logique et de modéliser finement les retards de propagation. L'horloge du circuit est alors définie à l'intérieur d'un nœud à partir de l'horloge de base.

Exemple

Considérons une horloge de base 10 ns et une horloge du circuit 100 ns. Dans la description suivante l'horloge du circuit H100ns est dérivée de l'horloge de base H10ns qui peut être représentée par une entrée booléenne vraie chaque 10 ns :

```
node horloge_circuit(H10ns : bool) returns(H100ns : bool);
var C1 : int; syn, Hsyn : bool;
let
C1= 0-> (pre(C1) + 1) mod 5;
syn=(C1=0);
Hsyn= (false when syn) -> not pre(Hsyn);
H100ns= current(Hsyn);
tel.
```

La variable Hsyn a comme horloge syn ("when syn" sur la valeur initiale impose l'horloge syn pour toute expression). Le chronogramme de l'horloge du circuit est montrée dans le tableau suivant I-2 :

H10ns	T T T T T T T T T T T T T T T T T T T T T T T T T T T T T
C1	0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0
syn	T F F F F T F F F F T F F F F T F F F F T F F F F T
Hsyn	F T F T F T
H100ns	F F F F F T T T T T F F F F F T T T T T F F F F F T

Tableau I-2 : Description de l'horloge du circuit à partir d'une horloge de base fine.

Prendre l'horloge de base similaire à l'horloge du circuit permet de simuler les circuits à un niveau plus fonctionnel. Certains circuits ne sont sensibles qu'à un front de l'horloge du circuit (montant ou descendant). On peut prendre alors l'horloge de base comme horloge du circuit. Le nœud le plus englobant aura la forme suivante :

node circuit(clk : bool, . . .)

clk étant donné en entrée sous la forme T T T T T . . . (T=True). Implicitement une période d'horloge sépare deux valeurs "true" de la variable clk.

D'autres circuits complexes sont sensibles aux fronts montants et descendants de l'horloge du circuit. De même, le nœud le plus englobant aura la forme :

node circuit(clk : bool, . . .)

mais clk sera définie en entrée par T F T F T . . . (T=True, F=False) avec implicitement une demi-période d'horloge du circuit séparant deux valeurs de clk.

Remarque 1

On peut aussi définir les deux fronts d'horloge du circuit d'une façon interne :

```
node circuit(H, . . .)
let
clk= false-> not pre(clk);
. . .
tel;
```

```
avec  H   =   T   T   T   T   T   T   T   T
      clk =   F   T   F   T   F   T   F   T
```

Deux valeurs de H sont séparées par une demi-période d'horloge.

Remarque 2

On peut ne pas définir explicitement l'horloge comme une entrée externe à condition que la première entrée déclarée soit définie à la fréquence de l'horloge. Le nœud suivant :

```
node circuit(clk, I: bool; (J : int) when I)
```

```
avec  clk =   T   T   T   T   T   T   T   T
      I   =   T   T   T   F   F   T   F   T
      J   =   J1  J2  J3              J4      J5
```

où J n'est défini que quand le signal I est vrai (échantillonnage sur l'horloge I), est équivalent à :

```
node circuit(I: bool; (J : int) when I)
```

```
avec  I   =   T   T   T   F   F   T   F   T
      J   =   J1  J2  J3              J4      J5
```

III. 1. 2. 2 - Description des retards dans les circuits combinatoires

Le retard de propagation d'un signal est défini par le temps nécessaire à partir de l'instant de changement de ce signal pour propager la nouvelle valeur de ce signal jusqu'à la sortie du circuit.

Deux solutions seront possibles pour modéliser ce retard. La première est d'utiliser une horloge de base très fine par rapport à l'horloge du circuit (l'horloge du circuit sera décrite comme on vu dans le paragraphe précédent) et d'ajouter un opérateur de retard ("pre") à chaque opérateur logique, figure I-13.

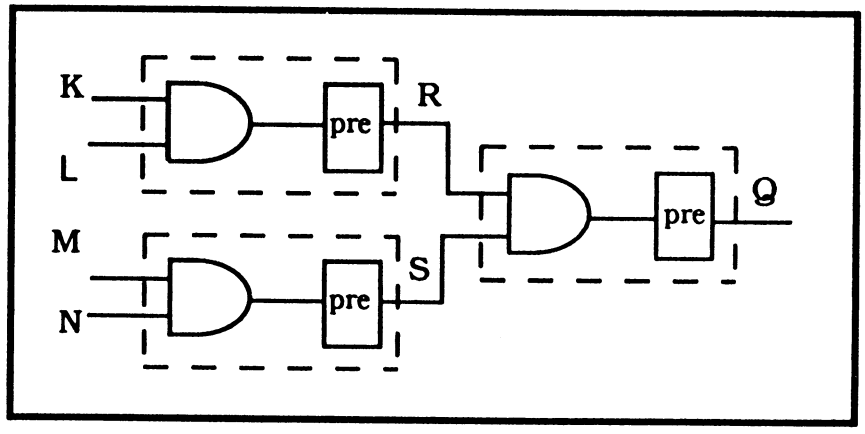


Figure I-13 : Association d'un "pre" à chaque opérateur logique.

Cette méthode présente les caractéristiques suivantes :

1) Si on considère que l'horloge de base est vraie chaque 10 ns, l'opérateur "pre" sur un signal décale ce signal d'une période de l'horloge de base. La sortie sera décalée à un nombre de périodes de base égal aux nombres d'étages de ce circuit. La figure I-14 montre les chronogrammes des signaux M et N et du résultat pre(M and N) de la figure I-13 .

2) Le retard obtenu est additif.

En fait, ce qui nous intéresse, c'est les spécifications fonctionnelles de conception d'un circuit. C'est pourquoi on se contente de modéliser le circuit au niveau fonctionnel. La deuxième solution consiste à considérer que l'horloge du circuit est similaire à l'horloge de base et à ajouter un "pre" à la sortie du circuit, figure I-15, pour les retards accumulés dans les différentes portes logiques.

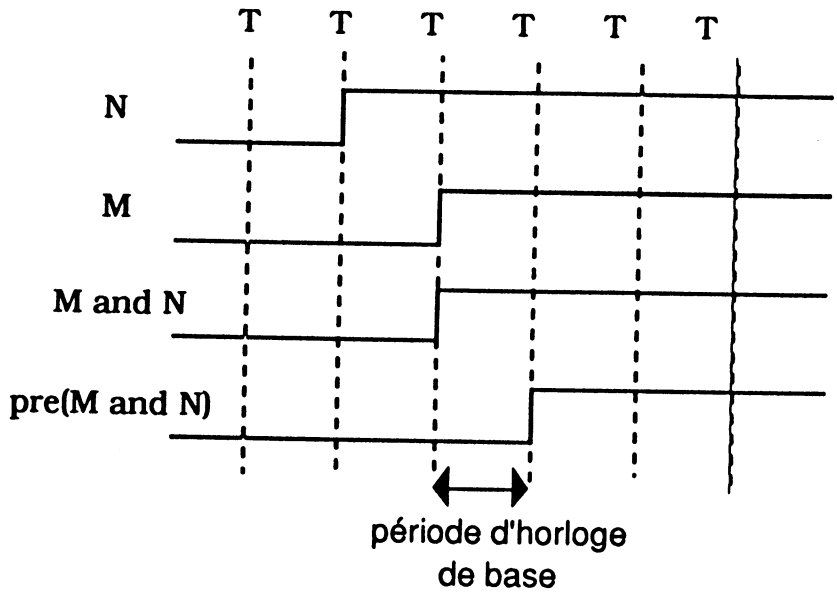


Figure I-14 : Retard d'une période d'horloge de base.

Cependant, cette méthode présente les caractéristiques suivantes :

1) En considérant cette solution, on spécifie que, une période d'horloge plus tard (ou une demi-période d'horloge), la sortie est correcte, mais la façon de décrire un circuit sera non modulaire. Ce qui pose des sérieux problèmes, lors de l'insertion de ce nœud dans une description englobante surtout avec plusieurs appels de ce nœud.

2) Le retard obtenu n'est pas additif.

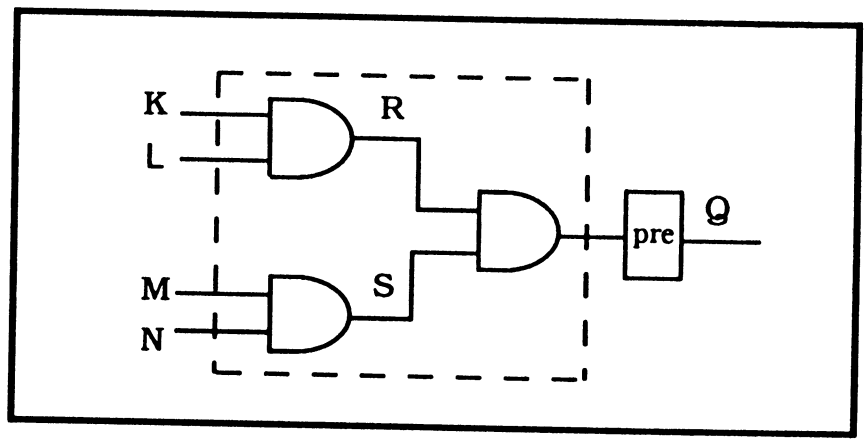


Figure I-15 : L'ajout d'un opérateur "pre" à la sortie du circuit.

Aucune de ces deux solutions n'est satisfaisante. Cependant, dans une description haut niveau, on peut ignorer les retards des signaux correspondant au temps de traversée des différentes portes logiques pour les circuits combinatoires : l'important est que les sorties soient échantillonnées correctement. Par contre, il est nécessaire de considérer le retard pour les bascules, ce qui sera l'objet du paragraphe suivant.

III. 1. 2. 3 - Modélisation des bascules et registres

Il est nécessaire de ne pas ignorer une des caractéristiques des bascules qui est le retard de propagation noté T_{PD} (figure I-16).

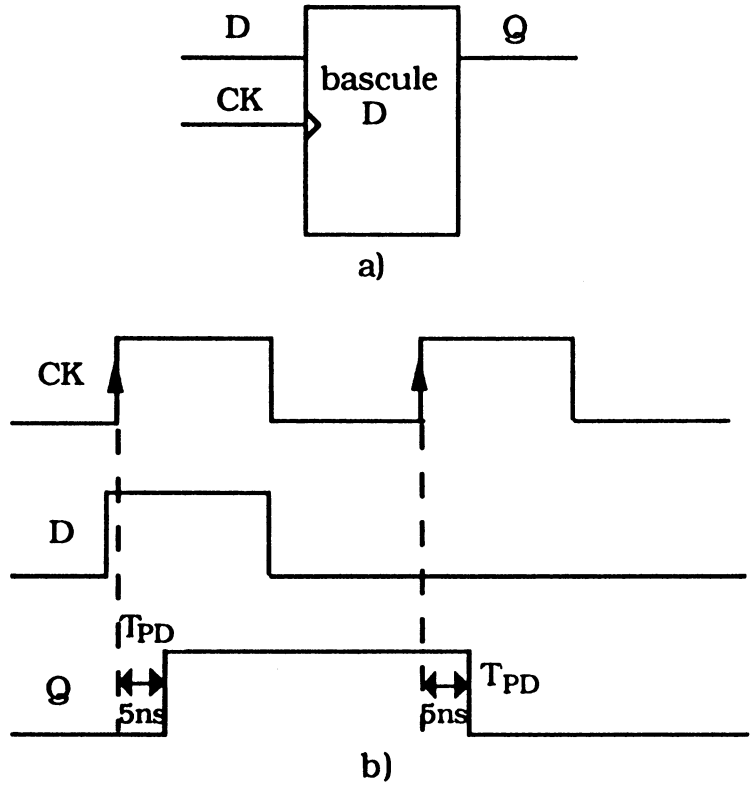


Figure I-16 : a) une bascule D ; b) le temps de propagation T_{PD}.

Si on ne tient pas compte de ce retard, un registre à décalage, formé d'une chaîne de bascules, ne peut pas fonctionner correctement : la valeur en entrée se propage instantanément dans tout le registre. Etant donnée la nature déclarative du langage LUSTRE, ce retard doit être décrit explicitement (ce qui n'est pas toujours nécessaire dans un langage impératif).

a) Définition fine des horloges

Il est possible de décrire ce retard en définissant finement l'horloge de base.

Exemple :

Considérons une bascule D synchronisée par un signal d'horloge CK, figure (I-16, a) (on a ignoré le signal de remise à zéro et le signal de remise à 1). Un temps de propagation T_{PD} (égal par exemple à 5 ns) est nécessaire pour que la valeur de l'entrée D échantillonnée sur le front montant de l'horloge CK arrive à la sortie Q montré dans la figure (I-16, b) (on a considéré que le temps de montée de l'horloge et de la donnée sont égaux à 0).

Une description de cette bascule prenant en compte le temps de propagation est indiquée ci-dessous. Elle est basée sur une définition fine du temps : l'horloge de base a une période de 1 ns.

On définit, d'abord, le signal nanoseconde (ns) qui peut être représenté par une entrée booléenne. Le signal (H5ns) est une autre variable booléenne définie sur l'horloge nanoseconde et vraie une fois sur cinq. On définit, ensuite, l'horloge de synchronisation de cette bascule (H20ns) définie sur l'horloge nanoseconde avec une période 20 ns et vraie une fois sur vingt. Le retard de traversée est spécifié égal à 5 ns. La description LUSTRE de cette bascule est la suivante :

```
node generation_horloge(ns : bool) returns(H5ns, H20ns : bool);
var C1, C2 : int;
let
C1= if (C2=0) then 0
     else 0-> (pre(C1) + 1) mod 5;
C2= 0-> (pre(C2) + 1) mod 20;
H5ns= (C1=0);
H20ns= (C2=0);
tel.
```

```
node bascule(H5ns, H20ns : bool; D : int) returns(Q : int);
var  D1, D2, D3 : int;
let
D1= current(D when H20ns);
D2= D1 when H5ns;
D3= (0 when H5ns)-> pre(D2);
Q= current(D3);
tel.

node gen_bascule1(ns : bool; D : int) returns(Q : int);
var H5ns, H20ns : bool;
let
(H5ns, H20ns)=generation_horloge(ns);
Q= bascule(H5ns, H20ns, D);
tel.
```

Les chronogrammes des différentes variables sont montrés dans le tableau suivant :

ns	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
C1	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0
C2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	0	1	2	3	4	5
H5ns	T	F	F	F	F	T	F	F	F	F	T	F	F	F	F	T	F	F	F	F	T	F	F	F	F	T
H20ns	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	F	F	F	F	F
D	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
D2	1					1					1					1					0					0
D3	0					1					1					1					1					0
Q	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Tableau I-3 : Description fine d'une bascule D.

Cette description a été écrite pour être la plus générale possible : elle peut être insérée dans toute description où l'unité de temps est la nanoseconde. Le nœud generation_horloge permet de générer le retard

considéré et l'horloge de synchronisation de cette bascule. Dans ce nœud le compteur correspondant au signal H5ns est initialisé à zéro chaque fois que le signal H20ns est vrai (début de la période de l'horloge de synchronisation). La compilation de cette description mène à un automate complexe avec 3 états.

Si on ne veut définir D et H20ns que quand le signal H5ns est vrai, on utilisera l'échantillonnage sur les entrées. Le nœud précédent s'écrit de la manière suivante en utilisant l'opérateur "when" dans l'entête du nœud "bascule" appliqué pour le paramètre d'entrée D et H20ns :

Exemple :

```
node bascule(H5ns : bool; (H20ns : bool) when H5ns; (D : int) when H5ns)
        returns(Q : int);
var  D1 : int;
let
D1= current(D when H20ns);
Q= (0 when H5ns)-> pre(D1);
tel.

node gen_bascule2(ns : bool; D : int) returns(Q : int);
var H5ns, H20ns : bool;
let
(H5ns, H20ns)=generation_horloge(ns);
Q= bascule(H5ns, H20ns, D);
tel.
```

Cette description calcule Q seulement quand H5ns est vrai.

b) Modélisation fonctionnelle

Il est souhaitable, pratiquement, de simplifier la prise en compte du retard décrit d'une façon fine ci-dessus pour les circuits séquentiels en considérant que l'horloge de circuit est similaire à l'horloge de base. Le retard sera alors réalisé par l'opérateur "pre" qui permet de retarder les variables d'une période d'horloge du circuit (ou d'une demi-période d'horloge).

On peut utiliser une horloge externe comme une horloge de synchronisation dans une description. Cette description dépend alors entièrement de la séquence d'entrées de cette horloge.

Exemple :

On définit "ck" de la manière suivante :

ck= T F T F T F ...

La description de la bascule est alors :

```
node front(clk : bool) returns(trans : bool);
let
trans= clk and (false ->not pre(clk));
tel.

node bascule3(ck : bool; D : int) returns(Q : int);
let
Q= 0-> pre(if front(ck) then D
                        else Q);
tel.
```

Si on ne veut définir Q que quand on a un front montant sur le signal d'horloge ck à l'instant précédent (échantillonnage des sorties sur une horloge interne), on écrira le nœud précédent en utilisant l'échantillonnage sur les sorties de la manière suivante :

Exemple :

```
node front(clk : bool) returns(trans : bool);
let
trans= clk and (false ->not pre(clk));
tel.
```

```
node bascule4(D : int) returns(ech : bool; Q : int);
var ck, fm : bool;
let
ck= true -> not pre(ck);
fm= front(ck);
ech= true -> pre(fm);
Q= D when ech;
tel.
```

Donc, les caractéristiques temporelles de ce langage permettent de l'utiliser pour la description de circuits à différents niveaux de détail.

III. 1. 3 - Types de données

Les données en LUSTRE comme en VHDL sont fortement typées ce qui permet un test sémantique pendant la compilation. Ceci exige que chacune variable doit être déclarée avant son utilisation. Cette contrainte ne semble pas déraisonnable pour une description comportementale de haut niveau, mais quand il s'agit de décrire une conception complètement structurelle à un niveau plus bas, l'obligation de déclarer chaque composant ou signal avant son utilisation rend le texte très lourd.

Les types de données utilisés en LUSTRE sont :

- les types prédéfinis : entier, réel et booléen

- les types importés définis par l'utilisateur dans le langage hôte (le langage C).

La conversion de types de données, qui sont déclarés dans un nœud, est possible en LUSTRE mais d'une façon forcément explicite. Toutefois, LUSTRE n'offre pas la facilité d'inclure plusieurs alternatives de conception dans la même description (Cette possibilité est faisable en VHDL).

Une comparaison des différentes caractéristiques, entre le langage VHDL et LUSTRE se trouve dans le tableau suivant :

Caractéristiques générales		VHDL	LUSTRE
niveau et type de description	niveau sous-système	x	x
	niveau logique	x	x
	circuits combinatoires	x	x
	circuits séquentiels	x	x
	circuits synch. et asynch.	x	x
	descri. comporte. impérative	x	
	descri. comporte. déclarative	x	x
	description structurelle	x	x
	description hiérarchisée	x	x
Caractéristiques du langage	modularité	x	x
	bibliothèque	x	-
	abstraction de données	x	x
	conversion de données	x	x
	alternative de conception	x	
	compilation séparée	x	
	interface explicite	x	x
	variables fortement typées	x	x
	parallélisme	x	x
	descriptions fonct. équiv.	x	x
	contrôle et données séparés	x	x
	assertion	x	x
	paramètres génériques	x	
	structures régulières	x	x
	structures récursives	x	
Caractéristiques temporelles	aspects temporels	x	x
	contraintes temporelles	x	x
	retard de propagation	x	
	tps inertiel et transport	x	
Extensibilité du langage	nouveaux types	x	x
	types définies par l'utilisa.	x	x
	nouveaux opérateurs	x	x
	indépendance de la techno.	x	x
	ajout d'autres outils	x	x

Tableau I-4 : Comparaison entre le langage VHDL et LUSTRE.

III. 2 - Extension du langage : des propositions

Dans l'optique de permettre à LUSTRE d'être utilisé dans le domaine de description de matériel d'une façon aisée, nous proposons une extension de ce langage. A l'heure actuelle, LUSTRE ne permet pas de décrire des structures régulières, ni d'utiliser des types de données structurés (tableaux ou enregistrements).

Cette extension du langage LUSTRE se situe sur deux axes principaux : d'abord, la définition de nouveaux types primitifs supplémentaires de données, ces types de données définis par l'utilisateur pouvant être ajoutés à une version future du langage, ensuite, l'introduction de nouvelles valeurs X, U, Z caractéristiques du matériel.

III. 2. 1 - Introduction de nouveaux types

Pour pouvoir décrire le matériel trois types supplémentaires sont nécessaires. Le premier type proposé (type "bit") permettrait de simplifier l'automate obtenu après la compilation d'un programme LUSTRE (une variable de type bit est une variable booléenne qui n'augmente pas le nombre d'états). Le deuxième type (type tableau), qui permet de définir un ensemble de booléens ou d'entiers, et le troisième type (type énuméré), qui permet à l'utilisateur de définir ses propres objets par un vecteur en énumérant les objets possibles, permettraient de simplifier la description.

III. 2. 1. 1 - Introduction du type "bit"

On peut utiliser le type "bool" du langage LUSTRE pour définir les signaux binaires (valeur 0 ou 1). L'utilisation de ce type augmente le nombre d'états de l'automate obtenu. Ce n'est pas toujours nécessaire. En fait, il faut distinguer deux types de signaux : signaux de contrôle et signaux de données. Il est logique que les signaux de contrôle soient de type booléens et contribuent à définir l'état de l'automate obtenu. Par contre, les signaux de

données n'ont pas d'influence directe sur l'état. Il faut donc éviter de les déclarer comme de type "bool".

Actuellement, on peut utiliser le type entier de LUSTRE "int" pour déclarer des variables de données binaires, en affectant à ces variables des valeurs appropriées mais cela nécessite un traitement supplémentaire et rend la description complexe et moins proche du circuit réel. Nous proposons un type "bit" (valeur 0 et 1), qui rendrait plus aisé les descriptions.

Exemple :

Reprenons l'exemple de bascule-D (paragraphe III. 1), en considérant que l'entrée et la sortie de cette bascule soient déclarées de type bit, la description devient :

```
node bascule(H5ns, H20ns : bool; D : bit) returns(Q : bit);
var  D1, D2, D3 : bit;
let
D1= current(D when H20ns);
D2= D1 when H5ns;
D3= (0 when H5ns)-> pre(D2);
Q= current(D3);
tel.

node gen_bascule1(ns : bool; D : bit) returns(Q : bit);
var H5ns, H20ns : bool;
let
(H5ns, H20ns)=generation_horloge(ns);
Q= bascule(H5ns, H20ns, D);
tel.
```

L'horloge de synchronisation est déclaré comme booléen ; l'entrée D et la sortie Q sont déclarées comme bit. Après la compilation de cette description, on obtient un automate à 3 états alors que la déclaration de l'entrée D et la sortie Q comme des booléens mène à un automate à 21 états.

III. 2. 1. 2 - Introduction du type tableau et énuméré

L'utilisation des tableaux comme type de données à structure régulière est très importante. Un tableau pourrait être déclaré en spécifiant le nombre d'éléments à réserver et le type utilisé par exemple :

VECTOR : int ^ 5 ;
BYTE : bit ^ 8 ;
VECTOR_BIT : bool ^ 4;

VECTOR est une variable de type tableau de 5 éléments de type entier, BYTE est une variable de type tableau de 8 éléments de type "bit" et VECTOR_BIT est une variable de type tableau de 4 éléments de type "bool".

Les éléments d'un tableau seront initialisés soit en désignant un élément particulier, soit en désignant un ensemble d'éléments.

Le tableau VECTOR par exemple, peut être initialisé de la façon suivante :

VECTOR[0]= Y;
VECTOR[1 . . 4]= Z[0 . . 3];

où Y : int;
Z : int ^ 4 ;

Le premier élément est initialisé par la valeur Y et les autres par le vecteur Z.

Une nouvelle version du langage LUSTRE destinée à traiter le problème des tableaux est actuellement en cours de développement. L'introduction de cette notion, qui permet de décrire des structures régulières sans répéter une instruction ou un ensemble d'instructions plusieurs fois, faciliterait la description de matériel.

Exemple :

Revenons à l'exemple de l'additionneur (paragraphe III. 1), l'utilisation d'un tableau pour faire l'addition sur "n" bits permet de simplifier le nœud STRUCTURAL_ADDER. La description de ce nœud devient la suivante, en utilisant le nœud ADDER_BIT décrit précédemment après les modifications nécessaires concernant les déclarations de ses entrées et sorties (on fait appel dans cette description au type bit, proposé plus haut) :

```
node STRUCTURAL_ADDER(X, Y : bit ^ 4)
        returns(Z : bit ^ 4 ; Cout : bit);
var C : bit ^ 5 ;
let
C[0]= 0;
(Z[1 . . 4], C[1 . . 4])= ADDER_BIT(X[1 . . 4], Y[1 . . 4], C[0 . . 3]);
Cout= C[4];
tel.
```

Cette syntaxe est facile à compiler ; toutefois elle ne permet pas de représenter les cas d'irrégularité.

De plus, l'introduction de type tableau pourrait permettre la description de cellules génériques. La description d'une cellule générique peut faire intervenir un paramètre de taille, par exemple, déclaré dans l'interface d'un nœud qui ne sera fixé qu'au moment de l'analyse statique à la compilation. Un registre générique serait défini par un paramètre de type "constante" qui précise le nombre de bits du registre.

Un additionneur générique par exemple est constitué par un nombre quelconque de sous-éléments (cellules additionneur 1 bit, 2 bits, 8 bits, . . .). Un additionneur générique k bits construit à partir des cellules additionneur 1 bit où k est un entier positif serait décrit par :

```
node k_adder [k] (X, Y : bit ^ k)
      returns(Z : bit ^ k; Cout :  bit);
var C : bit ^ k+1 ;
let
C[0]= 0;
(Z[1 . . k], C[1 . . k])= ADDER_BIT(X[1 . . k], Y[1 . . k], C[0 . . k-1]);
Cout= C[k];
tel.
```

Le type énuméré est aussi souhaitable. Il permet à l'utilisateur de définir ses propres objets par un vecteur en énumérant les objets possibles.

Exemple

```
bit : ('0', '1');
signal : ('0', '1', 'X', 'U', 'Z');
logique : ('low', 'high');
```

III. 2. 2 - Introduction de nouvelles valeurs

Pour qu'un langage soit capable de décrire le matériel d'une façon plus précise, il faut qu'il soit capable de représenter toutes les valeurs possibles prises par une variable par exemple (0, 1, X, U, Z) où X représente une valeur indifférente (soit '1', soit '0'), U représente une valeur inconnue conséquence d'un comportement de type transitoire entre 0 et 1, et Z représente une valeur haute-impédance.

A l'heure actuelle, les opérateurs logiques sont définis pour fonctionner sur des variables booléennes. Si on veut utiliser des variables binaires qui peuvent prendre les quatre valeurs possibles (0, 1, X, U) par exemple, il est nécessaire de déclarer ces variables comme des entiers avec des valeurs significatives ordonnées ce qui n'est pas très commode à utiliser et pose le problème de redéfinir toutes les fonctions logiques .

Par exemple, une variable donnée peut être déclarée comme un entier avec des valeurs significatives ordonnées de la façon suivante :

'0' exprimé par la valeur 0
'1' exprimé par la valeur 1
'X' exprimé par la valeur 2
'U' exprimé par la valeur 3

La description d'un opérateur AND, par exemple, sur de telles variables est la suivante :

```
a, b : bool;

.
.
.
c= a AND b;
```

```
a, b : int;

node my_and(A, B : int) returns(S : int);
let
S= if (A=0) or (B=0) then 0
   else if (A=1) and (B=1) then 1
   else if ((A=1) and (B=2)) or ((A=2) and (B=1))
   or ((A=2) and (B=2)) then 2
   else 3;
tel.

.
.
.

node . . .
let
A= a mod 4;
B= b mod 4;
S= my_and(A,B);
tel.
```

Il serait utile que dans une version "description de circuits" de LUSTRE, les valeurs X, U, Z soient prédéfinies et prises en compte naturellement par le langage (pour les variables de type "bit" et de type "entier").

Conclusion

Après avoir défini les différents termes utilisées dans ce texte, nous avons présenté, tout d'abord, le langage VHDL en tant que langage représentatif des langages de description de matériels. Ce langage présente des caractéristiques assez puissantes ce qui a mené récemment à l'adoption de VHDL comme un langage standard pour la description de matériel.

Ensuite, nous avons étudié l'utilisation du langage LUSTRE en tant que langage de description des circuits. Après avoir présenté brièvement ce langage en décrivant ses différents opérateurs, nous avons étudié les différentes caractéristiques de ce langage en faisant une analyse de ses caractéristiques temporelles, en détaillant les différentes possibilités offertes par ce langage ainsi que en décrivant les caractéristiques concernant le niveau et le type d'une description.

Une extension de ce langage finalement est proposée. Cette extension se situe sur deux axes principaux : la définition de nouveaux types primitifs de données et l'introduction de nouvelles valeurs X, U, Z caractéristiques du matériel. L'introduction de deux types nouveaux est proposée : le type "bit" et les types tableau et énuméré.

LUSTRE étant un langage déclaratif, une description LUSTRE est assez proche d'une description type "data-flow" de VHDL. LUSTRE a l'avantage d'avoir une sémantique bien définie. C'est pourquoi nous étudions dans le chapitre suivant la génération de test à partir d'une description LUSTRE. La motivation est que, même si LUSTRE ne s'impose pas en tant que langage de description de circuit, les résultats obtenus seront transposables à une description VHDL "data-flow" ; une traduction d'une description à l'autre est aussi envisageable.

CHAPITRE 2

ANALYSE DE TESTABILITE A PARTIR D'UNE DESCRIPTION EN LUSTRE

INTRODUCTION

Pendant ces dernières années les circuits intégrés à la demande ou ASIC ont reçu un grand succès, connaissant une utilisation de plus en plus importante dans les nouvelles conceptions. Les ASIC permettent d'intégrer une application spécifique sur une puce de silicium, en regroupant sur un seul boîtier toutes les fonctions numériques (ou même analogiques) nécessaires à l'application concernée.

Le concepteur d'un ASIC doit impérativement prendre en compte une *garantie de fonctionnement* de son circuit c'est à dire prévoir la préparation des programmes de test qui assurent le bon fonctionnement de cet ASIC. Cette garantie, surtout pour les ASIC de haut degré de complexité, doit être obtenue par différents outils de simulation et de vérification. Il est essentiel de tenir compte du problème de test dès les premières étapes de la conception et non plus à posteriori. Il est en particulier nécessaire de concevoir des circuits testables.

Intuitivement, la notion de testabilité d'un circuit (ou un système) vise à évaluer la facilité de construire un programme de test et la performance de ce programme. La testabilité intègre plusieurs notions :

La ***contrôlabilité*** caractérise la facilité de mettre chaque nœud du circuit à une valeur logique donnée à partir des entrées primaires.

L'***observabilité*** représente la facilité de déterminer la présence d'une valeur défectueuse sur le nœud concerné en examinant les sorties primaires.

Les mesures de contrôlabilité et d'observabilité sont souvent combinées pour fournir une mesure de ***testabilité*** au niveau du nœud ou du circuit [ROB 88]. La qualité de test sera donnée par la facilité de contrôler et observer chaque élément matériel du système. Les résultats d'évaluation de testabilité ont pour but d'infléchir la conception en proposant des modifications au cours même du processus de conception.

Le ***taux de couverture*** d'un test est défini par le pourcentage des fautes détectées par ce test par rapport à toutes les fautes possibles. Ce taux de couverture est relatif à la classe de fautes prises en compte.

La ***diagnosabilité*** : ce concept est complémentaire de la testabilité. Il s'agit d'apprécier la qualité de résolution du programme de test c'est à dire d'évaluer le degré de localisation des éléments matériels défectueux. Un haut degré de diagnosabilité doit ainsi faciliter la correction de la conception du circuit.

Le processus de spécification de programmes de test d'un circuit complexe (système) accompagne les étapes d'analyse de la testabilité et de la diagnosabilité ; il comporte deux phases :

- la décomposition en sous-structures facilement testables afin de rendre le test de ces circuits faisable.

- la recherche des algorithmes ou des méthodes de génération de stimuli de test, à partir d'une description du circuit (structurelle ou comportementale).

La complexité croissante des ASIC rend l'utilisation des générateurs automatiques de programmes de test (ATPG : Automatic Test Program generators) inefficace pour les circuits conçus actuellement [CRA 89]. Deux approches pour résoudre ce problème de test peuvent être distinguées :

- les techniques de conception de circuits facilement testables [BRE 85]. Ces méthodes sont telles que le test de ces circuits est indépendant de leur fonction, mais effectué par des dispositifs spécialisés en introduisant un second mode d'opération "mode de test". Parmi ces méthodes, on peut citer les techniques comme le LSSD, "Scan Path", "Scan/Set", "Random Access Scan" et le test intégré BILBO. Ces techniques permettent de diviser le circuit en parties que les ATPG peuvent traiter.

- les méthodes de génération de test basées sur une description fonctionnelle du circuit à tester.

Basé sur ce principe, notre approche est de générer le test à partir d'une description du circuit dans le langage de haut niveau LUSTRE. En effet, LUSTRE, qui décrit aussi bien la structure que le comportement, permet d'effectuer une partition d'un circuit en blocs fonctionnels (les nœuds de la description) qui peuvent être mis à plat en fusionnant les différent blocs en une description unique constituée par un ensemble de primitives interconnectées. Cette description facilite la phase de génération de test parce que les propagations avant/arrière nécessaires pour le test ainsi que l'analyse de test se font alors sur ce réseau, similaire à un réseau de portes mais à un niveau de description plus élevé.

Ce chapitre est organisé en trois parties. La première est consacrée aux méthodes développées dans le domaine de la génération de test. La deuxième est consacrée à la liaison entre LUSTRE en tant que langage de description de matériels de haut-niveau et l'outil d'évaluation de testabilité SATAN. Cette liaison se fait par la traduction des différentes primitives du langage en graphe SATAN associé pour l'évaluation de la testabilité et l'analyse de test d'un circuit. La troisième est consacrée à l'introduction des notions permettant la génération des séquences de test.

I - L'ETAT DE L'ART SUR LE TEST FONCTIONNEL

Le test peut être :

- un test de fin de conception pour rechercher des erreurs de conception ;
- un test de fin de fabrication assurant un bon rendement et vérifiant les spécifications données par le cahier des charges ;
- un test de validation ou de maintenance. Ce test est effectué pour assurer une garantie de fonctionnement tout au long de la durée de vie du produit.

Les applications de haute sécurité/haute fiabilité exigent des produits extrêmement fiables, or le test de fin de conception et de fin de fabrication s'avère toujours insuffisant [GOU 85]. Par conséquent, le développement des nouvelles méthodes de test est primordial. Ces méthodes doivent améliorer la qualité de test des circuits LSI/VLSI en les rendant facilement testables, et dans certains cas mener à intégrer des dispositifs de test dès la conception.

I. 1 - Méthodes de test

Les méthodes de test des circuits LSI/VLSI complexes sont très nombreuses [BEL 84], [SU 84]. Différentes classifications des méthodes de test ont été proposées dans la littérature [SU 84], [SAU 83].

On peut distinguer deux catégories de méthodes de test :

a) test aléatoire [DAV 76]
b) test déterministe.

En général, le test comporte les étapes suivantes :

- la génération des séquences de test qui sont souvent appelées *vecteurs de test* ;
- l'application de ces séquences au circuit à tester ;
- l'évaluation des réponses obtenues.

Le test déterministe se différencie essentiellement du test aléatoire par la méthode de génération des vecteurs de test. Les vecteurs d'un test déterministe sont calculés au cours d'une étude préalable du circuit, alors que pour un test aléatoire, les vecteurs sont générés au moment du test effectif.

Nous nous intéressons au test déterministe ; dans cette catégorie deux grandes familles de méthodes peuvent être identifiées [BEL 84] : le test par distinction (où des hypothèses d'erreurs sont prises en compte à priori) et le test par identification (sans hypothèses d'erreurs à priori).

Un ***test par distinction*** a pour but de distinguer un circuit conforme à ses spécifications de l'ensemble des circuits erronés. L'ensemble des circuits erronés est défini par les hypothèses d'erreurs.

Un ***test par identification*** vérifie que le circuit testé fonctionne correctement en identifiant son fonctionnement à ses spécifications.

On présentera les méthodes de génération de test proposées dans la littérature pour les cas suivants :

a) Test des circuits combinatoires : on distingue ici entre test généré à partir d'une représentation structurelle par l'interconnexion des portes logiques [ROT 66], et test généré à partir d'une représentation fonctionnelle [AKE 78].

b) Test des circuits séquentiels (automates) : on distingue ici entre test par distinction [POA 63] et test par identification [KOH 78]. Le test d'automate est présenté dans le chapitre 3 de ce texte, en particulier le test d'automate par identification.

c) Test des circuits complexes à partir d'une description de haut niveau : on distingue ici entre test généré à partir d'une description structurelle de haut niveau et test généré à partir d'une description comportementale.

I. 2 - Méthodes de génération de test des circuits combinatoires

Les méthodes de génération de test des circuits combinatoires basées sur une représentation structurelle (réseau de portes) sont essentiellement des méthodes de chemin sensible, dérivées en général du D_algorithme [ROT 66]. Par la suite, des algorithmes ont cherché à optimiser les phases de propagation avant/arrière : PODEM [GOE 81], FAN [FUJ 83].

Cependant, il existe des méthodes de génération de test basées sur une représentation fonctionnelle, utilisant un graphe appelé : diagramme de décision binaire, pour la modélisation des circuits [AKE 78]. Le diagramme de décision binaire est schématisé par un arbre binaire associé à une fonction : l'arbre représente tous les fonctionnements possibles en fonction des valeurs des entrées. Les nœuds de l'arbre sont les variables d'entrée et aux arcs sont associées les valeurs possibles (0, 1). Ensuite, la génération de vecteurs de test des fonctions utilisées se fait à partir de ce diagramme, avec des hypothèses de collage des arcs ou des nœuds du diagramme. Les travaux ultérieurs portent principalement sur les hypothèses de pannes envisagées.

I. 3 - Méthodes de génération de test des circuits complexes

Les méthodes de génération de test des circuits complexes peuvent être classées en trois catégories suivant le type de description qu'elles utilisent :

a) Méthodes de génération de test basées sur une description structurelle : elles consistent à générer le test à partir d'une description structurelle d'un circuit complexe, dont la structure est connue et décrite en utilisant des primitives de haut niveau d'un langage de description de matériel [LEV 82]. On utilise des algorithmes classiques de chemin sensible appliqués à ces primitives et à leur interconnexion, figure (II-1, a). Ces méthodes, qui nécessitent la connaissance interne du circuit, ne sont malheureusement pas applicables aux réseaux importants de primitives complexes, car elle nécessite d'associer à chaque primitive des expressions algébriques complexes [LEV 82], [LEV 83].

b) Méthodes de génération de test basées sur une description comportementale : elles consistent à générer le test à partir d'une description comportementale au niveau Transfert de Registre, qui décrit l'algorithme effectué par le circuit. On peut utiliser par exemple un graphe d'états interprété qui modélise toutes les transformations de données/contrôle effectuées par cet algorithme, figure (II-1, b), [BEL 84].

c) Méthodes de génération de test basées sur une description pseudo-structurelle souvent appelées : méthodes de test hiérarchisée. Le circuit à tester est décrit comme un réseau de blocs interconnectés, les blocs étant définis par l'utilisateur. La génération de test est alors à deux niveaux : traitements des blocs et traitement de leur interconnexion. On utilise, en général, les méthodes classiques de propagations avant/arrière sur ces interconnexions vers les entrées et les sorties primaires, figure (II-1, c). On distingue ici entre deux types suivant la nature de la description des blocs : description fonctionnelle impérative ou déclarative.

Par la suite, on présente quelques méthodes représentatives de chaque type de test.

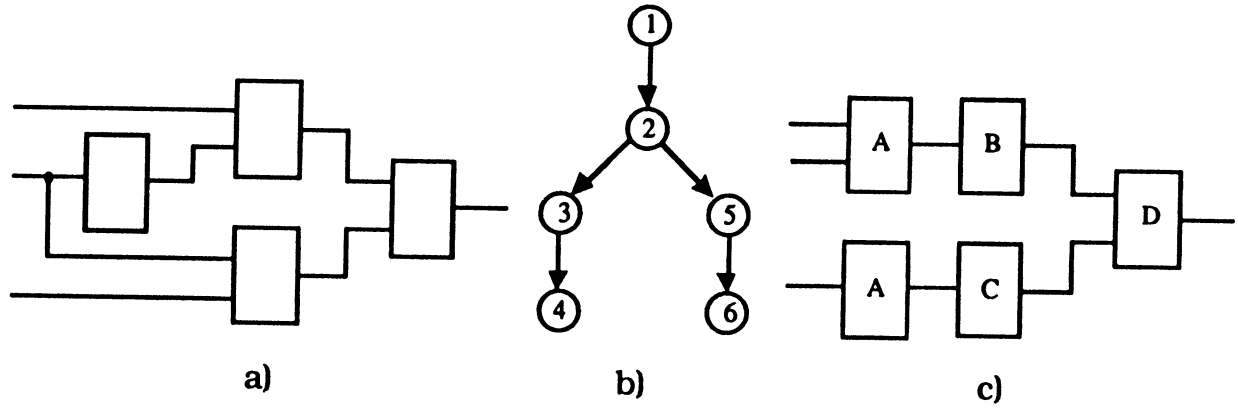


Figure II-1 : a) interconnexion de primitives ; b) graphe d'états interprété ; c) interconnexion de blocs décrits par l'utilisateur.

I. 3. 1 - Génération de test à partir d'une description structurelle

Les méthodes de génération de test à partir d'une description structurelle de haut niveau du circuit sont basées sur l'utilisation d'un algorithme classique de chemin sensible à l'origine appliqué au niveau logique. L'utilisation de primitives de haut-niveau accélère les propagations, mais ne permet pas de prendre en compte le test de ces primitives.

La génération du programme de test se décompose en trois phases : injection de la faute dans le système, propagation de la faute vers une sortie observable et justification de la faute vers les entrées contrôlables.

Breuer et Friedman [BRE 80] ont réalisé une extension du D-algorithme conçue pour être applicable aux primitives de haut niveau, de type registre, additionneur, registre à décalage. A chaque primitive est associée l'ensemble des informations nécessaires à la propagation avant et arrière. Cette méthode n'est plus applicable dans le cas des systèmes très complexes.

Levendel & Menon [LEV 82] ont proposé une méthode fondée sur l'utilisation des primitives d'un langage de description de haut niveau CHDL.

Cette méthode est basée sur l'utilisation du D-algorithme [LEV 82] et du *-algorithme [LEV 83].

Le D-algorithme, basée sur la sensibilisation de chemins, nécessite la description du système à tester sous forme d'un ensemble de D-cubes (un D-cube décrit un chemin sensible dans une primitive, permettant de propager une valeur D vers les sorties).

En [LEV 83], les mêmes auteurs utilisent un autre algorithme nommé *-algorithme pour assurer la génération complète du test. Cet algorithme est basé sur la sensibilisation des entrées critiques ; une entrée X_i d'une fonction $Z=f(X_1, X_2, \ldots, X_i, \ldots, X_n)$ est dite critique par rapport à une sortie Z, si un changement de valeur de X_i provoque un changement de valeur de Z. Un cube critique est l'ensemble maximal de valeurs d'entrées critiques. Les cubes critiques seront obtenus, soit à partir de leur définition par rapport aux D-cubes, soit à partir d'un système d'équations critiques appelés *-équations pour les opérateurs booléens de base. Un *-cube (équivalent d'un D-cube) sera associé à chaque primitive du langage.

Exemple

Soit un circuit composé d'un multiplexeur exprimé de la façon suivante :

$$Z = A\,B + \bar{A}\,C$$

où A, B, C sont des booléens.

La propagation d'une valeur D (valeur erronée 1 au lieu de 0) à travers de ce multiplexeur vers la sortie Z est décrite de la façon suivante :

$$Z^D = A^1\,B^D + A^0\,C^D + A^{\bar{D}}\,B^0\,C^1 + A^D\,B^1\,C^0$$

Cette expression montre qu'il y a quatre possibilités de D-propagation unique à travers de ce multiplexeur :

- A= 1 et B erroné (1 au lieu de 0) ;
- A= 0 et C erroné (1 au lieu de 0) ;
- B= 0, C=1 et A erroné (0 au lieu de 1) ;
- B= 1, C=0 et A erroné (1 au lieu de 0) ;

Cette expression peut être obtenue systématiquement à partir de l'expression de la fonction du circuit, à l'aide de manipulations algébriques, en appliquant des règles du type :

pour un OU $c^D = (a + b)^D = a^D b^0 + a^0 b^D + a^D b^D$

pour un ET $c^D = (a\ b)^D = a^D b^1 + a^1 b^D + a^D b^D$

$c^0 = (a\ b)^0 = a^0 + b^0 + a^D b^{\overline{D}} + a^{\overline{D}} b^D$

La génération du programme du test en utilisant le *-algorithme se décompose également en trois phases : *-propagation (c'est à dire construction d'un chaînage de cubes critiques), implication et justification.

Cette méthode, qui consiste à adapter le D-algorithme ou *-algorithme à des descriptions matérielles de haut niveau, peut être appliquée à tout type de circuit, à contrôle centralisé ou distribué. Par contre, elle nécessite des manipulations sur des expressions algébriques complexes pour déterminer les D-cubes ou les *-cubes. Les exemples traités en [LEV 82], [LEV 83] sont de faible complexité (blocs type compteur 4 bits, décodeur 1 vers 4, additionneur, . . .) et cette méthode ne semble pas applicable aux réseaux importants de primitives complexes.

I. 3. 2 - Génération de test à partir d'une description comportementale

La génération d'un *test comportemental* se fait à partir des fonctions réalisées par le circuit, indépendamment de sa structure interne. Les méthodes de test comportemental consistent à générer le test à partir d'une description de l'algorithme exécuté par le circuit.

Lin et Su dans [LIN 84] proposent l'utilisation des techniques d'exécution symbolique à travers des chemins d'activation appelés "function submodule" (FS). Le processus de génération du test consiste à injecter un ensemble de fautes fonctionnelles dans la description de l'algorithme. Les fautes fonctionnelles proposées ont neuf types. Ces types sont réduits à cinq par l'élimination des fautes équivalentes. Les cinq types de fautes sont sur les

conditions, les branchements, le transfert de données, le décodage des registres et des opérateurs.

La génération du test consiste à comparer les résultats fournis par l'exécution symbolique d'une machine correcte et d'une machine dans laquelle est introduite une faute [LIN 85-a]. Un arbre binaire appelé arbre d'exécution symbolique (SET : Symbolic Execution Tree) est défini. L'exécution symbolique est utilisée pour construire le SET associé à une machine correcte et le SET associé à une machine défectueuse.

Exemple

Considérons les trois instructions suivantes au niveau RTL :

5 : R1 ← R2 + R3 → 6 (l'instruction suivante)
6 : R3 ← shr R1 → 7
7 : R2 ← R3 - 1 → 0

Après l'exécution de l'instruction 7, la valeur symbolique du registre R2 sera shr($R2 + $R3) - 1. Cette valeur est représentée par l'arbre suivant :

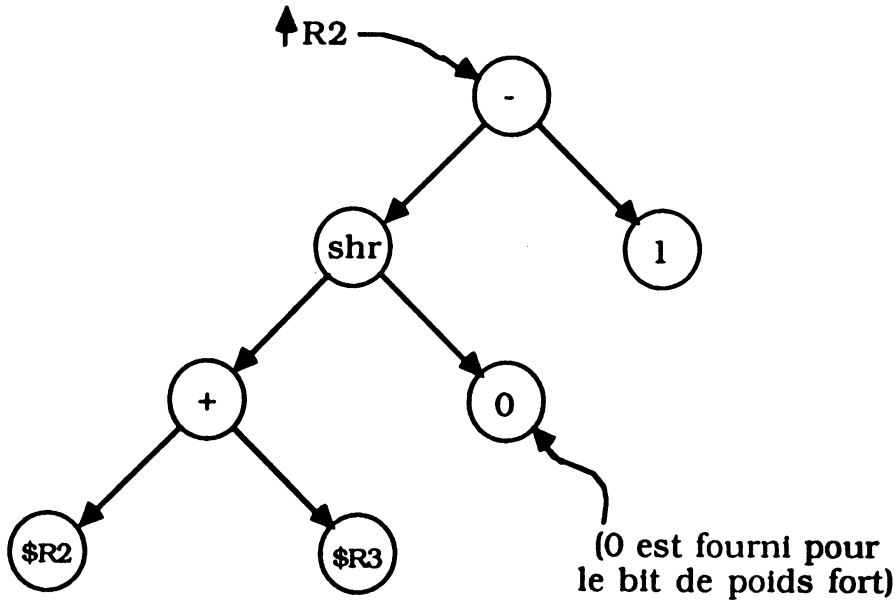


Figure II-2 : Arbre représentant la valeur symbolique du résultat.

où $R2, $R3 sont les valeurs symboliques initiales des registres R2 et R3 respectivement.

↑R2 représente la valeur symbolique courante de R2.

Une erreur α pourrait être "l'addition est transformée en soustraction", ce qui donne la description suivante de la machine fausse :

5α : R1 ← R2 - R3 → 6
6 : R3 ← shr R1 → 7
7 : R2 ← R3 - 1 → 0

Pour détecter l'erreur, il faut déterminer des valeurs initiales telles que ↑R2 ≠ ↑R2 α.

Un algorithme global de génération du test est présenté dans [LIN 84] et [LIN 85-b]. Il se décompose en trois phases :

- pré-processeur : qui analyse la description et qui détermine les FS et les chemins de couverture ;
- processus principal appelé "S-algorithm" (Symbolic algorithm) : qui, pour une FS donnée, détermine les vecteurs de test ;
- post-processeur : qui vérifie que toutes les FS, les instructions et les fautes ont été traitées.

Un générateur de test a été réalisé. Cependant, la description utilisée a le défaut de présenter de multiples occurrences de variables ou d'opérateurs. Un modèle de pannes réaliste devrait être un modèle de fautes multiples (la même faute dans chaque occurrence).

Cette méthode a été appliquée à un ensemble de "function submodule" d'un calculateur simple. Les auteurs estiment pouvoir appliquer cette méthode à des systèmes digitaux LSI/VLSI.

I. 3. 3 - Génération de test à partir d'une description pseudo-structurelle

Les méthodes de génération de test à partir d'une description pseudo-structurelle consistent à utiliser les méthodes classiques de propagations

avant/arrière sur une description de haut niveau. Suivant le type (impératif ou déclaratif) de description des blocs interconnectés, le problème de génération de test est différent :

- si les blocs sont décrits dans un langage impératif, il est nécessaire de traiter par deux méthodes différentes les blocs eux-mêmes et leur interconnexion (de nature déclarative) comme proposé en [RAR 85].

- si les blocs sont décrits dans un langage déclaratif, des méthodes classiques de chemin sensible sont capables de traiter à la fois les blocs eux-mêmes et leur interconnexion.

On présentera deux méthodes, l'une basée sur une description initiale impérative, l'autre sur une description VHDL de type "data-flow".

a) Méthode proposée par Lai & Siewiorek

Lai & Siewiorek ont proposé une méthode qui peut s'appliquer à n'importe quel système digital, en particulier à un microprocesseur [LAI 83]. A partir d'une description impérative de l'exécution du jeu d'instructions, une première étape consiste à déterminer une description "data-flow" : graphe de transformation d'état (STG : State Transformation Graph) auquel est associé un ensemble de variables d'états. Ce graphe représente le flot d'information dans le système modélisé sous forme d'un graphe orienté. Les arcs représentent les chemins de données/contrôle entre les nœuds. Les nœuds représentent les opérateurs de transformation de données.

Les modèles de fautes introduits sont multi-niveaux. Ils se situent : soit au niveau du graphe de transformation d'états (erreur sur un arc) : ces fautes sont dites de niveau macro (haut niveau) ; soit au niveau de primitives de description ; alors ces fautes sont dites de niveau micro (niveau inférieur).

Exemple

Le graphe de transformation d'états d'un ordinateur à trois instructions est montré dans la figure II-3. Une instruction comporte 16 bits dont deux bits de code opération et une adresse de 14 bits ; les trois instructions

sont : add (addition de l'accumulateur avec un mot mémoire), st (rangement la valeur de l'accumulateur en mémoire) et jump (branchement).

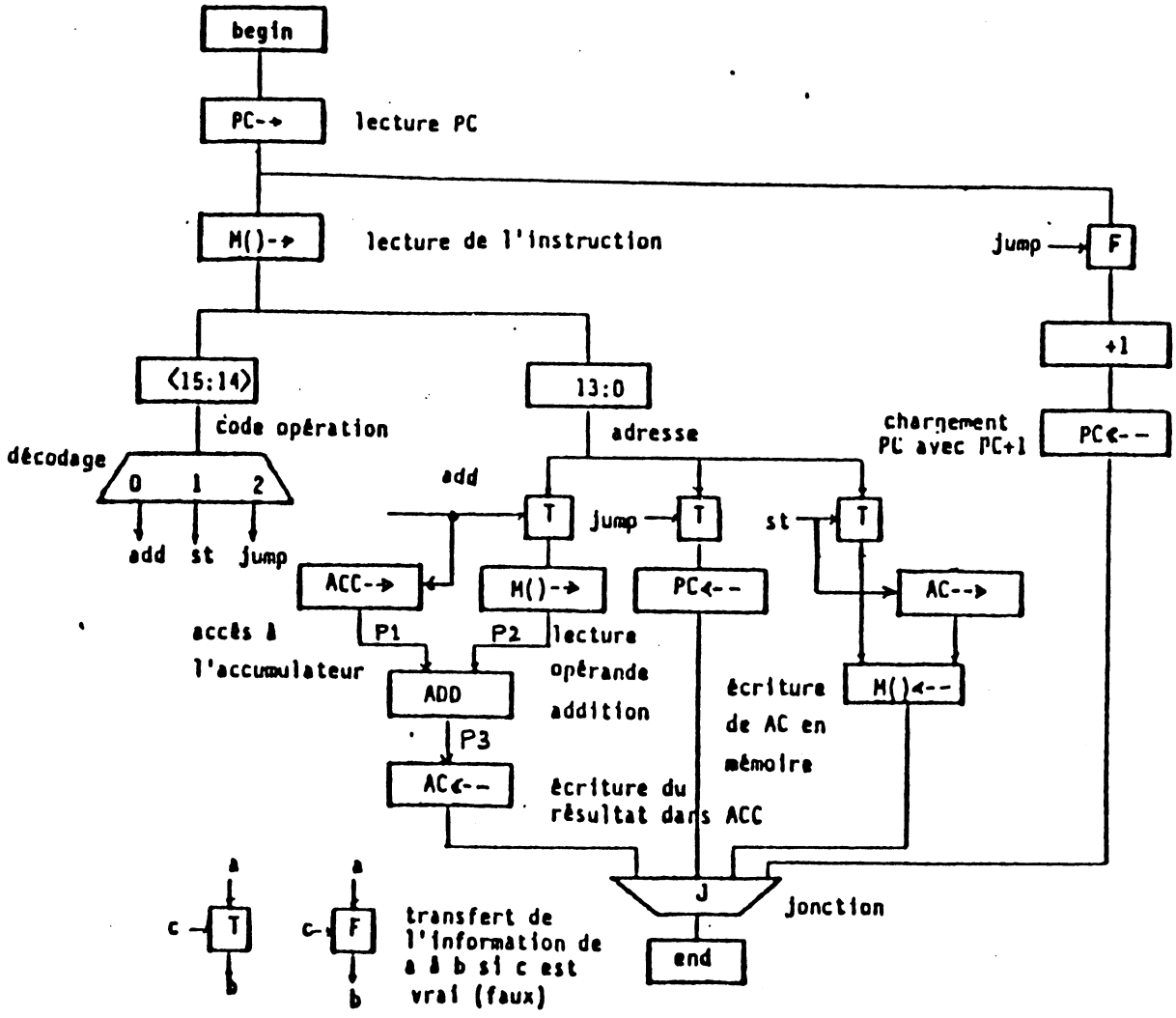


Figure II-3 : Le "STG" : graphe de transformation d'états.

La génération de programme de test fonctionnel proposée se décompose en cinq phases :

* Introduction des paramètres de test : cette phase introduit des valeurs symboliques qui représentent, d'une part, les paramètres de test d'une faute, et d'autre part, les résultats de test de la faute (dans la figure II-3 par exemple, P1, P2, P3 pour une faute dans l'additionneur).

* Implication : cette phase se fait par une propagation arrière du paramètre de test, situé au niveau de la faute, vers une source, où l'ensemble des nœuds et des arcs traversés constitue un chemin sensible, et par une propagation en aval du résultat de test, situé au niveau de la faute, vers un puits. Les propagations arrière et avant des paramètres de test et de résultat de test délivrent des contraintes de sensibilisation du chemin de test (add= 1, donc <15:14>= 00 dans notre exemple).

* Justification : cette phase a pour rôle de spécifier les valeurs des chemins non affectés par les deux propagations avant et arrière précédentes.

* Synthèse de test : la synthèse d'un test défini par les trois phases précédentes, a pour rôle de substituer les paramètres symboliques de test (au niveàu source) par des vecteurs de test récupérés dans une base de données.

* Synthèse du programme de test : la synthèse du programme de test (pour le cas d'un microprocesseur) consiste, en regroupant les instructions en classes (même mode d'adressage, même nombre d'opérandes, . . . etc), à générer un programme de test écrit en assembleur qui optimise le nombre d'instructions et le nombre de vecteurs de test.

L'originalité de cette méthode consiste à ne parler que de variables ou paramètres de test : la propagation est indépendante des valeurs effectives des vecteurs de test, et donc de la sensibilisation de la faute. La recherche de la sensibilisation du chemin de test va par contre déterminer les contraintes sur les entrées de contrôle. En plus, cette méthode peut être utilisée dès les premières phases de la conception d'un circuit même sans savoir sa structure.

Cette méthode a été appliquée à l'unité centrale d'un calculateur PDP-8, en considérant les collages simples sur les chemins de données. Le taux de détection des erreurs par l'application de programme de test de 731 instructions obtenu avec cette méthode (détection de 98% de pannes de collages des registres) était supérieur à celui du programme du fabricant d'environ 10 000 instructions.

Cependant, l'inconvénient majeur du graphe de transformation d'état de Lai & Siewiorek est que un même bloc matériel est représenté de

nombreuses fois dans le graphe. Ces multiples occurrences de chaque bloc matériel complique la génération de programme de test (pour un seul bloc en panne, il est nécessaire de prendre un modèle de fautes multiples).

b) Méthode proposée par Armstrong & Barclay

Armstrong & Barclay proposent la génération de test sur un "modèle VHDL "data-flow" du circuit à tester [ARM 86]. Leur méthode consiste d'une part, à définir un ensemble de modèles de fautes, et d'autre part, à générer le test proprement dit.

Les modèles de fautes sont basés sur deux principaux types de fautes : les erreurs de contrôle et les erreurs de micro-opération (régles de perturbation d'un modèle) [ARM 88]. Les modèles de fautes de perturbations d'une micro-opération sont les plus cruciaux et délicats. Pour modéliser la perturbation d'une micro-opération, il faut déterminer une nouvelle micro-opération de façon à obtenir le meilleur recouvrement possible des fautes pouvant intervenir dans une micro-opération.

La génération de test se fait en quatre phases :

- Détermination des fautes associées à chaque primitive : les modèles de fautes suivants sont utilisés : une micro-opération VHDL est erronée et transformée en une autre opération, une instruction d'affectation est erronée (pas d'opération), une instruction "if" est fausse (collage à "then" ou collage à "else") ou une clause d'une instruction "case" est fausse (pas d'opération).

- Génération du test fondamental : la génération du test est basée sur l'utilisation des méthodes classiques de test (injection d'une faute, sensibilisation de la faute par la propagation avant/arrière). Le test fondamental consiste à réaliser différents buts ou objectifs. Ces buts font introduire l'arbre de but "Tree Goal" décrit en dessous.

- Résolution des buts : un but est décomposé en un ensemble de sous-buts jusqu'à l'obtention de buts primaires faciles à réaliser (lecture sur les entrées primaires ou écriture sur les sorties primaires).

- Extraction d'un vecteur de test : parmi les solutions obtenues par la phase précédente, il faut choisir un vecteur de test ; certaines solutions peuvent être contradictoires, donc non réalisables.

La recherche des buts se fait par un arbre de buts ET/OU qui modélise les objectifs à effectuer. La stratégie utilisée est de partitionner un grand problème en sous-problèmes (stratégie diviser pour régner). Dix types de buts peuvent être utilisés dans lesquels un objet représente un signal donné.

Cet algorithme est réalisé pour quelques primitives du langage VHDL : if, case-of et des affectations seulement [ARM 86]. Deux types de données seulement sont utilisés : les booléens et les entiers. Les types de structure de données utilisés sont seulement les signaux scalaires. Les expressions sont composées par des opérations sur les entiers et les booléens en utilisant des attributs pour le changement de données ou pour leur retard. L'auteur prévoit l'ajout d'autres types, structures ou fonctions dans le futur. L'algorithme est implémenté en Prolog ; la source VHDL est traduite en une représentation intermédiaire. Cette représentation structurée sous forme d'un arbre est le résultat de l'analyseur VHDL. Cette représentation est traduite en ensemble de prédicats Prolog.

Un générateur de test comportemental (BTG : "Behavioral Test Generator") est proposé en [ONE 89]. Ce générateur est basé sur la méthode précédente. Il se décompose en deux phases :

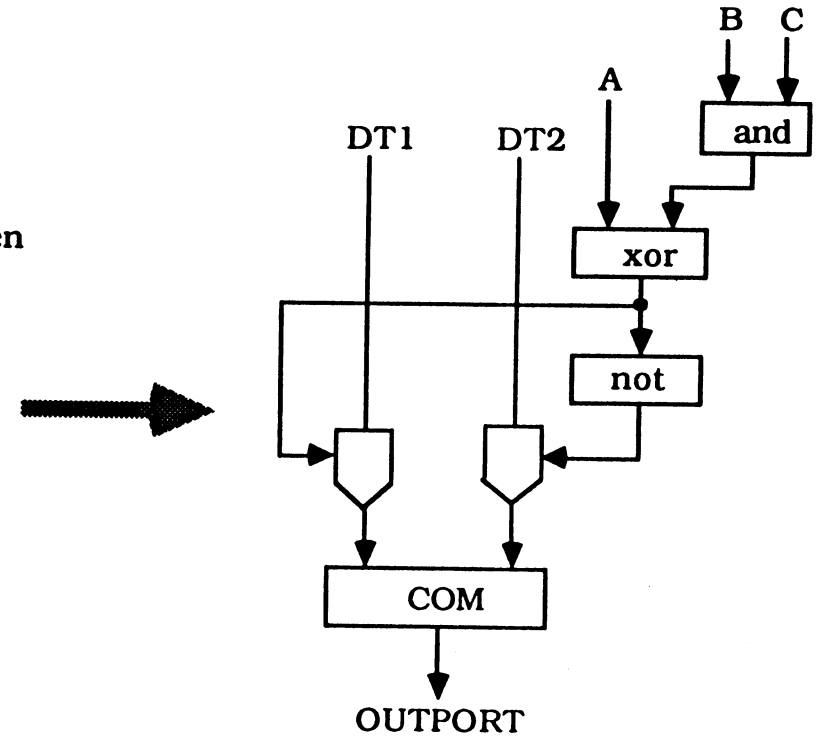
- pré-processeur : qui a le rôle de traduire les sources VHDL en prédicats Prolog et de construire une liste de fautes au niveau comportemental.

- algorithme de génération de test : qui contient des règles pour les trois buts considérés suivants : justification, exécution et propagation. Il génère un test pour chaque faute dans la liste en utilisant le format obtenu après la traduction de la phase précédente.

Une extension de la méthode de Armstrong par un algorithme noté E-algorithme [NOR 89] est proposée. Cette extension du D-algorithme permet de propager les syndromes d'une faute à travers les structures de données et de justifier les valeurs requises pour sensibiliser cette faute. Cette extension

utilise un modèle graphique pour les opérateurs utilisés en considérant deux types de fautes : défaut pour les opérations fonctionnelles d'un bloc et les fautes de collage pour les lignes de contrôle ou de données. Un exemple de traduction d'une description textuelle est montré dans la figure II-4.

```
if (A xor (B and C)) then
   COM<= DT1;
else
   COM<= DT2;
endif;
OUTPORT<= COM;
```



a) description VHDL b) modèle graphique

Figure II-4 : Traduction d'une description textuelle en modèle graphique.

Ce générateur de test comportemental est intéressant pour deux raisons principales. D'abord, il est fondé sur une recherche de la signification et de la définition de fautes fonctionnelles. Les perturbations de modèle proposées ont été validées en estimant leur taux de couverture par rapport au modèle de collage sur une représentation équivalente en portes logiques. D'autre part, l'utilisation d'un langage de type "data-flow" et les techniques d'arbres de buts et des représentations graphiques pour une description textuelle est prometteuse pour la génération du test.

Néanmoins, cette technique n'a pas été pour l'instant appliquée sur des circuits réellement complexes.

Conclusion

Parmi les nombreuses méthodes de génération de test fonctionnel qui ont été proposées, celles qui ont mené à la réalisation d'un système de génération de test capable de traiter des circuits relativement complexes et/ou à fonctionnement parallèle sont basées sur des descriptions de type data-flow, soit à l'origine (Armstrong & alt.), soit par traduction d'une description impérative (Lai & Siewiorek). Il semble donc qu'étudier la génération de test de circuits complexes à partir d'une description LUSTRE, qui est un langage "data-flow", puisse mener à un outil de génération de test efficace.

Nous étudierons ici une première étape : l'analyse de testabilité à l'aide de l'outil SATAN. Ce logiciel, à partir d'un graphe biparti qui représente une interconnexion de modules élémentaires, détermine les ensembles minimaux de modules activés pour un transfert d'information des entrées primaires aux sorties primaires ; un tel ensemble est appelé écoulement. SATAN détermine aussi l'ensemble des écoulements nécessaires pour activer tous les modules du circuit, ce qui constitue une spécification de son test.

Après avoir présenté le système SATAN, la traduction d'une description LUSTRE en modèle SATAN est étudiée, et le logiciel de traduction qui a été réalisé est présenté. Le modèle SATAN est un graphe abstrait : nous proposons de le compléter en introduisant la temporisation des écoulements. Nous étudions ensuite comment les valeurs des entrées de contrôle du circuit nécessaires à l'activation d'un écoulement pourraient être déterminées.

On obtient ainsi des activations de test d'un circuit, parametrées comme dans la méthode de (Lai & Siewiorek). Il ne reste alors qu'à effectuer la synthèse du test en utilisant un outil classique sur une description niveau logique. L'utilisation du langage LUSTRE (au lieu du STG) permet d'effectuer la génération du test sur un modèle plus réaliste du circuit.

II - TRADUCTION LUSTRE-SATAN

La méthode de génération de test fondée sur un modèle du circuit à tester, décrit par un langage de haut niveau, à partir des différentes primitives de ce langage est une méthode prometteuse parce qu'elle offre des caractéristiques intéressantes : le nombre de primitives est limité et la traduction de ces primitives est faisable. Pour cela plusieurs chercheurs se sont orientés vers l'utilisation de ce principe pendant ces trois dernières années (voir la méthode proposée par Armstrong et al.).

Notre objectif est d'automatiser la liaison entre LUSTRE en tant que langage de description de matériels de haut-niveau et l'outil d'évaluation de testabilité SATAN : il s'agit donc, à partir d'un fichier de description d'un circuit en LUSTRE dans lequel la carte ou le circuit est considéré comme un ensemble de composants, d'obtenir automatiquement le modèle de testabilité du système SATAN dans une première phase, puis le programme de test nécessaire dans une deuxième phase.

Dans un premier temps (première phase), nous présenterons la compilation des différents opérateurs primitifs du langage LUSTRE sous la forme d'un réseau d'opérateurs (qui peut se présenter d'une façon graphique par un graphe des différents opérateurs primitifs). On présentera ensuite, l'outil d'aide à la testabilité SATAN modélisé par un graphe abstrait. On étudiera ensuite, la traduction du réseau d'opérateurs d'un circuit en graphe SATAN associé. Cette traduction se fait en définissant un ensemble de règles de traductions.

Dans le paragraphe suivant (paragraphe III), nous introduirons des notions permettant la génération des séquences de test à partir du graphe de testabilité SATAN obtenu (deuxième phase) : introduction du temps, contraintes sur les lignes de contrôle et propagation de ces contraintes dans l'écoulement (un chemin d'information à travers le système électronique considéré). Le programme de test est constitué par les séquences de valeurs

des entrées primaires (les entrées contrôlables) permettant de couvrir l'ensemble des écoulements du système.

II. 1 - Compilation LUSTRE en réseau d'opérateurs

Une description textuelle LUSTRE d'un circuit est constituée par un ensemble d'opérateurs primitifs connus du langage et par des fonctions définies par l'utilisateur. Cette description peut être compilée en réseau d'opérateurs qui représente les composants ou les opérateurs du circuit ainsi que la connectique entre ces composants ou ces opérateurs.

Les opérateurs primitifs du langage LUSTRE (Cf. II. 2) peuvent être distingués en trois catégories suivant leur fonctionnalité :

- opérateur de multiplexage ;
- opérateurs arithmétiques ou booléens ;
- opérateur "pre".

Dans la suite, on présentera ces trois types en généralisant leurs représentations.

a) Opérateur de type multiplexage (contrôle logique)

Cet opérateur permet suivant la valeur du contrôle logique c de choisir une des deux entrées disponibles à l'entrée de celui-ci : e_1 et e_2. Cette structure est décrite en LUSTRE par une construction "if-then-else", figure II-5.

s = if c then e_1
else e_2.

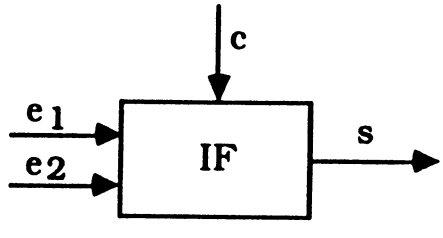


Figure II-5 : Primitive de type multiplexage.

b) Opérateurs arithmétiques ou booléens (de type transfert de donnée)

Ces opérateurs primitifs permettent de transférer, de traiter ou de manipuler des données sans condition spécifique. Des opérateurs usuels (ou fonctions) avec une (comme par exemple s= not e) ou plusieurs entrées de données (comme par exemple s= e_1 + e_2) peuvent être des opérateurs arithmétiques ou booléens, figure II-6.

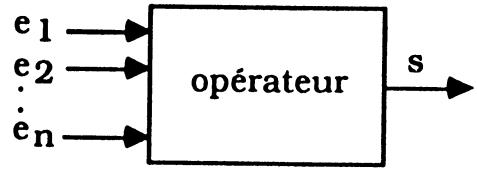


Figure II-6 : Opérateur arithmétique ou booléen.

c) Opérateur "pre"

L'opérateur "pre" (précédent) de LUSTRE référence la valeur de son argument à l'instant précédent. Cet opérateur est un opérateur sur les suites et permet de pouvoir faire référence au passé. Il permet par conséquent de distinguer deux portions distinctes du circuit, la première correspondante à l'instant précédent et la deuxième à l'instant courant.

La variable Y, par exemple, égale à X mais décalée un instant de temps :

Y= init -> pre(X).

L'opérateur "->" (suivi de) qui est considéré comme un opérateur conditionnel et temporel sert à initialiser les variables.

Le réseau d'opérateurs, construit en utilisant ces trois types d'opérateurs primitifs possibles du langage, exprime schématiquement les opérations effectuées. Il est constitué par des nœuds et des connexions. Les nœuds représentent, soit des opérateurs primitifs de LUSTRE comme not, and, or, pre, -> et if, soit des fonctions ou des sous-réseaux définis par l'utilisateur. Ces nœuds sont étiquetés par leurs opérateurs primitifs ou leurs fonctions.

Les connexions relient les sorties d'un nœud à des entrées d'autres nœuds ou à des sorties primaires et les entrées de ce nœud à des sorties d'autres nœuds ou à des entrées primaires. Ces connexions représentent les chemins de données/contrôle entre les nœuds. Elles peuvent être étiquetées par des identificateurs. Elles sont associées à des variables ou à des signaux.

Ce réseau d'opérateurs peut se présenter d'une façon graphique par une interconnexion structurelle d'opérateurs. Dans le graphe obtenu, chaque variable apparait une fois contrairement au graphe de transformation d'états de Lai & Siewiorek étudié auparavant.

L'outil de génération du réseau d'opérateurs de LUSTRE est un des utilitaires LUSTRE déjà implantés.

Exemple :

La description textuelle suivante décrit un additionneur 1 bit. Le nœud "XOR" est défini avec des paramètres formels. Le nœud "ADDER_bit" référence le nœud "XOR" deux fois avec ses paramètres effectifs.

```
node XOR(A, B : bool) returns(C : bool);
let
C= (A and not B) or (not A and B);
tel.

node ADDER_bit(X, Y, Cin : bool) returns(SUM, Cout : bool);
var S : bool;
let
S= XOR(X, Y);
SUM= XOR(S, Cin);
Cout= (X and Y) or (S and Cin);
tel.
```

Le réseau d'opérateurs obtenu associé à cette description textuelle est le suivant :

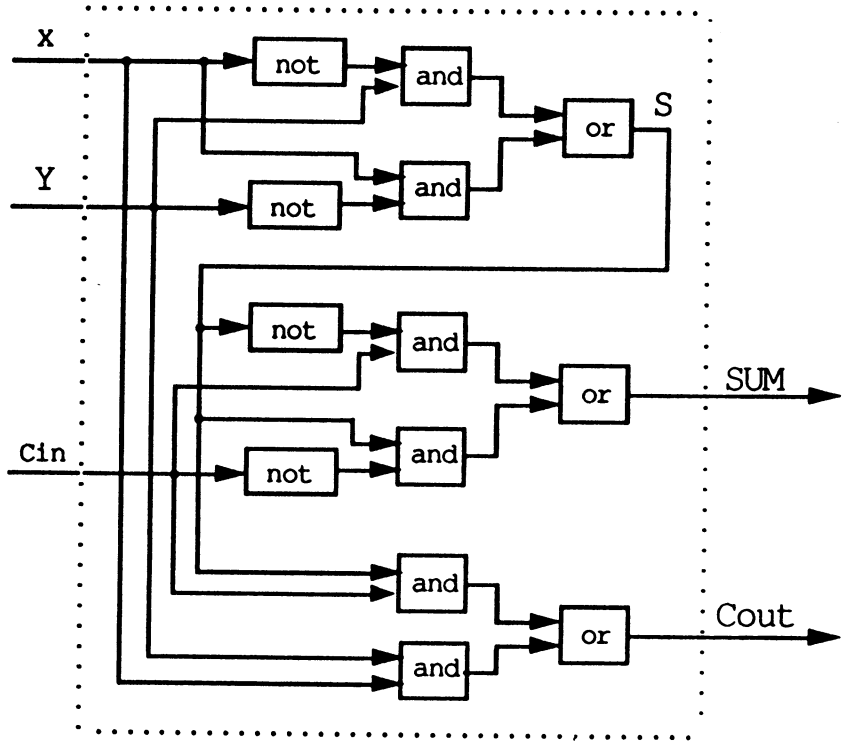


Figure II-7 : Réseau d'opérateurs obtenu pour un additionneur 1 bit.

II. 2 - Outil d'aide à l'analyse de testabilité : Système SATAN

Le système SATAN sert à l'évaluation de testabilité d'un circuit. Cette évaluation est fondée sur une représentation graphique : le modèle de testabilité, qui décrit les transferts d'information entre composants. Le concept de base de cette représentation est la partition au niveau fonctionnel en modules tel que chaque module réalise une fonction (ou un ensemble de fonctions) élémentaire.

Le modèle de testabilité SATAN est un graphe abstrait graphique biparti orienté. Il permet d'évaluer la testabilité d'une carte et montre les transferts de données entre les composants de celle-ci. Ce graphe caractérise tous les fonctionnements possibles d'un composant en terme de transfert de données, ce qui détermine les conditions nécessaires pour les activations fonctionnelles globales. Ces activations fonctionnelles sont considérées comme chemin de données à partir des entrées primaires jusqu'aux sorties primaires [ROB 88].

Le modèle SATAN permet, d'une part, d'évaluer de façon descendante la testabilité, de manière à obtenir des performances et des coûts optimaux de test (et ce au cours même de la conception), et d'autre part, d'infléchir cette conception ou de proposer des modifications en particulier par insertion de points de test pour augmenter les possibilités d'accès.

Le modèle SATAN est entièrement défini par un ensemble de places, de transitions et d'arcs. Une place est soit un module qui caractérise l'état interne du composant ou exécute une fonction spécifique, soit un nœud terminal du graphe. Les nœuds terminaux correspondent aux entrées primaires (nœuds sources) ou aux sorties primaires (nœuds puits).

Une transition entre deux places caractérise les conditions et le mode de transfert de données entre ces deux places. Cette transition noté T est décrite par : / liste des places prédécesseurs / liste des places successeurs /. Les arcs connectant places et transitions représentent les supports de données.

Quatre modes de transfert de données peuvent être considérés; leur description graphique et textuelle en SATAN sont données figure II-8.

a) Mode de jonction

Les données des modules sources S_1, S_2, . . ., S_n sont toutes nécessaires pour le fonctionnement du module destination D, figure (II-8, a). La transition T_j de ce mode est définie, alors, par T_j= / S_1, S_2, . . ., S_n / D /.

b) Mode d'attribution

Le fonctionnement du module destination D nécessite seulement une donnée d'un module parmi les modules sources S_1, . . ., S_n, figure (II-8, b). Les transitions T_{t1}, T_{t2}, . . ., T_{tn} sont définies par T_{t1}= / S_1 / D / ; . . . ; T_{tn}= / S_n / D /.

c) Mode de distribution

La donnée issue du module source S n'est observable qu'à travers tous les modules destinations $D_1, \ldots, D_n$, figure (II-8, c). La transition T_d de ce mode est définie par T_d= / S / $D_1, D_2, \ldots, D_n$ /.

d) Mode de sélection

La donnée issue du module source S est observable à travers chacun des modules destinations $D_1, \ldots, D_n$, figure (II-8, d). Les transitions $T_{s1}, \ldots, T_{sn}$ sont définies par T_{s1}= / S / D_1 / ; . . . ; T_{sn}= / S / D_n /.

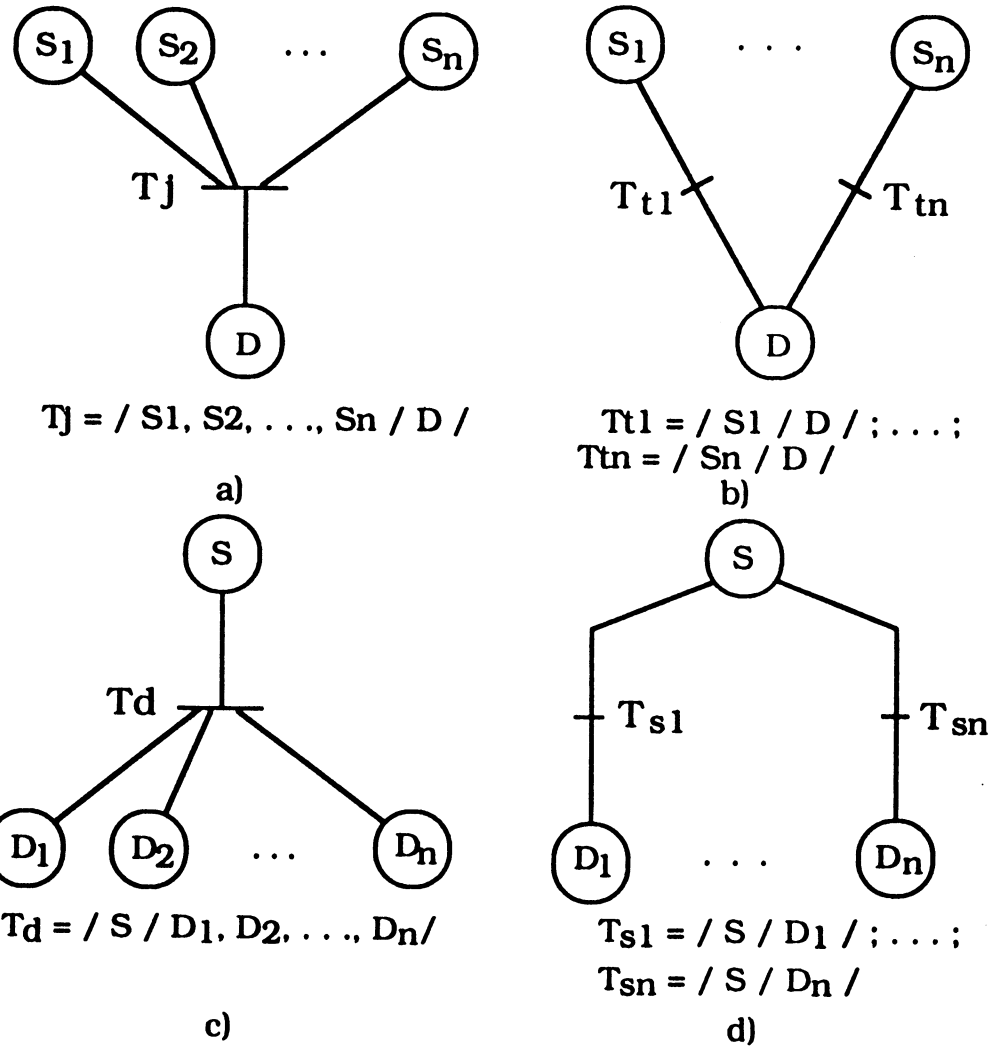


Figure II-8 : a) mode de jonction. b) mode d'attribution.
c) mode de distribution. d) mode de sélection.

Un modèle générique correspond au modèle de testabilité intrinsèque d'un composant. Il représente tous les fonctionnements élémentaires de celui-ci ainsi que les transferts de données effectués.

Par exemple, le modèle générique de SATAN d'un registre dont les commandes sont : reset, chargement (ch) et horloge H, où E est l'entrée de donnée du registre, est le suivant [WOD 89] :

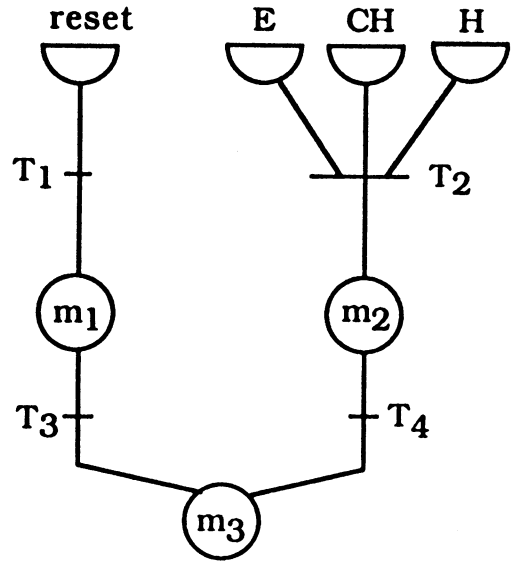


Figure II-9 : Modèle générique d'un registre.

II. 3 - Objectif du travail

L'utilisation de système SATAN, à l'heure actuelle, pose plusieurs problèmes aux différents niveaux. Il existe, à l'heure actuelle, un lien entre le système de C.A.O. SDS de Silvar Lisco et l'outil de testabilité SATAN. Le système SDS est fondé sur deux langages : HHDL et SDL ; le premier est utilisé pour la modélisation du comportement de circuits ; le second, SDL, est utilisé pour décrire l'interconnexion entre modules.

Donc, on peut utiliser le langage SDL pour décrire les connexions entre modules (par un fichier net_list ou on peut utiliser également un logiciel graphique) mais il faut créer une base de donnée SATAN, (bibliothèque de modèles génériques), manuellement écrite en respectant la syntaxe SATAN et

en spécifiant toutes les transitions possibles où chaque transition est décrite par : / liste des places prédécesseurs / liste des places successeurs /. Cette bibliothèque de modèles génériques contient la description de tout composant pouvant être utilisé lors de la conception.

Notre objectif est de faire une traduction automatique, directe et complète des différents opérateurs primitifs du réseau d'opérateurs obtenu après la compilation d'une source décrite par le langage LUSTRE (qui décrit aussi bien la connectique que le comportement) vers le graphe de testabilité SATAN associé.

La traduction en graphe de testabilité SATAN va consister à définir un ensemble de règles de traduction de différents opérateurs primitifs ou de connexions du réseau d'opérateurs. Cette méthode est basée sur le fait que le nombre d'opérateurs d'un langage de haut niveau est fixe et donc on peut définir l'ensemble de régles de traduction pour obtenir directement les graphes de testabilité associés.

II. 3. 1 - Règles élémentaires de traduction

Trois règles de traduction sont définies. Les règles 1 et 2 sont appelées règles de traduction d'opérateurs primitifs vers les graphes de testabilité SATAN correspondants tandis que la règle 3 est une règle de traduction d'interconnexion au vu de la connectique de la carte ou du circuit qui va permettre de concaténer les graphes obtenus précédemment.

II. 3. 1. 1 - Première règle de traduction

Pour la primitive de type multiplexage de LUSTRE (primitive "if-then-else"), figure (II-10, a), qui permet d'aiguiller les données, la traduction en forme de graphe de testabilité SATAN va consister à générer deux chemins d'information ayant même puits et dont les sources sont définies par la condition et les données associées au chemin considéré.

Ces deux chemins d'information sont distincts, l'un pour les données sélectionnées par la valeur "vrai" de la variable de contrôle, l'autre pour les données sélectionnées par la valeur "faux" de la même variable. Donc, deux

modules SATAN m_1 et m_2 ayant pour transitions prédécesseurs T1 et T2 où T_1 = / e_1, c / m_1 / et T_2 = / e_2, c / m_2 /, et pour transitions successeurs T3 et T4 où T_3 = / m_1 / s / et T_4 = / m_2 / s /, sont introduits pour permettre de spécifier les fonctions de cet opérateur.

La sortie s du graphe SATAN associé à cette primitive qui représente le multiplexage de deux entrées e_1 et e_2, figure (II-10, b), est réalisée par l'attribution de deux valeurs. Chacune de ces valeurs est exprimée par la jonction de la condition d'activation et de l'entrée correspondante.

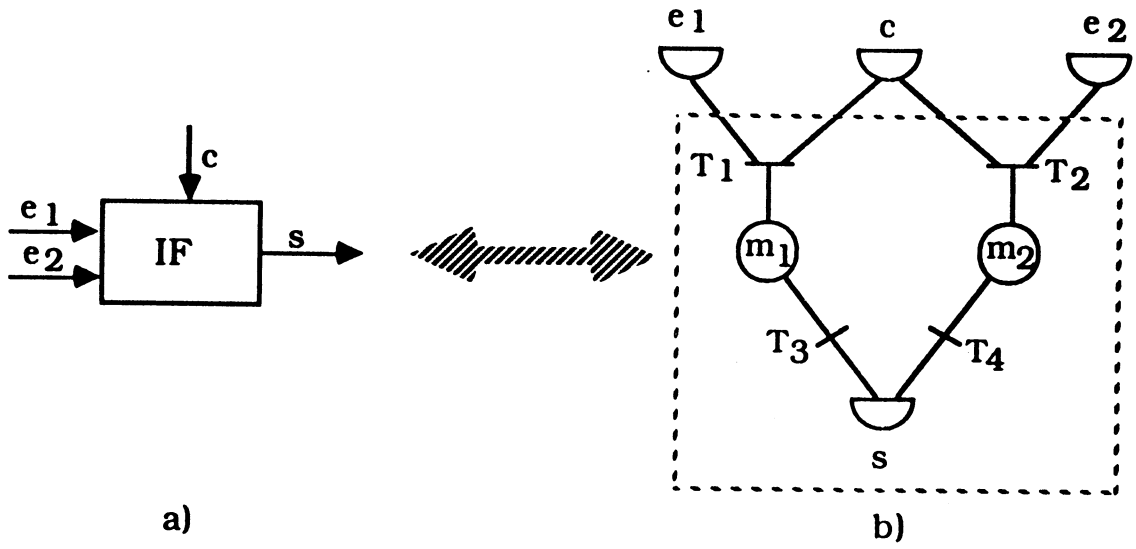


Figure II-10 : Première règle de traduction.

Une réduction de ce graphe SATAN associé est réalisée en analysant la ligne de contrôle et les données associées ce qui dérive un modèle réduit (les règles de traduction de l'opérateur de multiplexage avec valeur de contrôle constante ou valeurs de données d'initialisation seront discutées dans le paragraphe II. 3. 2).

II. 3. 1. 2 - Deuxième règle de traduction

Pour les opérateurs du type transfert de donnée, la traduction d'une fonction ou d'un opérateur à une ou plusieurs entrées, va consister à créer une transition dont les places prédécesseurs sont les opérandes de l'opérateur et ayant une place successeur formée par la sortie correspondante. La sortie "s" d'un opérateur (figure II-11, a) est réalisée par la jonction des entrées

$e_1, e_2, \ldots, e_n$, figure (II-11, b). La transition T est définie, alors, par $T= / e_1, e_2, \ldots, e_n / s /$.

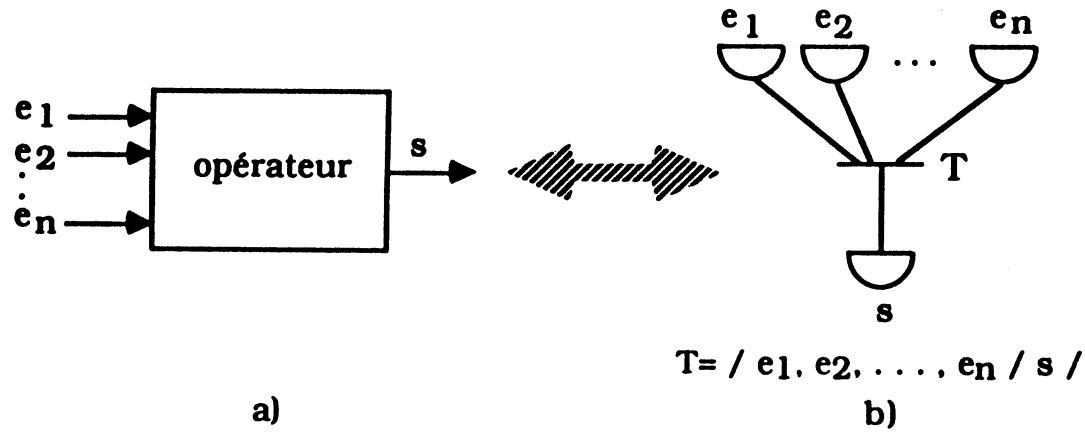


Figure II-11 : Deuxième règle de traduction.

Remarque :

L'opérateur de retard de LUSTRE "pre" (troisième type d'opérateurs de LUSTRE) est traduit, pour l'instant, comme un simple opérateur de transfert de données par une seule transition T où T= / e / s /, figure II-12. Cette traduction ne considère pas l'aspect temporel de cet opérateur qui est de retarder la variable concernée. Dans le paragraphe suivant (Cf. III. 1), une quatrième règle de traduction est introduite concernant la prise en compte du temps.

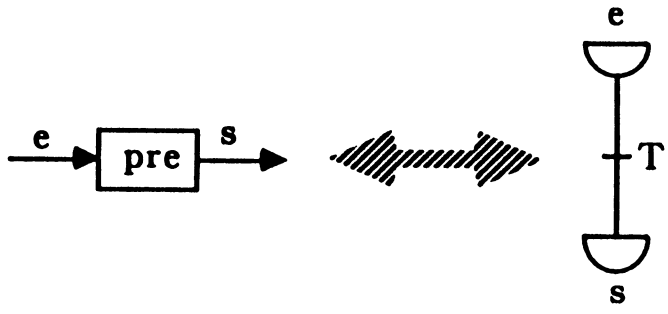


Figure II-12 : Traduction non temporelle de l'opérateur pre.

II. 3. 1. 3 - Troisième règle de traduction

Cette règle concerne l'interconnexion des différents opérateurs de type transfert de donnée. Cette interconnexion est prise en compte en spécifiant un chemin d'information pour chaque destination. Par exemple, si le résultat d'un opérateur op_i est émis vers plusieurs opérateurs : $op_1, op_2, \ldots, op_n$, figure (II-13, a), le graphe SATAN associé à cet ensemble d'opérateurs est montré dans la figure (II-13, b). Il est défini par la création d'un module m_i ayant pour transition prédécesseur T_i où T_i= / e / m_i /, et pour transitions successeurs $T_1, T_2, \ldots, T_n$ où T_1= / m_i / S_1 /, . . ., T_n= / m_i / S_n /.

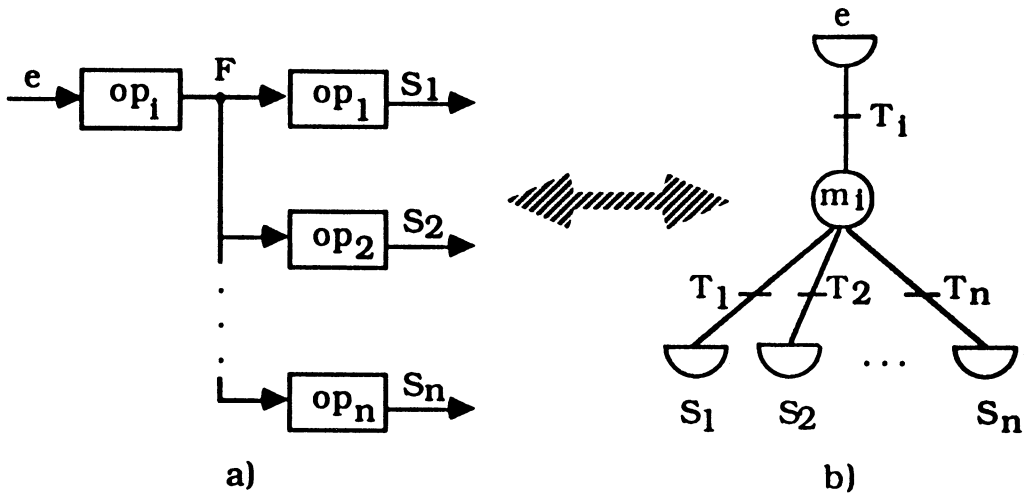


Figure II-13 : Troisième règle de traduction.

où $S_1, S_2, \ldots, S_n$ représentent respectivement $op_1(F), op_2(F), \ldots, op_n(F)$.

Cette interconnexion est représenté en modèle SATAN par la sélection d'un des chemins possibles correspondant aux opérateurs $op_1, op_2, \ldots, op_n$.

II. 3. 2 - Règle d'optimisation

Le graphe SATAN obtenu peut être optimisé en prenant en compte l'utilisation : soit de valeurs de contrôle logique constantes, soit de valeurs des variables d'initialisation en utilisant des règles spécifiques de traduction nommées règles d'optimisation.

Une ligne de contrôle qui vaut une valeur constante (vraie ou fausse) peut impliquer une réduction du domaine de fonctionnement total ce qui induit un sous-modèle représentant les fonctions réellement effectuées (un fonctionnement partiel) à partir du graphe SATAN initial.

II. 3. 2. 1 - Utilisation des valeurs de contrôle logique constantes

L'utilisation d'une ligne de contrôle d'une primitive de multiplexage de type "if-then-else" qui vaut une valeur constante, c="true" (resp. c="false") implique la réduction de la première règle de traduction et l'élimination d'une fonction complète ce qui donne un sous-modèle minimisé où la sortie s est égal à e_1 (resp. à e_2) comme il est montré dans la figure suivante :

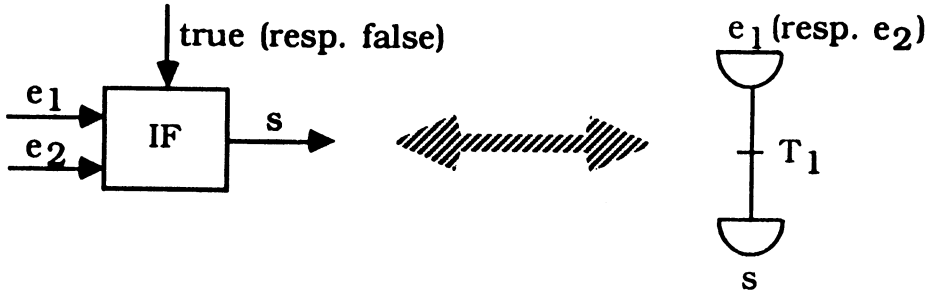


Figure II-14 : Minimisation de la première règle de traduction.

II. 3. 2. 2 - Utilisation des valeurs de données d'initialisation

Le graphe de testabilité SATAN d'un registre, figure II-15, ayant un signal de remise à zéro "reset" est montré dans la figure II-16.

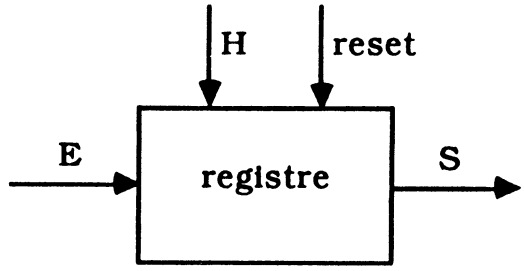


Figure II-15 : Registre simple ayant un signal de remise à zéro.

On distingue deux fonctionnements principaux du registre, un correspond à la remise à zéro du registre (quand le signal de "reset" est actif), l'autre correspond au chargement du registre par l'horloge H et l'entrée D. Donc, le graphe de testabilité SATAN est constitué par l'attribution de deux chemins de données correspondants aux deux fonctionnements ainsi évoqués.

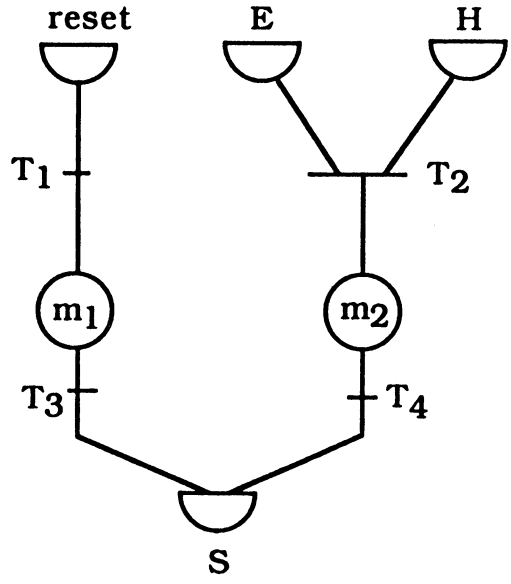


Figure II-16 : Modèle de testabilité SATAN d'un registre simple.

La description fonctionnelle LUSTRE de ce registre nécessite pour réaliser le "reset" de pouvoir initialiser un nœud. La réinitialisation d'un nœud en LUSTRE doit être explicite.

Par exemple, si on écrit un nœud COMPTE qui à chaque instant prend la valeur précédente plus 1 comme :

```
node COMPTE(inc : int) returns(x : int);
let
x=0-> pre(x) + inc;
tel;
```

La valeur 0 ne concerne que l'instant initial de fonctionnement. Si on veut être capable de le remettre à zéro en cours de fonctionnement, sa description sera la suivante :

```
node COMPTE-R(raz : bool; inc : int) returns(x : int);
let
x=0-> if raz then 0
        else pre(x) + inc;
tel;
```

Donc, pour effectuer cette réinitialisation, on a utilisé une primitive "if-then-else" avec une valeur interne constante égale 0. Cette valeur est considérée interne parce que la validation de la condition raz implique le transfert immédiat de cette valeur.

La description textuelle du registre figure II-15 est la suivante ; on a utilisé deux primitives "if-then-else" pour décrire les deux fonctionnements du circuit :

```
node Front(H : bool) returns(trans : bool);
let
trans= false-> H and not pre(H);
tel.

node Registre(reset, H : bool; D : int) returns(S : int);
let
S=  if reset then 0
    else if front(H) then D
         else 0-> pre(S);
tel.
```

Le réseau d'opérateurs obtenu pour le graphe SATAN de la figure II-16, est montré dans la figure II-17, (le réseau d'opérateurs pour le nœud Front(H) a été omis).

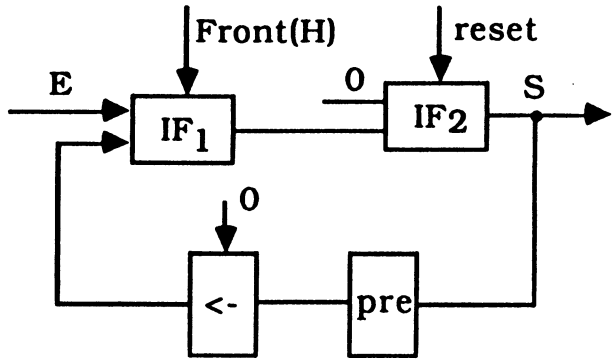


Figure II-17 : Réseau d'opérateurs obtenu.

La prise en compte du signal de reset de la primitive IF2 de cet exemple, montré dans la figure (II-18, a), qui correspond à initialiser le registre, provoque des modifications sur la 1ère règle de transformation où la valeur interne est ignorée. Le graphe de testabilité SATAN est montré dans la figure (II-18, b).

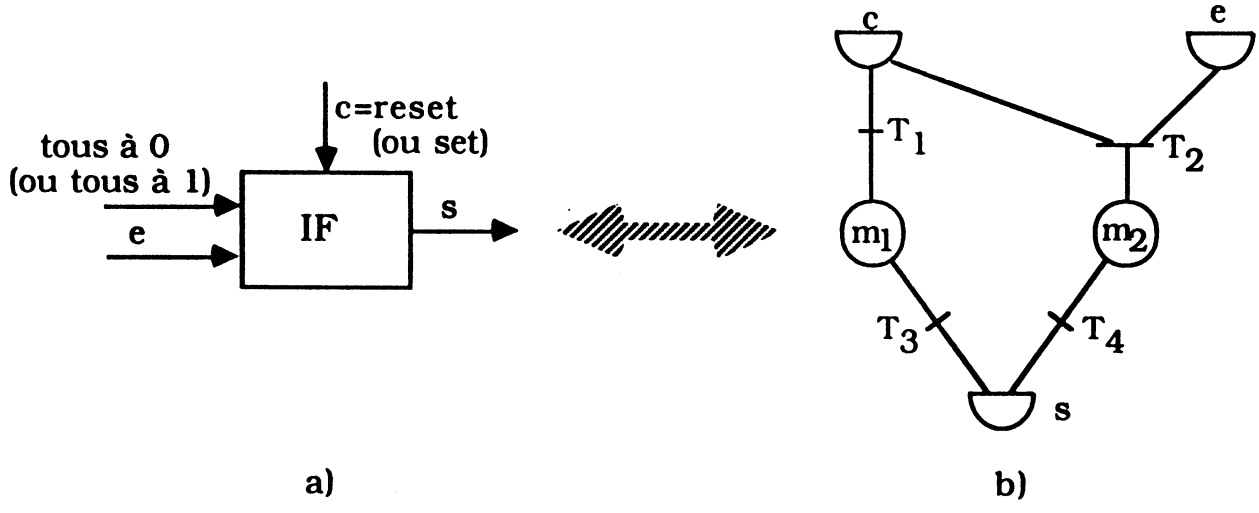


Figure II-18 : Règle de traduction réduite.

L'utilisation d'une valeur de contrôle logique constante du signal précédent (par exemple, reset= vrai de l'exemple précédent) donne un simple

générateur d'un mot égal à 0 et le modèle devient une source générant une valeur immédiate, figure II-19.

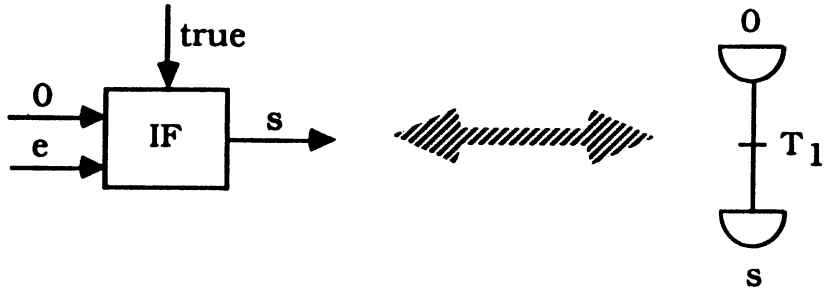


Figure II-19 : Générateur d'un mot égal à 0.

II. 3. 2. 3 - Utilisation des valeurs constantes de données

L'utilisation d'un opérateur primitif arithmétique ou booléen (de type transfert de donnée) avec une entrée constante autorise la réduction de la deuxième règle de traduction ; on ne tient pas compte de cette entrée, figure II-20.

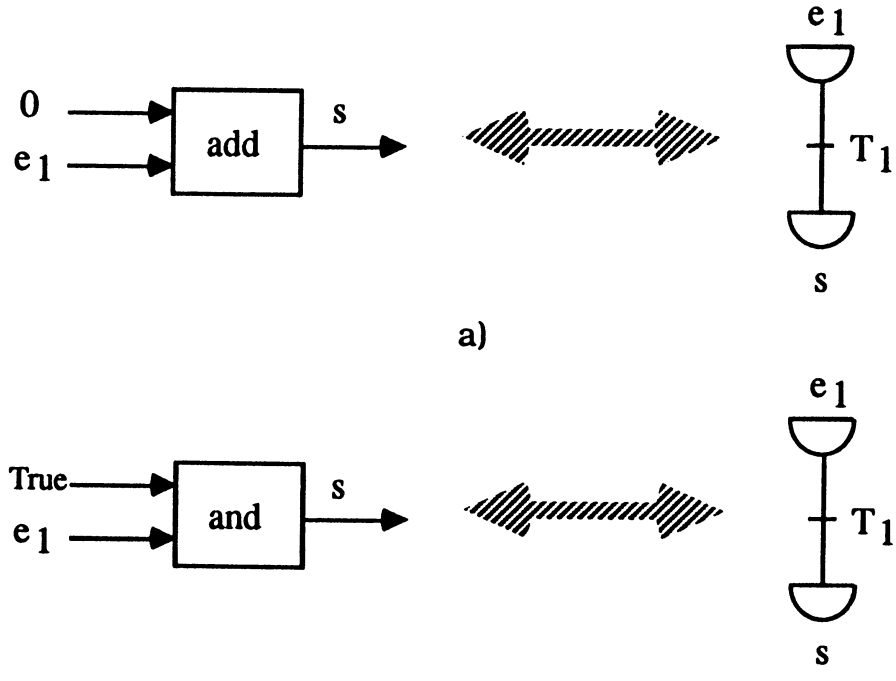


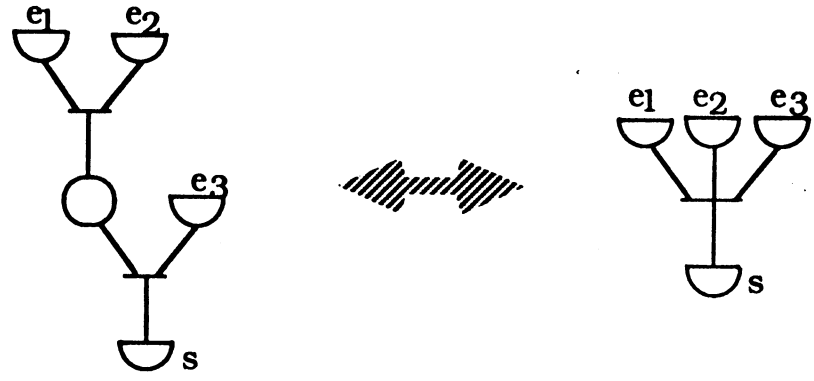
Figure II-20 : Minimisation de la deuxième règle de traduction.

Remarque :

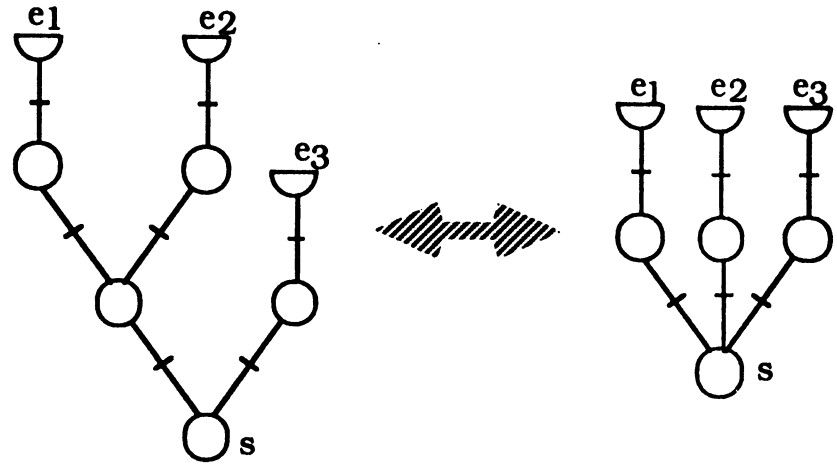
Le modèle final obtenu d'après l'utilisation des règles élémentaires de traductions et les règles d'optimisation peut être simplifié en utilisant les deux règles suivantes :

* la jonction de deux valeurs où la première est composée de la jonction de deux autres valeurs est considérée comme la jonction de ces trois valeurs, figure II-21, a.

* l'attribution de deux valeurs où la première est composée de l'attribution de deux autres valeurs est considérée comme l'attribution de ces trois valeurs, figure II-21, b.



a)



b)

Figure II-21 : Règles de simplification.

II. 3. 3 - Ordre d'application des règles de traduction

L'ordre d'application pour les règles de traduction précédentes d'une description textuelle compilée en forme d'un réseau d'opérateurs est primordial.

D'abord, il faut commencer par la traduction des différentes connexions au vu de la connectique de la carte considérée. Ceci va permettre de concaténer les graphes de testabilité SATAN qui nous allons avoir.

Ensuite, il faut traduire les opérateurs primitifs du réseau d'opérateurs (les boîtes d'un réseau d'opérateurs) en graphes de testabilité SATAN correspondants (les règles 1 et 2) en considérant éventuellement les minimisations possibles.

II. 3. 4 - Obtention d'un modèle de testabilité et d'écoulements

On peut obtenir le modèle de testabilité de la carte en partant de ses primitives ; c'est à dire rechercher le graphe de testabilité SATAN pour chaque opérateur primitif de la carte en considérant la connectique entre ces opérateurs primitifs. Une connexion entre deux graphes SATAN représentant deux primitives connectées est effectuée par le regroupement des puits du graphe qui représentent les sorties de la première primitive et des sources du graphe qui représentent les entrées de la primitive connectée. Ce regroupement est effectué par la transformation d'un ensemble de puits-sources connecté en un simple module.

Le graphe de testabilité final SATAN obtenu précise les fonctionnements globaux de la carte et permet de trouver les écoulements ou les chemins d'information de cette carte. Un écoulement, qui correspond à un fonctionnement spécifique de la carte, est déterminé à partir des sorties externes "observables" du modèle de testabilité puis en remontant à travers les différents modules jusqu'à atteindre les entrées externes "contrôlables".

Un écoulement (ou une activation fonctionnelle) E est un ensemble stable de places et de transitions par rapport aux opérations suivantes :

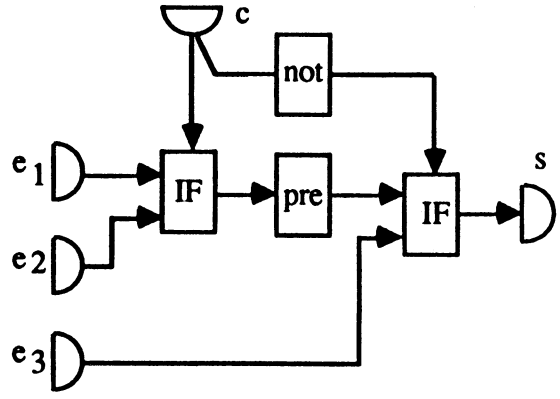
- si une transition appartient à un écoulement, toutes les places prédécesseurs et successeurs y appartiennent ;

- pour toute place de l'écoulement, il existe un chemin dans l'écoulement allant d'une source à un puits via cette place.

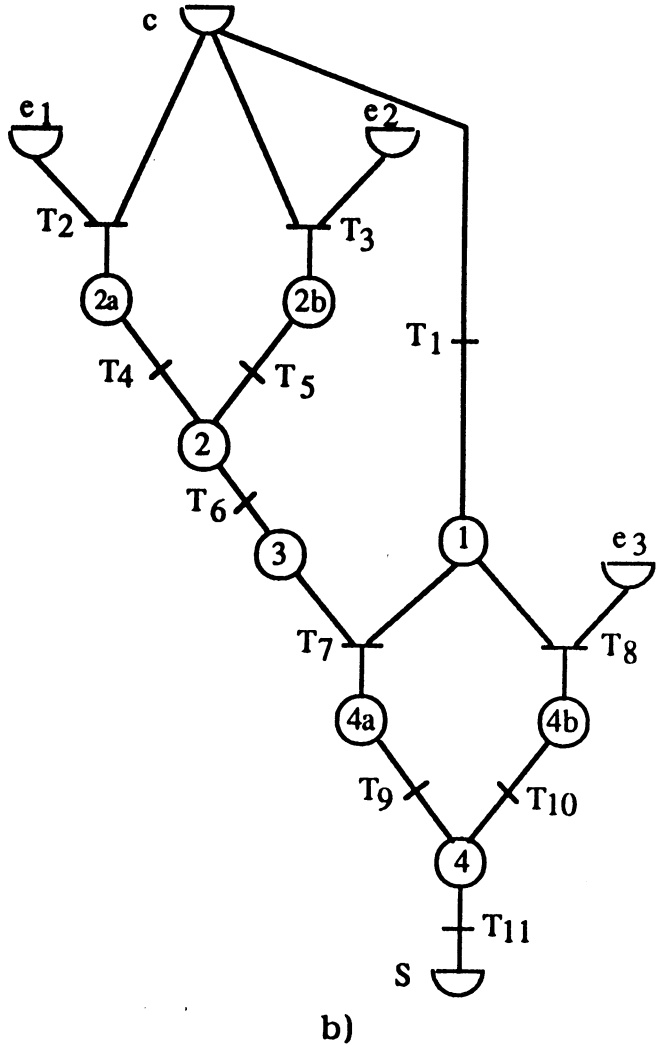
Un écoulement correspond à la notion intuitive de sous-ensemble matériel pouvant fonctionner complètement et isolément du reste du système.

Exemple

Soit le réseau d'opérateurs montré dans la figure (II-22, a). La traduction de ce réseau en considérant les trois régles de traduction et les régles d'optimisation va donner le graphe de testabilité montré dans la figure (II-22, a).



a)



b)

Figure II-22 : a) Réseau d'opérateurs ; b) Graphe SATAN obtenu.

Trois écoulements distincts sont obtenus en appliquant le système SATAN sur le graphe de testabilité obtenu après la traduction. Ces écoulements sont montrés dans la figure II-23.

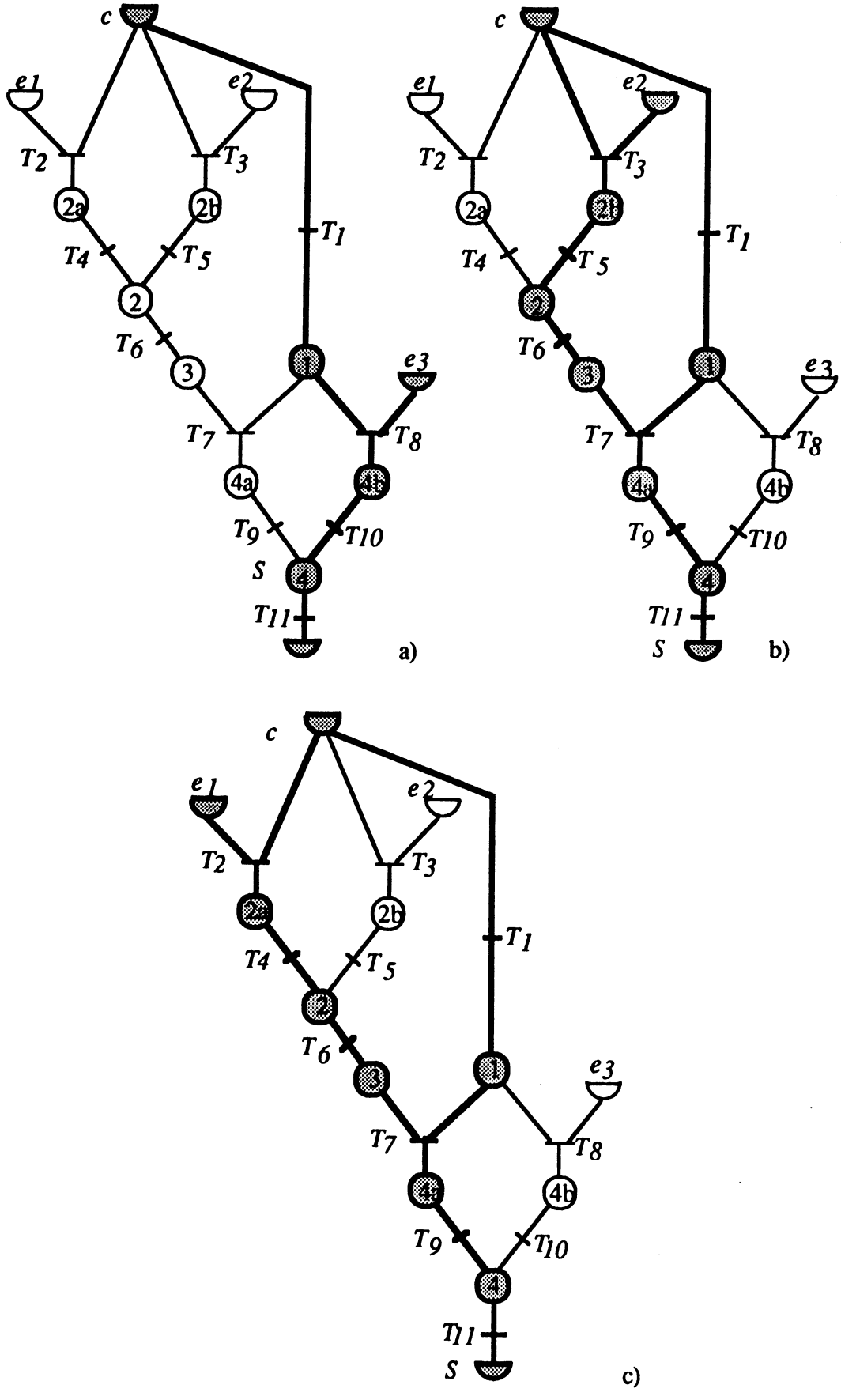


Figure 23 : Les écoulements obtenus pour l'exemple de la figure (II-22, b).

III - EXTENSION DU GRAPHE SATAN : GRAPHE INTERPRETE

Le graphe de testabilité SATAN obtenu en utilisant les règles de traduction précédentes ne spécifie aucune information concernant le temps. Pourtant, les caractéristiques temporelles d'un circuit jouent un rôle considérable dans la génération de son test. Il est donc dommage que ce graphe de testabilité ignore ces caractéristiques temporelles.

Dans ce paragraphe, on introduira la prise en compte du temps spécifié par l'utilisation de l'opérateur de retard "pre" de LUSTRE. Cet opérateur permet de retarder les variables d'un instant, et donc d'obtenir un graphe de testabilité temporisé où on associe à chaque place ou transition son instant d'activation. Ceci permet de générer les séquences de test nécessaire pour le circuit étudié, séquences constituées par les variables de contrôle ainsi que les données nécessaires à chaque instant.

De plus, on étudiera les conditions d'activation d'un écoulement (un chemin d'information à travers le système), les contraintes sur les lignes de contrôle et la propagation de ces contraintes dans l'écoulement.

III. 1 - Prise en compte du temps

La prise en compte du temps est réalisée par une traduction plus précise de l'opérateur "pre" de LUSTRE que celle qu'on a vu auparavant, par la règle suivante :

III. 1. 1 - Quatrième règle de traduction

L'opérateur "pre" de LUSTRE appliqué à une variable va permettre de distinguer deux instants, l'un correspondant à la valeur courante de la variable, l'autre correspondant à la valeur précédente. Cette règle consiste à transformer le module correspondant à cet opérateur en deux carrés séparés,

l'instant t est associé au premier et auquel correspond la valeur courante, l'instant t-1 est associé au deuxième et auquel correspond la valeur précédente de la variable considérée.

Exemple :

La fonction "and" entre la valeur actuelle de ck et l'inverse de sa valeur précédente est modélisée comme indiqué, figure II-24. Cette fonction permet de détecter le front montant de la variable ck.

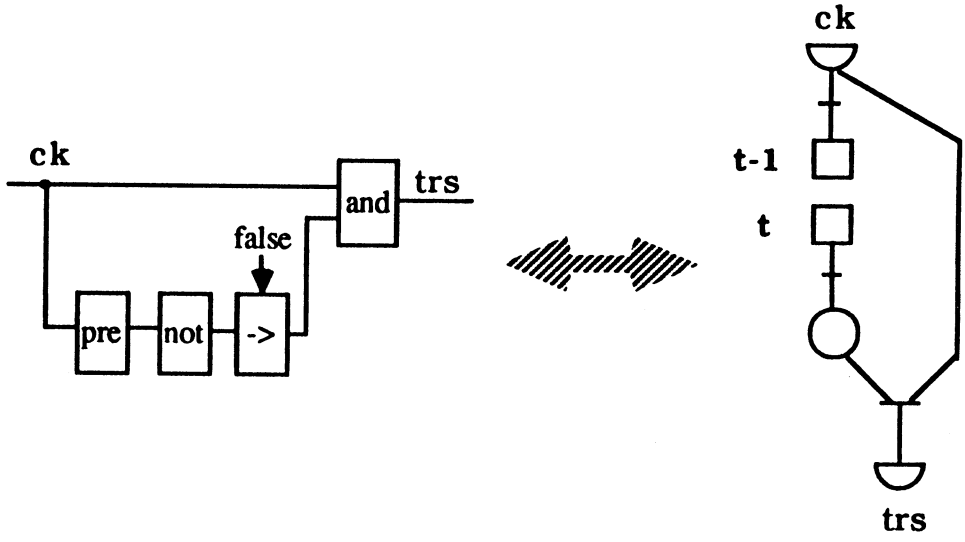


Figure II-24 : Réseau d'opérateurs et graphe SATAN de cet exemple.

III. 1. 2 - Temporisation des écoulements obtenus

La traduction de l'opérateur de retard "pre" de LUSTRE en graphe de testabilité SATAN en considérant la quatrième régle de traduction permet donc de temporiser les écoulements obtenus.

En partant à partir du puits auquel est associé l'instant t pour un écoulement, on associe à chaque place ou transition son instant. L'opérateur de retard "pre" de LUSTRE, qui est représenté par deux carrés (au premier est associé l'instant t et au deuxième est associé l'instant t-1), permet de décrémenter l'instant d'activation. L'instant t-1, alors, est associé à toutes les

places et les transitions qui se situent en amont de ce double carré jusqu'à l'arrivée à un autre double carré qui représente un autre opérateur "pre".

A chaque place d'un écoulement de l'ensemble d'écoulements, on associe l'instant d'activation I ou éventuellement l'ensemble d'instants d'activation $\{I_1, I_2, \ldots, I_n\}$ pour une place à laquelle des instants différents sont associés. L'interprétation de ce genre de places à instants d'activation multiple est que plusieurs valeurs de la variable correspondante sont nécessaires pour activer cet écoulement.

Pour un écoulement dans le graphe de testabilité temporisé, on associe à chaque transition aussi l'instant d'activation (ou éventuellement l'ensemble d'instants pour une transition à laquelle des instants différents sont associés). Les transitions seront, alors, définies par un couple : $\delta = (T, I)$.

où T est la transition définie par : / liste des places prédécesseurs / liste des places successeurs / ;

I est son instant (ou éventuellement l'ensemble d'instants associés).

On obtient alors un ensemble des écoulements ou chemins d'information (SATAN) temporisés qui feront apparaître les différents transferts de données effectués à chaque instant. Ces écoulements permettent de préciser les instants où les entrées de contrôle ou de données sont prises en considération. A noter qu'une boucle dans un écoulement dans le graphe temporisé obtenu n'est parcouru qu'une fois.

Exemple :

La traduction du même réseau d'opérateurs, figure II-18 en introduisant la quatrième régle de traduction va donner un graphe de testabilité montré dans la figure II-25 :

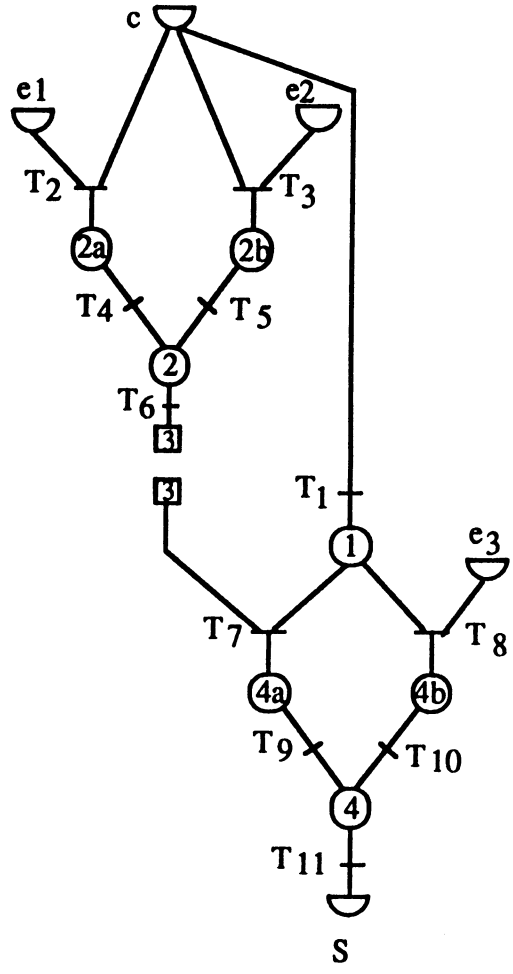
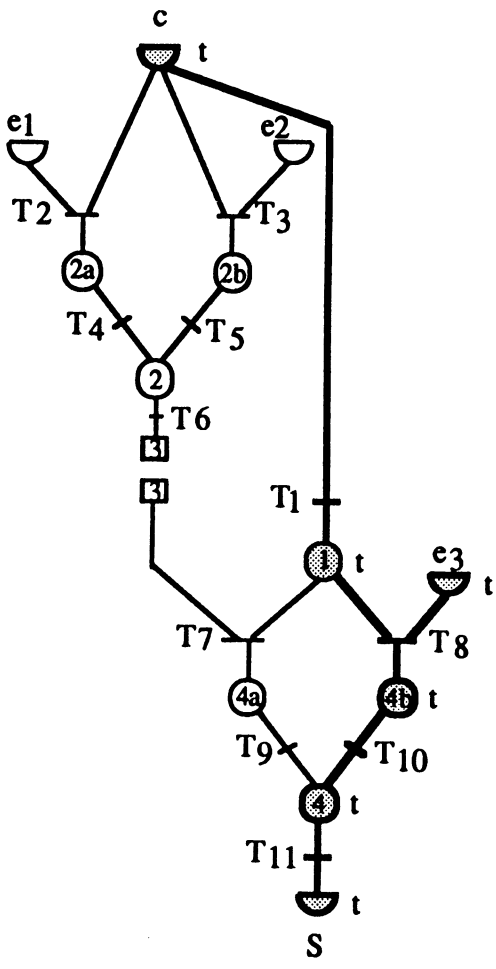
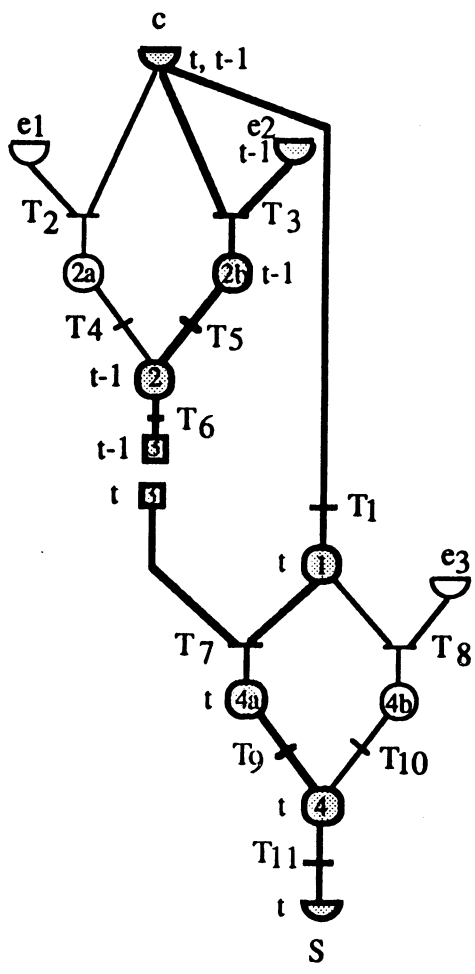


Figure II-25 : Graphe de testabilité SATAN en introduisant la quatrième régle de traduction.

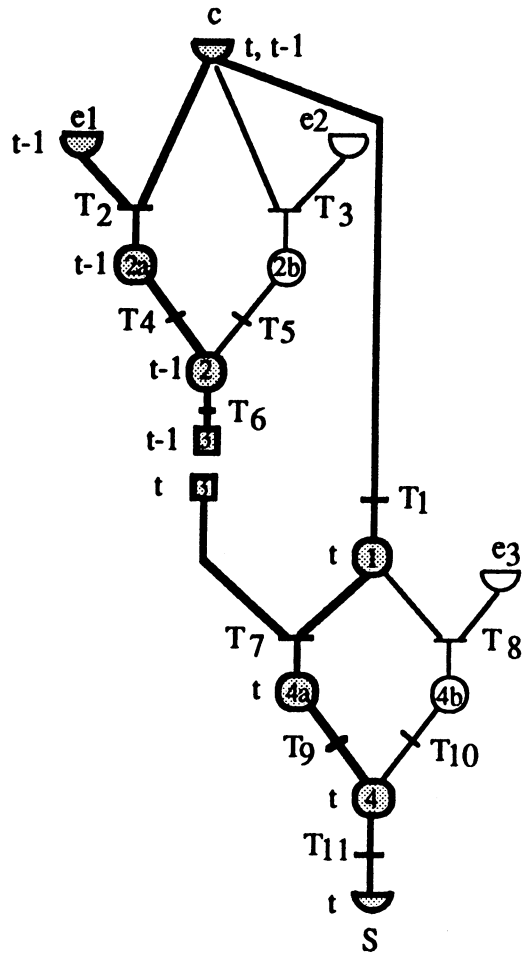
Les trois écoulements obtenus précédemment, figure II-20, peuvent être temporisés en distinguant deux instants distincts t et t-1 pour le deuxième et le troisième écoulement. Ces écoulements sont montrés dans la figure II-26.



a) écoulement 1 :
l'activation nécessite des valeurs de c et e_3 à l'instant t.



b) écoulement 2 :
l'activation nécessite des valeurs de c à l'instant t et t-1, et e_2 à l'instant t-1.



c)

écoulement 3 : l'activation nécessite des valeurs de c à l'instant t et t-1, et e_1 à l'instant t-1.

Figure II-26 : Les écoulements temporisés.

A l'aide de la temporisation des écoulements, on précise à quel instant les valeurs des entrées primaires de commandes et de données doivent être positionnées pour activer l'écoulement souhaité.

De plus, le graphe de testabilité obtenu va permettre d'associer les caractéristiques soit de contrôle, soit de donnée de certaines fonctions. La propagation de la valeur de contrôle vers les sources va permettre de distinguer les places soumises éventuellement à des contraintes d'activation (contrôle) et celles non contraintes qui contiendront les données de test proprement dites. Ainsi, les sources (les entrées primaires) seront de deux types : les sources de contrôle sur lesquelles l'action sera limitée car elles sont

liées à l'activation d'un écoulement et les sources de données de test effectives.

La séquence des entrées primaires commandes et données nécessaires, pour chaque écoulement de la carte correspondant aux différents instants, constitue ce qu'on va l'appeler séquence de test (stimuli).

Pour un écoulement donné d'un circuit, la génération de programme de test nécessite la définition, d'une part, les variables de type contrôle permettant d'activer sa fonction et les données associées et, d'autre part, les instants d'activation des différentes entités fonctionnelles.

III. 2. - Détection des conflits

L'activation d'une fonction dépend généralement de valeurs prises par des signaux de commande qui sont les conditions d'activation de cette fonction. Ces conditions d'activation sont décrites au niveau de la transition concernée par un ou plusieurs arcs pour lesquels il faut déterminer la valeur qui autorise l'exécution de cette fonction.

Une activation fonctionnelle (ou un écoulement) d'un circuit dépend, d'une part, de la définition des conditions d'activation par la détermination des valeurs des variables de type contrôle autorisant l'exécution de cet écoulement ainsi que les données associées et, d'autre part, des différents instants des différentes places qui sont activées par cet écoulement. Il s'agit donc, au niveau d'un écoulement, de spécifier les conditions d'activation de chaque place ainsi que l'instant (ou éventuellement l'ensemble d'instants associés).

Exemple :

Reprenons la traduction de la primitive de multiplexage de LUSTRE (Cf. figure 6), la valeur de c permet de sélectionner un des deux chemins distincts, l'un pour les données sélectionnées par la valeur "vrai" de cette variable, l'autre pour les données sélectionnées par la valeur "faux" de la même

variable. Donc, l'activation de ce graphe SATAN est spécifiée par la valeur de c, figure II-27.

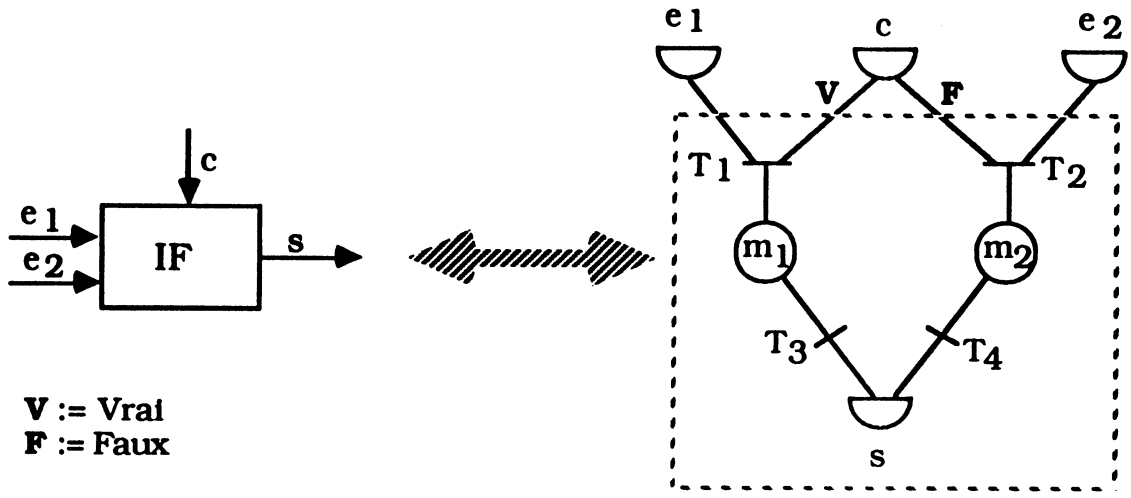


Figure II-27 : Condition d'activation pour la primitive if-then-else.

Un conflit, d'une façon générale, est détecté si une place doit générer deux valeurs différentes pour un même signal à un même instant, figure II-28.

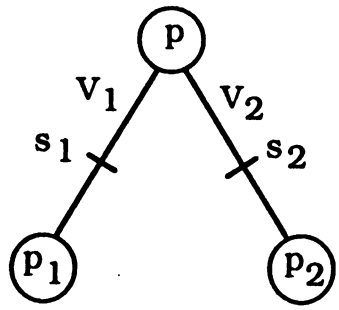
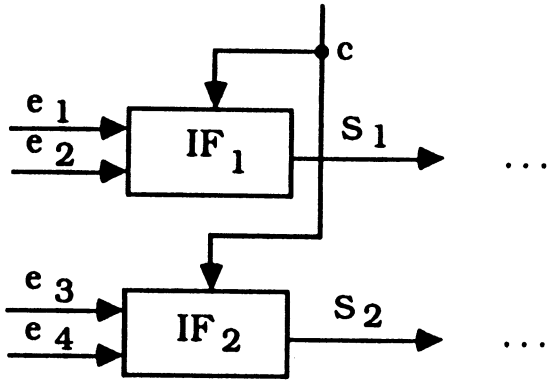


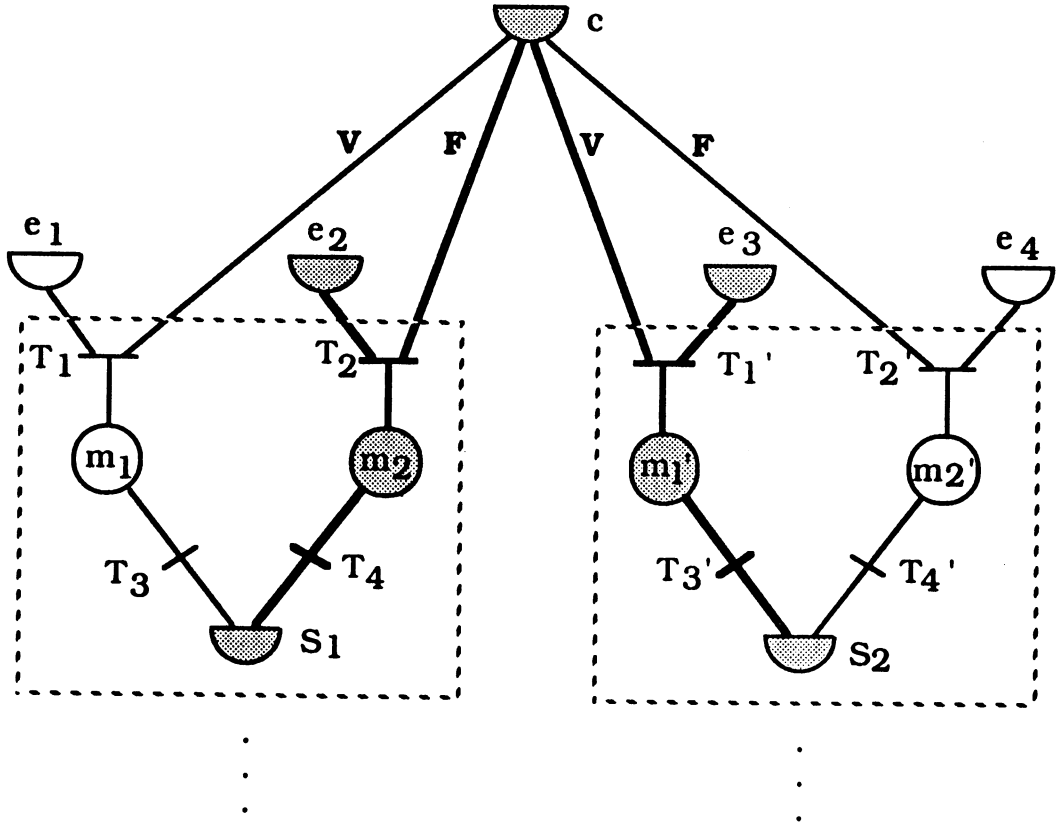
Figure II-28 : Détection des conflits.

La détection d'un conflit va être faite à partir des sorties de l'écoulement vers ses entrées. Un conflit pour un écoulement peut être dû à la traduction de l'interconnexion de deux primitives de multiplexage "if-then-else" par la même ligne de contrôle logique, figure II-29, qui se fait par la troisième règle de traduction où l'entrée c est prise en compte pour cet écoulement par deux chemins dont un nécessite une valeur "vrai" et l'autre une valeur "faux" de la même variable.

De tels écoulements doivent être supprimés de la liste des écoulements obtenue.



a)



b)

Figure II-29 : a) Réseau d'opérateurs ; b) Conflit Détecté.

III. 3 - Logiciel de traduction

Un logiciel de traduction des différentes primitives du langage LUSTRE en utilisant les différentes régles de traduction et les régles d'optimisation définies auparavant, a été réalisé en PASCAL sur μVAX VMS.

Ce logiciel prend comme entrée le réseau d'opérateurs obtenu après la compilation d'une source LUSTRE dans un format interne, traduit les différentes primitives du langage avec d'éventuelles optimisations, figure II-30, et donne, comme résultat, le modèle de testabilité temporisé de cette source modélisé en format SATAN (exprimé pour chaque transition par : liste des places prédécesseurs / liste des places successeurs).

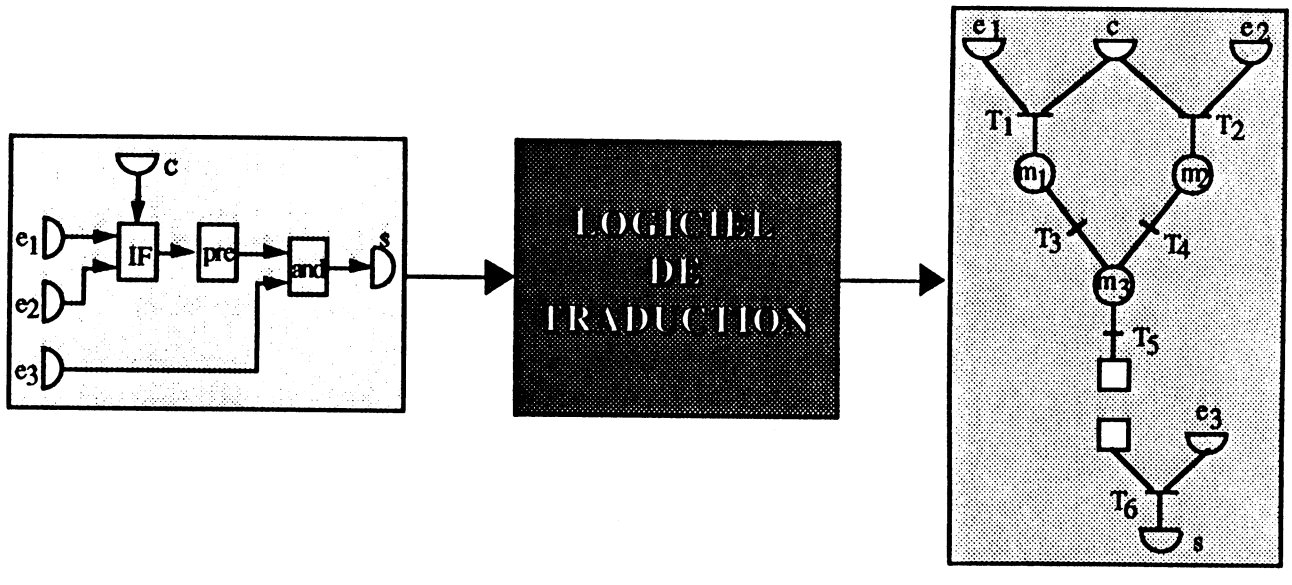


Figure II-30 : Logiciel de traduction.

III. 4 - Traitement d'un exemple

L'exemple suivant illustre les différentes étapes nécessaires pour obtenir les écoulements d'un circuit et la temporisation de ces écoulements.

Considérons un circuit qui comporte deux registres simples (REG1 et REG2) ayant un signal de remise à zéro RESET et un signal de chargement CH, deux multiplexeurs (MUX1 et MUX2) et une fonction F, figure II-31.

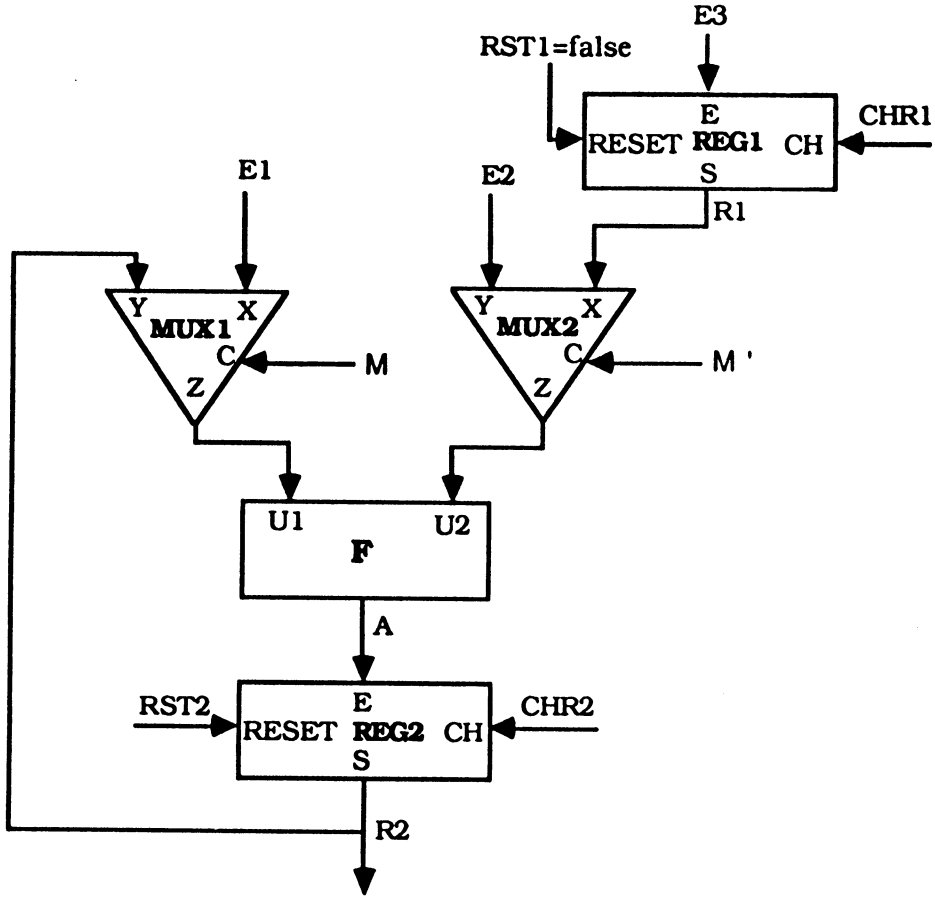


Figure II-31 : Le circuit à tester.

La description LUSTRE de ce circuit est la suivant :

```
node REG(RESET, CH : bool; E : int) returns(S : int);
let
S =  if RESET then 0
     else 0 -> pre( if (CH and not RESET)  then E
                                           else S);
tel.
```

```
node MUX(C : bool; X, Y : int) returns(Z : int);
let
Z = if C  then X
          else Y;
tel.

node F(U1, U2 : int) returns(A : int);
let

...

tel.

node circuit(M, M', CHR1, CHR2, RST2 : bool; E1, E2, E3 : int)
                returns(R2 : int);

var R1, U1, U2, A : int;
let
R1= REG(false, CHR1, E3);
U1= MUX(M, E1, R2);
U2= MUX(M', R1, E2);
A= F(U1, U2);
R2= REG(RST2, CHR2, A);
tel.
```

Le réseau d'opérateurs associé à ce circuit est montré dans la figure II-32. Les différentes primitives LUSTRE sont numérotées pour pouvoir les traduire en graphe de testabilité SATAN associé.

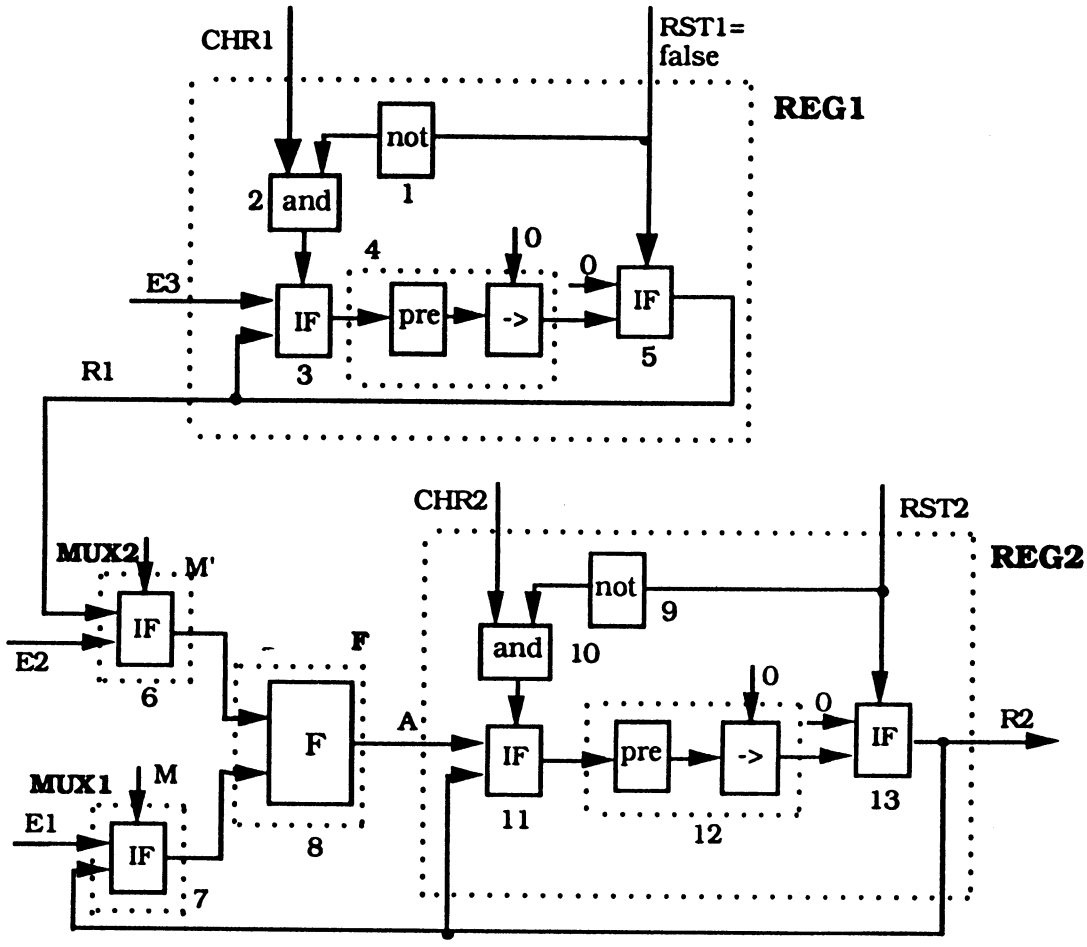


Figure II-32 : Réseau d'opérateurs obtenu.

Remarque : L'ensemble de l'opérateur "pre" et l'opérateur "->" est numéroté par un seul numéro.

Une traduction en graphe de testabilité SATAN est effectuée en utilisant les règles de traduction et les règles d'optimisation. Chaque primitive de type "if-then-else" est traduite par l'attribution de deux chemins d'information. Chacun de ces deux chemins d'information est repéré suivant la valeur de contrôle (V= vrai; F= faux). Le graphe SATAN obtenu pour ce circuit est montré dans la figure II-33.

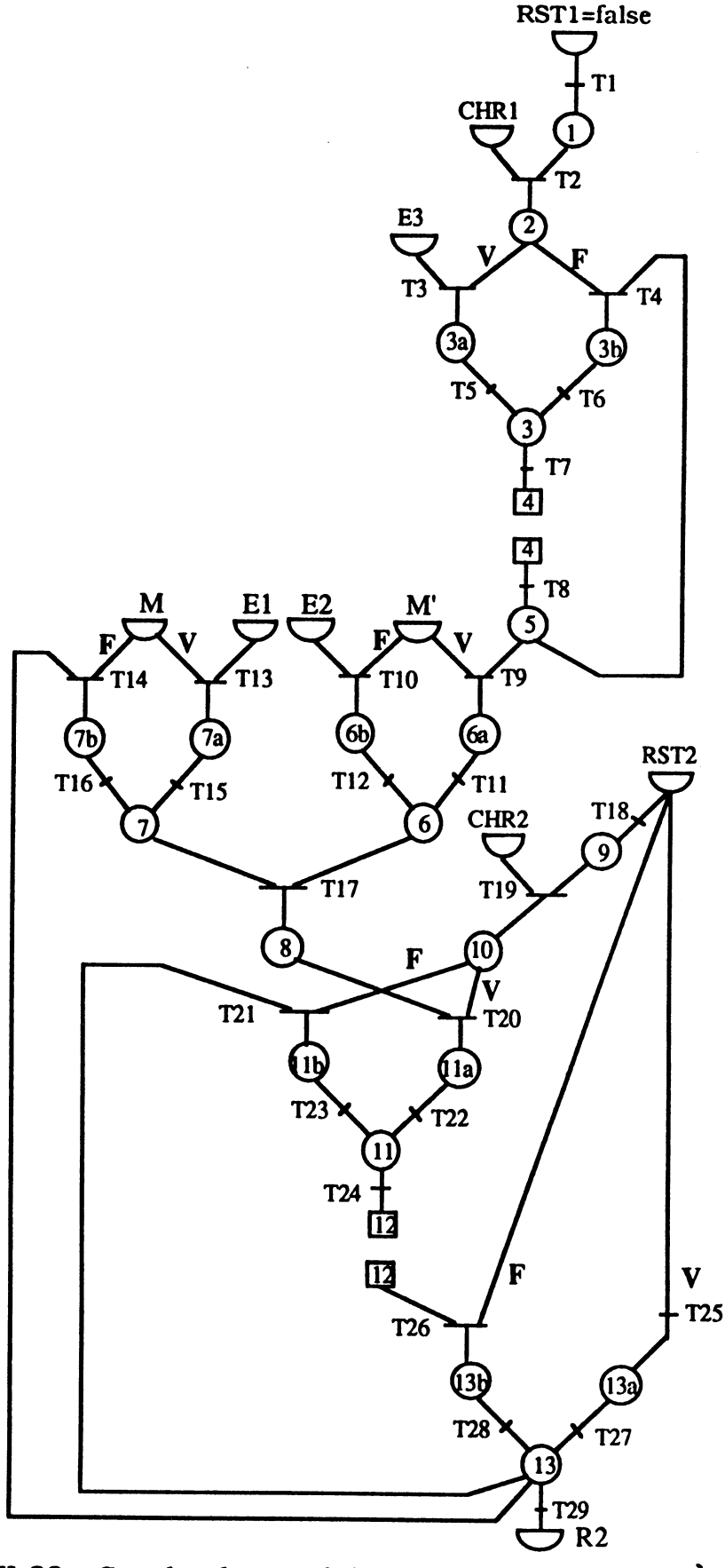


Figure II-33 : Graphe de testabilité en introduisant la 4ème règle.

Huit écoulements distincts sont déterminés. Une analyse sur ces écoulements doit être effectuée, d'abord, pour annuler les écoulements incohérents (les écoulements qui mènent à des conflits), ensuite, pour déterminer les entrées primaires permettant d'activer l'écoulement souhaité. Par exemple pour le troisième écoulement, figure II-34, la valeur de RST(t-1) à l'instant t-1 et la valeur de CHR2(t-1) à l'instant t-1 doivent être sélectionnées telles que la condition d'activation de la transition T20 est satisfaite : m10(t-1)= vrai. Aussi pour le septième écoulement, figure II-35, CHR1(t-2) doit être sélectionnée de telle sorte que m2(t-2)=vrai en considérant en compte que RST1(t-2)= faux.

Pour cet exemple, il n'y a pas d'écoulements incohérents à supprimer parce que le réseau d'opérateurs ne possède pas deux primitives de multiplexage connectées par la même variable de contrôle et simultanées (qui peuvent entraîner ce genre d'écoulements incohérents).

Les figures II-34 et II-35 montrent le troisième et le septième écoulement dans lequel on distingue trois instants (t, t-1, t-2). Les séquences d'entrée de contrôle ou de donnée nécessaires pour chaque écoulement sont les suivantes :

Premier écoulement :

Modules : 13a, 13;
Transitions : T25, T27, T29;
Séquence d'entrée : à l'instant t : RST2(t)= vrai.

Deuxième écoulement :

Modules : 9, 10, 11b, 11, 12, 13b, 13 ;
Transitions : T18, T19, T21, T23, T24, T26, T28, T29;
Séquences d'entrée :

à l'instant t : RST2(t)= faux ;
à l'instant t-1 : { RST2(t-1), CHR2(t-1) / m10(t-1)= faux };

les valeurs de RST2(t-1) et CHR2(t-1) sont nécessaires de telle sorte que m10(t-1)= faux ;

Troisième écoulement :

Modules : 6b, 6, 7a, 7, 8, 9, 10, 11a, 11, 12, 13b, 13 ;
Transitions : T10, T12, T13, T15, T17, T18, T19, T20, T22, T24, T26, T28, T29;
Séquences d'entrée :

à l'instant t : RST2(t)= faux ;
à l'instant t-1 : { RST2(t-1), CHR2(t-1) / m10(t-1)= vrai }; M(t-1)= vrai ; E1(t-1) ; M'(t-1)= faux ; E2(t-1).

Quatrième écoulement :

Modules : 6b, 6, 7b, 7, 8, 9, 10, 11a, 11, 12, 13b, 13 ;
Transitions : T10, T12, T14, T16, T17, T18, T19, T20, T22, T24, T26, T28, T29;
Séquences d'entrée :

à l'instant t : RST2(t)= faux ;
à l'instant t-1 : { RST2(t-1), CHR2(t-1) / m10(t-1)= vrai }; M(t-1)= faux ; M'(t-1)= faux ; E2(t-1).

Cinquième écoulement :

Modules : 1, 2, 3a, 3, 4, 5, 6a, 6, 7a, 7, 8, 9, 10, 11a, 11, 12, 13b, 13 ;
Transitions : T1, T2, T3, T5, T7, T8, T9, T11, T13, T15, T17, T18, T19, T20, T22, T24, T26, T28, T29;
Séquences d'entrée :

à l'instant t : RST2(t)= faux ;
à l'instant t-1 : { RST2(t-1), CHR2(t-1) / m10(t-1)= vrai }; M(t-1)= vrai ; E1(t-1) ; M'(t-1)= vrai ;
à l'instant t-2 : { CHR1(t-2) / m2(t-2)= vrai et RST1(t-2)= faux }; E3(t-2).

Sixième écoulement :

Modules : 1, 2, 3b, 3, 4, 5, 6a, 6, 7b, 7, 8, 9, 10, 11a, 11, 12, 13b, 13 ;
Transitions : T1, T2, T4, T6, T7, T8, T9, T11, T13, T15, T17, T18, T19, T20, T22, T24, T26, T28, T29;
Séquences d'entrée :

à l'instant t	:	RST2(t)= faux ;
à l'instant t-1	:	{ RST2(t-1), CHR2(t-1) / m10(t-1)= vrai }; M(t-1)= vrai ; E1(t-1) ; M'(t-1)= vrai ;
à l'instant t-2	:	{ CHR1(t-2) / m2(t-2)= faux et RST1(t-2)= faux};

Septième écoulement :

Modules : 1, 2, 3a, 3, 4, 5, 6a, 6, 7b, 7, 8, 9, 10, 11a, 11, 12, 13b, 13 ;
Transitions : T1, T2, T3, T5, T7, T8, T9, T11, T14, T16, T17, T18, T19, T20, T22, T24, T26, T28, T29;
Séquences d'entrée :

à l'instant t	:	RST2(t)= faux ;
à l'instant t-1	:	{ RST2(t-1), CHR2(t-1) / m10(t-1)= vrai }; M(t-1)= faux ; M'(t-1)= vrai ;
à l'instant t-2	:	{ CHR1(t-2) / m2(t-2)= vrai et RST1(t-2)= faux }; E3(t-2).

Huitième écoulement :

Modules : 1, 2, 3b, 3, 4, 5, 6a, 6, 7b, 7, 8, 9, 10, 11a, 11, 12, 13b, 13 ;
Transitions : T1, T2, T4, T6, T7, T8, T9, T11, T14, T16, T17, T18, T19, T20, T22, T24, T26, T28, T29;
Séquences d'entrée :

à l'instant t	:	RST2(t)= faux ;
à l'instant t-1	:	{ RST2(t-1), CHR2(t-1) / m10(t-1)= vrai }; M(t-1)= faux ; M'(t-1)= vrai ;
à l'instant t-2	:	{ CHR1(t-2) / m2(t-2)= faux et RST1(t-2)= faux};

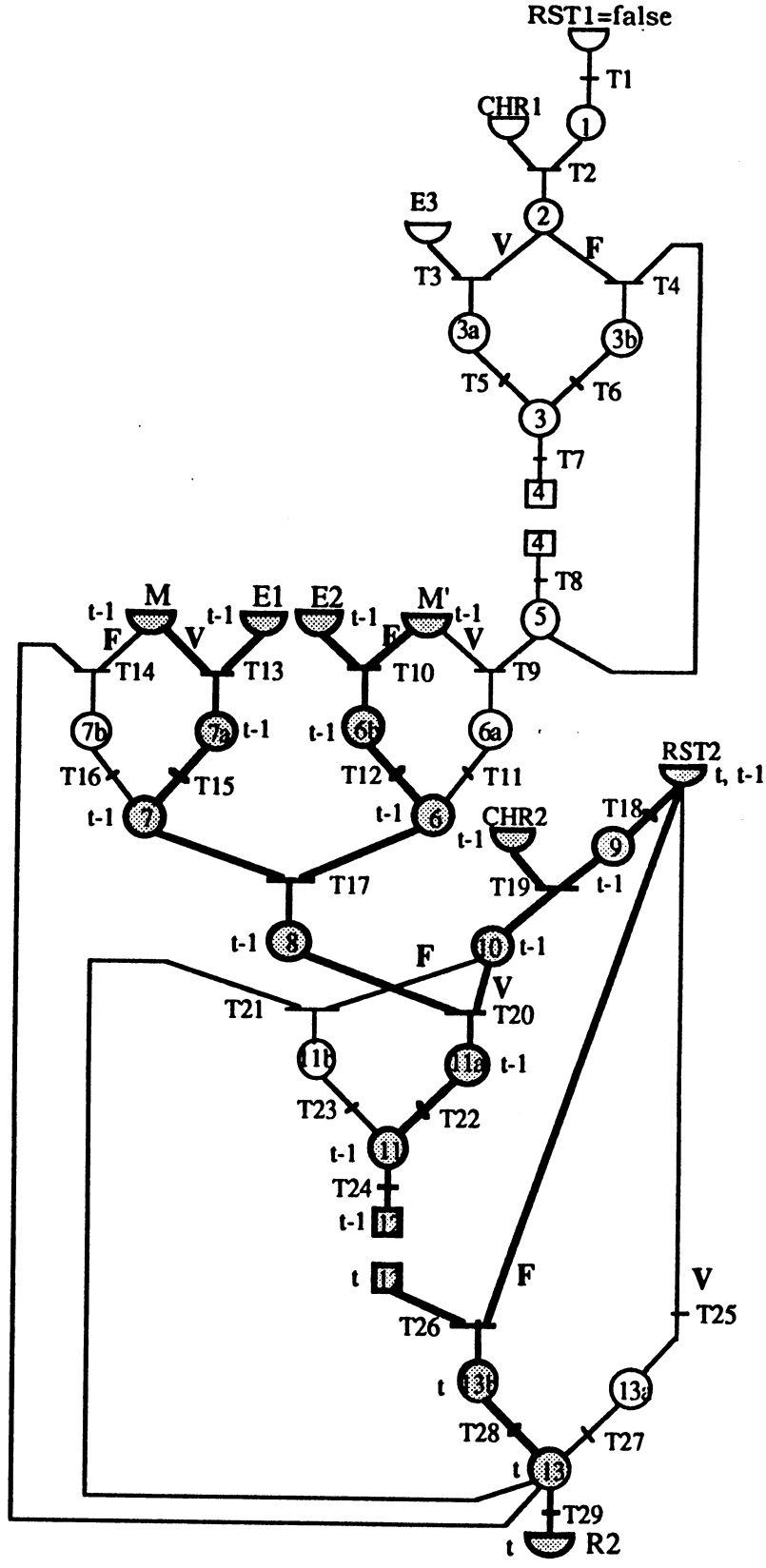


Figure II-34 : Troisième écoulement.

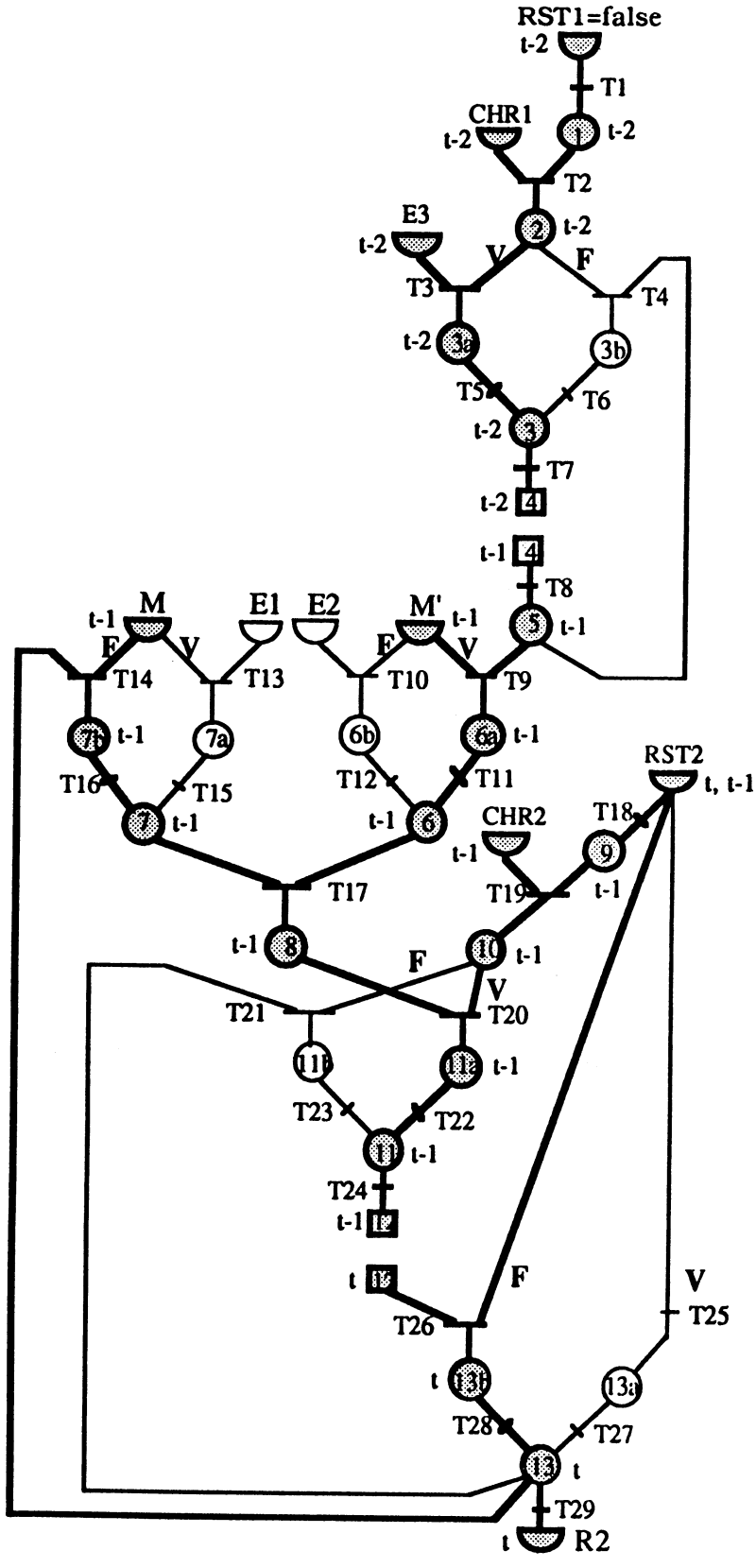


Figure II-35 : Septième écoulement.

III. 5 - Exploitation pour le test

Les écoulements temporisés d'un circuit obtenus en associant un instant d'activation à chaque transition ou place peuvent être utilisés pour l'analyse de testabilité de ce circuit et servent comme un outil pour la détermination des séquences de test. La détermination des séquences de test d'un circuit consiste à définir, d'une part, les variables de type contrôle permettant d'activer sa fonction et les données associées et, d'autre part, les instants d'activation des différentes entités fonctionnelles.

Donc, on peut récapituler les phases nécessaires parcourus pour l'établissement d'une séquence de test d'un circuit (Cf. figure II-36).

a) Détermination du réseau d'opérateurs à partir d'une description textuelle LUSTRE.

b) Traduction de ce réseau en graphe de testabilité SATAN par les règles de traduction et les règles d'optimisation.

c) Recherche d'écoulements du graphe de testabilité obtenu et temporisation des écoulements en associant à chaque place ou transition dans l'écoulement son instant (ou éventuellement l'ensemble d'instants).

d) Pour chaque écoulement, détermination des séquences d'entrée de contrôle ou de donnée de test permettant d'activer cet écoulement.

Les trois premières phases sont réalisées tandis que la dernière reste à faire.

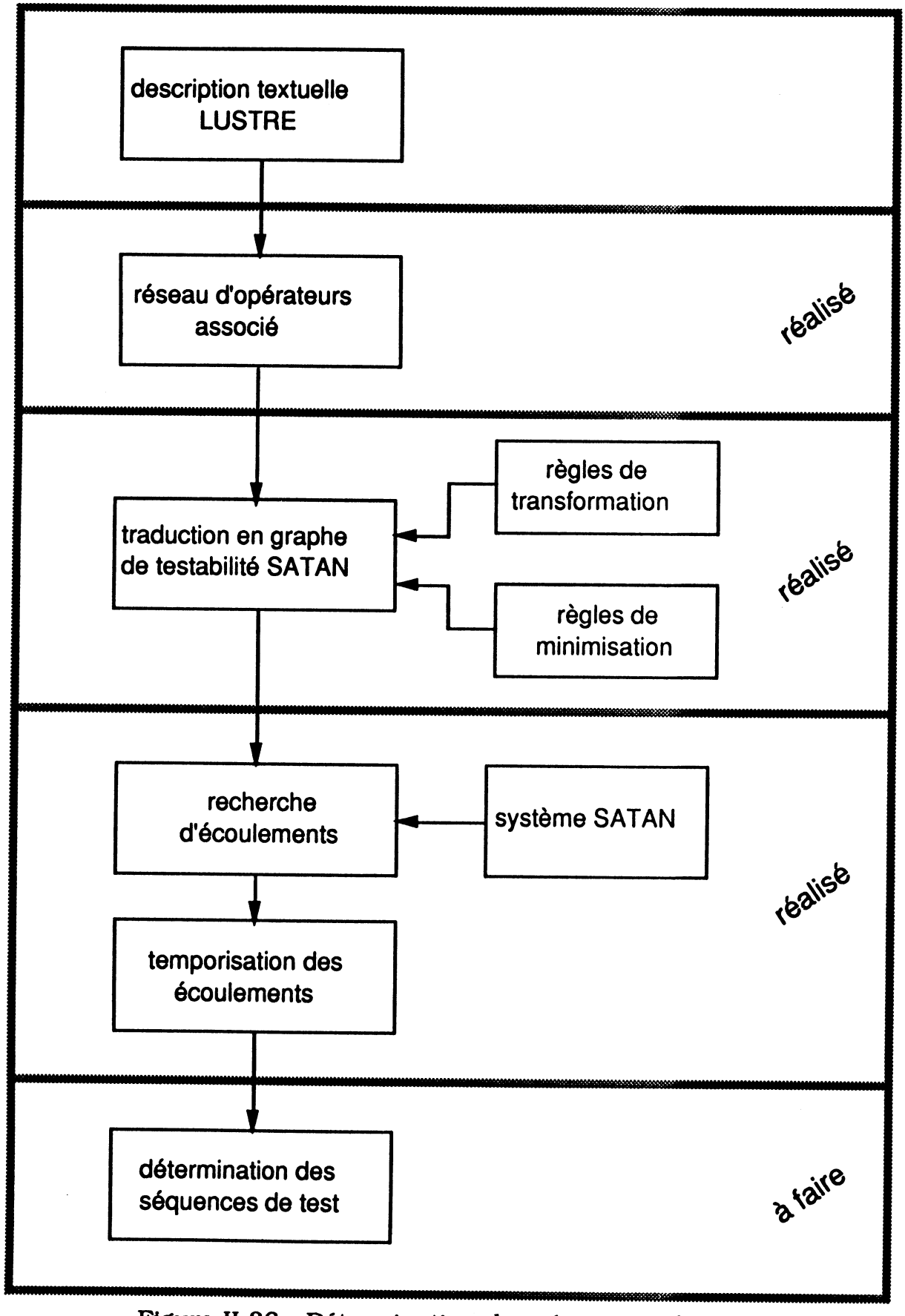


Figure II-36 : Détermination des séquences de test.

La génération de test est fondée sur des notions de couvertures fonctionnelles pour tous les écoulements possibles de la carte. Elle consiste à choisir un écoulement et à déterminer les séquences d'entrée de type contrôle nécessaires pour activer cet écoulement et les séquences d'entrée de type donnée qui constituent les données de test proprement dites. L'ordre de choix des écoulements est déterminé par quelques critères comme par exemple : la stratégie de test utilisée pour couvrir l'ensemble de ces écoulements, la nécessité d'initialiser certains modules qui sont nécessaires pour l'activation d'un écoulement et peuvent être positionnés par un autre écoulement. Alors, le dernier écoulement doit être activé auparavant. L'organigramme de détermination du programme de test est montré dans la figure II-37.

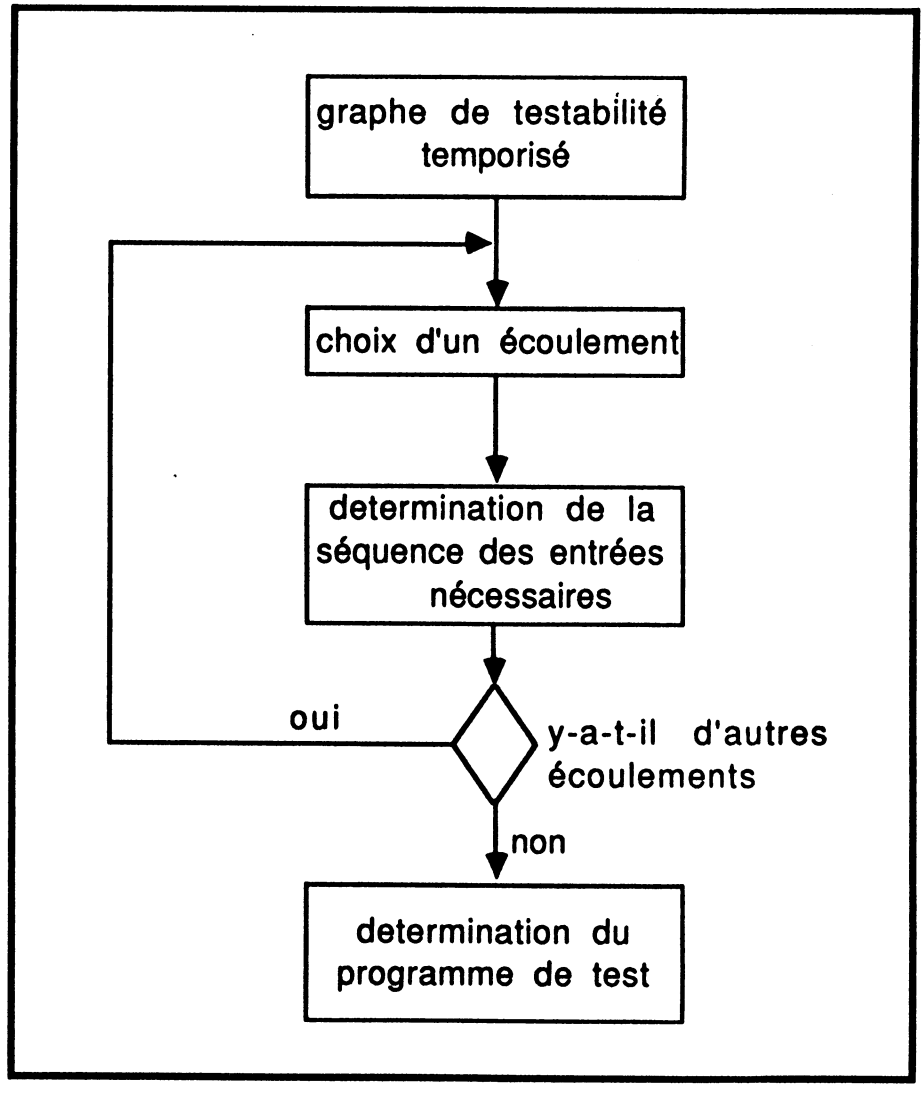


Figure II-37 : Organigramme de détermination du programme de test.

Une stratégie de test de type "start-small" (boule de neige) [ROB 85], peut être employée pour permettre l'écriture d'un programme de test performant qui sera composé de ces séquences d'entrée.

Exemple :

A partir des écoulements temporisés, qui sont montrés dans la figure II-26 pour l'exemple dont le réseau d'opérateurs est donné dans la figure (II-22, a), on définit les séquences d'entrée d'un écoulement.

Le premier écoulement, pour cet exemple, nécessite le signal c(t) à l'instant courant t tel que m1(t)= faux et la donnée $e_3(t)$. Le deuxième écoulement nécessite le signal c(t) à l'instant courant t tel que m1(t)= vrai, le signal c(t-1)= faux à l'instant t-1 et la donnée $e_2(t-1)$. Le troisième écoulement nécessite le signal c(t) à l'instant courant t tel que m1(t)= vrai, le signal c(t-1)= vrai à l'instant t-1 et la donnée $e_1(t-1)$.

Le test de ce circuit consiste à activer successivement ces écoulements en prenant en compte les valeurs des séquences d'entrée de contrôle nécessaires et en choisissant des valeurs des séquences d'entrée de donnée de test adéquates.

Exemple

La détermination des séquences d'entrée de test, pour le circuit de la figure II-31, consiste à déterminer les séquences d'entrée nécessaires pour activer l'écoulement choisi. L'ordre de choix des écoulements est déterminé suivant la stratégie de test utilisée (de type "start-small") et la nécessité que certains modules doivent être initialisés précédemment.

Par exemple, le quatrième écoulement utilise la valeur de la place m13 qui doit être initialisée précédemment pour cela le test de cet écoulement doit être effectué sans doute après le test d'un écoulement permettant d'initialiser ce module, par exemple : le troisième écoulement. Ce jugement est aussi valable pour l'ensemble des écoulements : $6^{ème}$ et $8^{ème}$ qui

nécessitent l'initialisation de la place m5 qui doit être effectuée par des écoulements comme le 5ème ou le 7ème.

Le programme de test de ce circuit sera la concaténation des séquences d'entrée nécessaires pour activer les écoulements obtenus.

Conclusion

Dans ce chapitre, nous avons proposé une méthode d'évaluation de la testabilité d'un circuit et de détermination de programme de test fondée sur un modèle du circuit, décrit par un langage de haut niveau, déclaratif : LUSTRE à partir des différentes primitives de ce langage.

Cette méthode est basée sur la traduction des différentes primitives de LUSTRE compilées sous forme d'un réseau d'opérateurs en graphe de testabilité SATAN. Le fait que le nombre de primitives soit limité simplifie la tache de traduction et permet d'introduire un ensemble de régles de traduction. Un ensemble de règles spécifiques d'optimisation est également introduit et on obtient un graphe de testabilité.

Les écoulements du graphe de testabilité obtenu sont déterminés par le système SATAN. L'introduction de la prise en compte du temps a permis de temporiser ces écoulements où on associe à chaque place ou transition son ou ses instants d'activation. Cela permet de préciser les instants où les entrées de contrôle ou de donnée doivent être positionnées pour activer l'écoulement souhaité. La détermination des séquences de test consiste à définir, d'une part, les variables de type contrôle permettant d'activer la fonction et les données associées et, d'autre part, les instants d'activation des différentes entités fonctionnelles.

Un logiciel de traduction des différentes régles introduites est réalisé. Ce logiciel permet de traduire des différentes primitives du langage LUSTRE en graphe de testabilité temporisé SATAN. Ce logiciel a été appliqué à un ensemble d'exemples (comportant l'équivalent de 500 à 1000 portes) avec succès.

CHAPITRE 3

TEST PAR IDENTIFICATION DE LA PARTIE CONTROLE A TRAVERS LA PARTIE OPERATIVE D'UN CIRCUIT COMPLEXE

INTRODUCTION

Le test d'automate (machine séquentielle) a été très étudié dans la littérature. On peut distinguer deux approches : test par distinction et test par identification.

Le ***test par distinction*** a pour but de distinguer la machine juste de l'ensemble des machines fausses qui sont représentées soient par un tableau d'états, soient par un graphe d'états. Cette approche est basée sur la détermination de l'ensemble des automates faux à partir d'hypothèses d'erreurs. Elle a été proposée par Poage en [POA 63].

Le ***test par identification*** des automates ("checking experiments") a pour but de vérifier que l'automate fonctionne correctement, c'est à dire d'identifier son fonctionnement à ses spécifications. Cette méthode de test concerne des automates réduits, déterministes, à graphe fortement connexe. L'hypothèse de base est que les erreurs dans l'automate à tester n'augmentent pas le nombre d'états [KOH 78].

Nous proposons une extension des méthodes classiques de test par identification pour tester les automates générés à partir d'une description d'un circuit par le langage LUSTRE.

Notre but est de définir le test de la P.C. à travers la P.O. par des méthodes d'identification. Ce test est établi en utilisant des séquences élémentaires prédéfinies.

Après avoir présenté brièvement le test par identification pour des machines séquentielles simples classiques (automates), on présentera le test d'un automate généré à partir d'une description de haut niveau LUSTRE pour les circuits constitués d'une *partie contrôle* "P.C." et d'une *partie opérative* "P.O."

I - TEST PAR IDENTIFICATION

Le but du test par identification est d'effectuer un test exhaustif de la partie combinatoire de l'automate, figure III-1.

Si il y a n entrées et m variables d'états, il y a 2^{n+m} combinaisons de valeurs de test. Notons que, si le test exhaustif du circuit combinatoire est réalisé, le test exhaustif de l'ensemble des éléments de mémorisation est effectué.

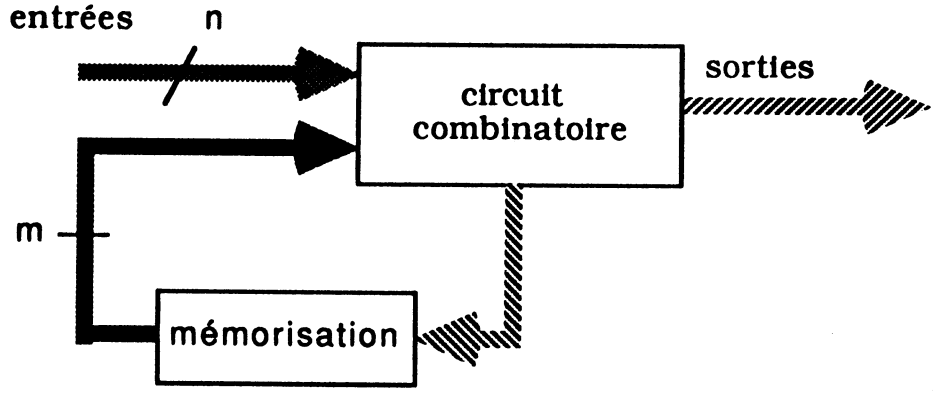


Figure III-1 : Automate.

Le test comporte trois parties :

- positionnement de l'automate dans un état connu.
- identification de tous les états par observation des vecteurs de sortie.
- vérification de toutes les transitions, par observation du vecteur de sortie durant la transition (automate de Mealy) et identification de l'état d'arrivée.

La séquence de test est constituée de deux parties [KOH 78] :

a) Séquence de test adaptative : Cette partie du test vise à transférer l'automate dans un état prédéterminé. Elle est adaptative parce que les séquences d'entrées suivantes sont déterminées en fonction des vecteurs de sortie précédents.

b) Séquence de test prédéterminée : Cette partie a pour but de vérifier tous les états et toutes les transitions possibles de l'automate. Le résultat (automate correct/incorrect) est déterminé par examen de la séquence de sortie.

Avant de rappeler les définitions des séquences élémentaires nécessaires pour les méthodes classiques de test par identification d'automates, rappelons qu'un automate à n états (automate d'états fini) est défini par un quintuplet $M = (X, Z, Q, f, g)$.

où X : est un ensemble fini de valeur des entrées.
Z : est un ensemble fini de valeur des sorties.
Q : est l'ensemble des n états.
f : $Q * X \rightarrow Q$: est la fonction de transition qui, à l'état présent et à une valeur d'entrée, fait correspondre un état suivant.
g : $Q * X \rightarrow Z$: est la fonction de sortie, (automate de Mealy), qui fait correspondre une valeur de sortie à un couple : (état, entrée).

I. 1 - Séquence de synchronisation

Une séquence de synchronisation est une séquence d'entrée telle que, quel que soit l'état de départ, l'application de cette séquence positionne l'automate dans le même état connu. Un automate a une séquence de synchronisation $X= X^1, X^2, \ldots, X^m$ de longueur m si :

$$\forall q_j , \exists X / f(q_j, X) = q_k.$$

Où q_j est l'état de départ, q_k est l'état d'arrivée.

On appelle ***liste d'états*** associée à une séquence d'entrée X, l'ensemble des états de la machine M auxquels elle peut arriver quand on applique la séquence d'entrée X, pour l'ensemble des états de départ possibles. Cet ensemble est représenté par une liste non ordonnée de ses membres.

Si V_1 est une liste d'états associée à la séquence d'entrée $X^1, X^2, \ldots, X^{m-1}$ et V_2 est est une liste d'états associée à la séquence $X^1, X^2, \ldots, X^{m-1}, X^m$, alors V_2 est appelée le ***X^m-successeur*** de V_1.

$$Q^m = V2 = (q_1^m, q_2^m, \ldots, q_n^m)$$

où $q_1^m, q_2^m, \ldots, q_n^m \in Q$.

Remarque :

Il s'agit d'une liste et non d'un ensemble parce qu'il est possible que certains états soient répétés.

La ***liste initiale d'états*** (incertitude initiale) pour une machine séquentielle à n états est l'ensemble de tous les états :

$$Q^0 = (q_1^0, q_2^0, \ldots, q_n^0)$$

L'état obtenu q_i^m en appliquant une séquence de vecteurs d'entrée $X^1, X^2, \ldots, X^m$ à partir de l'état initial q_i^0 est tel que :

$$q_i^1 = f(q_i^0, X^1)$$
$$q_i^2 = f(q_i^1, X^2) = f(f(q_i^0, X^1), X^2) = f(q_i^0, X^1 X^2)$$
$$\cdot$$
$$\cdot$$
$$q_i^m = f(q_i^0, X^1 X^2 \ldots X^m).$$

On appelle ***liste d'états identiques*** "LEI" une liste dans laquelle le même état est répété n fois (n étant le nombre d'états).

Q^k est une LEI ssi $\forall\, q_i^k, q_j^k \in Q^k \Rightarrow q_i^k = q_j^k$ avec $i \neq j$.

Une ***séquence de synchronisation*** X pour la machine M est telle que la liste d'états associée à cette séquence est constituée par une liste d'états identiques. Cette séquence est obtenue par un arbre des successeurs qui sera nommé l'arbre de synchronisation.

L'arbre de synchronisation :

L'***arbre de synchronisation*** est construit de la manière suivante : on commence par la liste initiale d'états pour la machine M qui représente la racine de l'arbre. Cette liste contient tous les états possibles de l'automate et est associée au nœud de niveau 0. A partir d'un nœud, l'ensemble des arcs est défini par toutes les combinaisons possibles de valeurs des entrées. A chaque niveau, on place toutes les listes d'états qui constituent les nœuds de l'arbre, associées aux séquences d'entrées correspondantes. Les niveaux sont numérotés 0, 1, . . . ,j, ... Une séquence de j arcs successifs, à partir de la liste initiale d'états qui se termine à un nœud au jème niveau, est référée comme un chemin de l'arbre et j est sa longueur. Un nœud dans le jème niveau devient terminal dans un des cas suivants :

1) une liste d'états associée à un nœud est déjà associée à un des nœuds d'un des niveaux précédents.
2) une liste d'états tous identiques est associée à un nœud du jème niveau.

La 1ère règle est appelée ***règle d'échec***, la 2ème est appelée ***règle de succès*** et permet de trouver une séquence de synchronisation. Cette séquence est donnée par la suite d'entrées associée à un chemin dans l'arbre qui mène de la liste initiale d'états à une liste d'états tous identiques.

Exemple :

Considérons la machine M_1 donnée figure III-2 ; l'arbre de synchronisation est montré figure III-3.

	X=0	X=1
A	B , 0	D , 0
B	A , 0	B , 0
C	D , 1	A , 0
D	D , 1	C , 0

état suivant, sortie.

Figure III-2 : Tableau d'états de la machine M_1.

La liste initiale d'états pour cette machine M_1 est (A,B,C,D). Le 0-successeur de la liste initiale d'états est (B,A,D,D) et le 1-successeur de la liste initiale d'états est (D,B,A,C). Cette dernière liste d'états est associée à un nœud terminal dans le niveau 1 parce que cette même liste est associée au nœud dans le niveau 0. Le 1-successeur de la liste d'états (B,A,D,D) est (B,D,C,C).

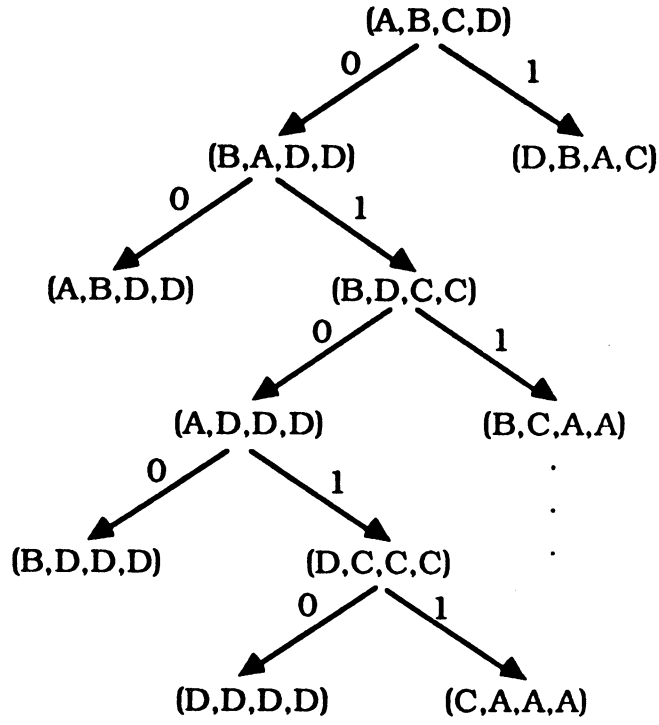


Figure III-3 : Arbre de synchronisation de la machine M_1.

On constate que la séquence : 01010 est une séquence de synchronisation : quel que soit l'état de départ, l'application de cette séquence positionne l'automate dans l'état D.

I. 2 - Séquence de positionnement

Une ***séquence de positionnement*** ("homing sequence") est une séquence d'entrée qui amène l'automate, quel que soit l'état de départ, à un état qui peut être déterminé par observation de la séquence de sortie.

Exemple :

Considérons la machine M_2, figure III-4.

	X=0	X=1
A	B , 1	D , 0
B	D , 1	C , 0
C	A , 0	B , 0
D	B , 1	A , 1

Figure III-4 : Tableau d'états de la machine M_2.

Cette machine ne possède pas de séquence de synchronisation mais elle possède des séquences de positionnement (par exemple 101). Les séquences de sortie obtenues en appliquant la séquence d'entrée 101 pour tous les états sont indiquées ci-dessous :

état initial	séquence de sortie	état final
A	010	C
B	000	D
C	011	A
D	110	C

Tableau III-1 : Séquences de sortie de M_2 pour la séquence d'entrée 101.

En observant la séquence de sortie produite, on peut déterminer l'état final obtenu par l'application de cette séquence d'entrée quel que soit l'état de départ.

Un ***vecteur d'incertitude*** (vecteur d'états) est un ensemble de listes d'états qui sont distinguées suivant la valeur de la séquence de sortie produite. Ce vecteur est associé à une séquence d'entrée $X^1, X^2, \ldots, X^m$ et constitué de plusieurs composants (listes d'états) séparés par des

parenthèses. Pour un vecteur d'incertitude associé à une séquence d'entrée, on crée autant de composants qu'il y a de séquences de sortie possibles.

Si V_1 est le vecteur d'incertitude obtenu par l'application de la séquence d'entrée $X^1, X^2, \ldots, X^{m-1}$ et V_2 est le vecteur d'incertitude obtenu par l'application de la séquence $X^1, X^2, \ldots, X^{m-1}, X^m$, alors V_2 est appelé le ***X^m-successeur*** de V_1. V_2 est construit en séparant chaque composant de V_1 suivant la sortie produite en réponse à l'entrée X^m appliquée. Deux états appartiennent au même composant de la liste si et seulement si ils produisent la même séquence de sortie.

Le ***vecteur initial d'états*** pour une machine séquentielle à n états est l'ensemble de tous ses états représentés par une liste $(q_1, q_2, \ldots, q_n)$. Le vecteur initial d'états donc est un vecteur ayant un seul composant. En effet, le vecteur initial d'états et la liste initiale d'états définie auparavant sont identiques.

Un vecteur d'incertitude, dont chacun des composants contient soit des singletons (un seul état), soit des états identiques répétés, est appelé ***vecteur d'incertitude homogène.***

Un vecteur d'incertitude dont chacun des composants contient un singleton est appelé ***vecteur d'incertitude trivial***. Un vecteur d'incertitude trivial est donc un vecteur d'incertitude homogène dont tous les composants se réduisent à des singletons.

La séquence de positionnement est obtenue par un arbre des successeurs qui sera nommé l'***arbre de positionnement***. Elle est obtenue par le chemin le plus court à partir du vecteur initial d'états jusqu'à l'arrivée à un vecteur d'incertitude homogène ou trivial.

L'arbre de positionnement :

L'arbre de positionnement est construit de la manière suivante : le vecteur initial d'états est associé à un nœud du niveau 0 de l'arbre. A chaque niveau, on place les vecteurs d'incertitude associés aux nœuds de l'arbre. Chaque entrée est associée à un arc de l'arbre. Les niveaux sont numérotés

0, 1, . . . , j, ... Cet arbre permet de montrer graphiquement le X^m-successeur du vecteur initial d'états pour toutes les séquences d'entrées X^m.

Les règles suivantes précisent les nœuds terminaux de l'arbre où le développement des nœuds s'arrête :

1) au nœud est associé un vecteur d'incertitude dont les composants sont non homogènes et qui est déjà associé à un des nœuds d'un des niveaux précédents.

2) à un nœud du $j^{ème}$ niveau est associé un vecteur d'incertitude trivial ou homogène.

La première règle est appelée règle d'échec, la deuxième règle est appelée règle de succès et permet de trouver une séquence de positionnement.

Exemple :

L'arbre de positionnement pour la machine précédente M_1, figure III-2, est montré dans la figure III-5.

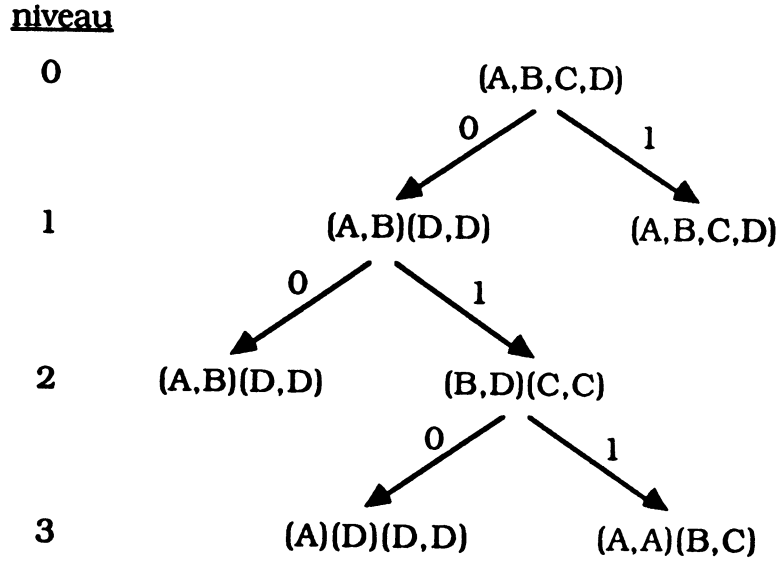


Figure III-5 : Arbre de positionnement de M_1.

Le vecteur initial d'états est (ABCD). Le 0-successeur du vecteur initial d'états est (AB)(DD). Le 1-successeur du vecteur initial d'états est le vecteur (ABCD). Ce vecteur est associé à un nœud terminal du niveau 1 puisqu'il est associé au nœud du niveau 0. Le 1-successeur du vecteur d'incertitude (AB)(DD) est (BD)(CC). Le nœud associé au (AB)(DD) du 2ème niveau est encore un terminal parce qu'il est identique au vecteur d'incertitude (AB)(DD) du niveau 1. Le vecteur d'incertitude (A)(D)(DD) est un vecteur d'incertitude homogène donc, la séquence de positionnement la plus courte est (010).

Le tableau suivant montre les différentes séquences de sortie obtenues par l'application de la séquence (010) à partir de tous les états :

état initial	séquence de sortie	état final
A	000	A
B	001	D
C	101	D
D	101	D

Tableau III-2 : Séquences de sortie de M_1 pour la séquence d'entrée 010.

I. 3 - Séquence de distinction

Une ***séquence de distinction*** est une séquence d'entrée qui produit, pour chaque état de départ, une séquence de valeurs de sortie différente de celles associées aux autres états de départ. Il est donc possible de déterminer l'état initial par application d'une séquence de distinction et observation de la séquence de sortie.

Exemple :

Considérons la machine M_3 donnée figure III-6. La séquence 11001 est une séquence de distinction. Les différentes séquences de sortie obtenues en réponse à cette entrée sont montrées dans le tableau III-3.

	x = 0	x = 1
A	C , 0	B , 0
B	D , 0	A , 0
C	D , 0	E , 1
D	E , 0	C , 0
E	E , 1	B , 1

Figure III-6 : Tableau d'états de la machine M_3.

état initial	séquence de sortie	état final
A	00000	C
B	00001	B
C	11001	B
D	01111	B
E	10000	C

Tableau III-3 : Séquences de sortie de M_3 pour la séquence d'entrée 11001.

Toute séquence de distinction est aussi une séquence de positionnement, la séquence de sortie permettant de déterminer l'état final. Toutefois, une séquence de positionnement n'est pas toujours une séquence de distinction puisqu'il existe des séquences de positionnement qui ne permettent pas de déterminer l'état initial. Une séquence d'entrée est une séquence de distinction si et seulement si elle est une séquence de positionnement qui ne crée pas de convergence. Une séquence d'entrée crée une ***convergence*** si et seulement s'il existe deux états initiaux p et q tels que cette séquence d'entrée amène la machine de l'état p et de l'état q au même état final tout en produisant la même séquence de sortie. La convergence est aisément détectée par la répétition d'un état dans un composant d'un vecteur d'incertitude.

La séquence de distinction peut être obtenue par un arbre des successeurs qui sera nommé l'***arbre de distinction***. Cet arbre est construit de la même façon que l'arbre de positionnement. La seule différence est les règles de terminaison.

Un nœud de l'arbre de distinction devient terminal si :

1) au nœud est associé un vecteur d'incertitude dont les composants sont non homogènes et qui est déjà associé à un des nœuds d'un des niveaux précédents.
2) au nœud est associé un vecteur d'incertitude contenant un composant homogène (même état répété : convergence).
3) à un nœud dans le jème niveau est associé un vecteur d'incertitude trivial.

La troisième règle est appelée règle de succès; les deux premières règles sont les règles d'échec. Une séquence de distinction est déterminée par le chemin qui commence par le vecteur initial d'états et se termine par un vecteur d'incertitude trivial associé à un nœud de l'arbre.

Exemple :

Reconsidérons la machine M_2 (figure III-4), la figure III-7 montre son arbre de distinction.

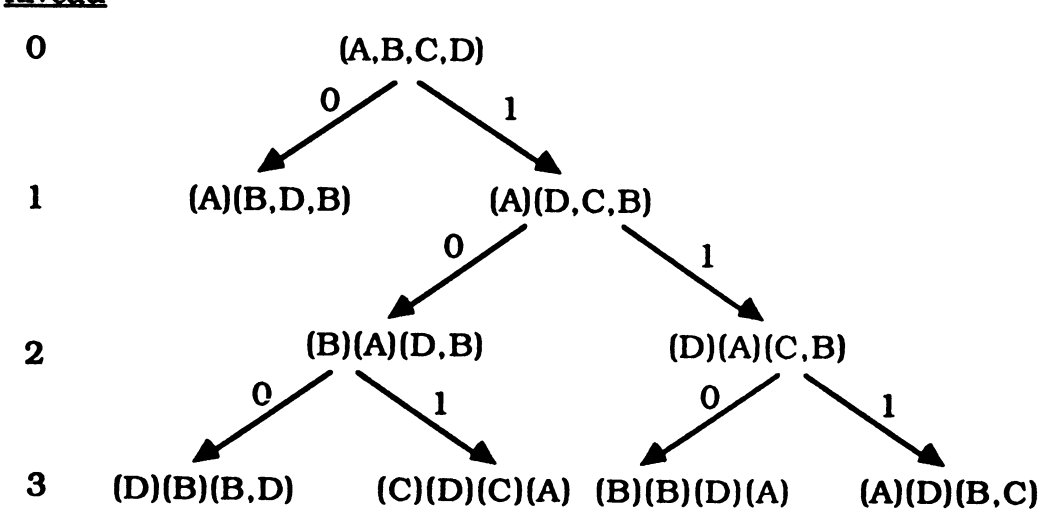


Figure III-7 : Arbre de distinction de la machine M_2.

On remarque que le vecteur d'incertitude (A)(BDB) qui est le 0-successeur du vecteur initial d'états est un terminal puisqu'on a une convergence. Cette convergence est caractérisée par la répétition de l'état B. Les nœuds du niveau 3 sont terminaux parce qu'on a un vecteur d'incertitude trivial (C)(D)(C)(A) associé à un de ses nœuds. Cette machine a deux séquences de distinction de longueur 3 : 101, 110.

L'arbre de distinction d'une machine fournit un moyen simple pour déterminer l'existence d'une séquence de distinction ou non. Tout automate réduit, déterministe, à graphe fortement connexe possède au moins une séquence de positionnement mais pas toujours une séquence de distinction. Par exemple l'arbre de distinction de la machine M_1 se termine au niveau 1 (voir l'arbre de positionnement de cette machine figure 5-III) parce que le vecteur d'incertitude (ABCD) est identique au vecteur initial d'états, et le vecteur (AB)(DD) a un composant caractérisé par une convergence [FRI 71], [LIN 80].

A partir de ces séquences élémentaires, il est possible de construire une séquence de test d'identification. Une méthode de construction sera exposée dans le paragraphe II. 5.

II - TEST D'UN AUTOMATE A PARTIR D'UNE DESCRIPTION DE HAUT NIVEAU

Comme nous avons vu, la description de circuits complexes par le langage de haut niveau LUSTRE est synthétisée sous forme d'automate d'états fini. Nous proposons ici d'étendre l'utilisation des méthodes de test d'automate par identification pour générer un test pour ces automates.

Un circuit sous test est considéré comme une "boîte noire" avec seulement ses entrées/sorties primaires accessibles. En observant les séquences de sorties du circuit produites en réponse à une séquence de vecteur d'entrées, on vérifie si le circuit sous test respecte ses spécifications ou non. Cette méthode de test de "boîte noire" est très adaptée pour les circuits dont la description est donnée par un langage de haut niveau.

Les séquences de test sont indépendantes de la conception effective du circuit à tester et correspondent à une méthode fonctionnelle.

Nous nous intéressons ici aux circuits complexes composés d'une partie contrôle "P. C." et d'une partie opérative "P. O." (figure III-8). La partie opérative est une interconnexion de modules matériels de mémorisation et de calcul permettant d'effectuer un ensemble de traitements. La partie contrôle est un automate qui contrôle les traitements de l'information par la partie opérative.

Notre but est de définir le test par identification de la P.C. à travers la P.O. (les entrées et sorties de la P.C. ne sont pas toutes directement commandables et observables).

Les modules de la P.O. peuvent être soit des éléments de mémorisation qui enregistrent les valeurs de données suivant les commandes provenant de la partie contrôle, soit des circuits combinatoires effectuant des calculs sur les entrées et les variables internes du circuit.

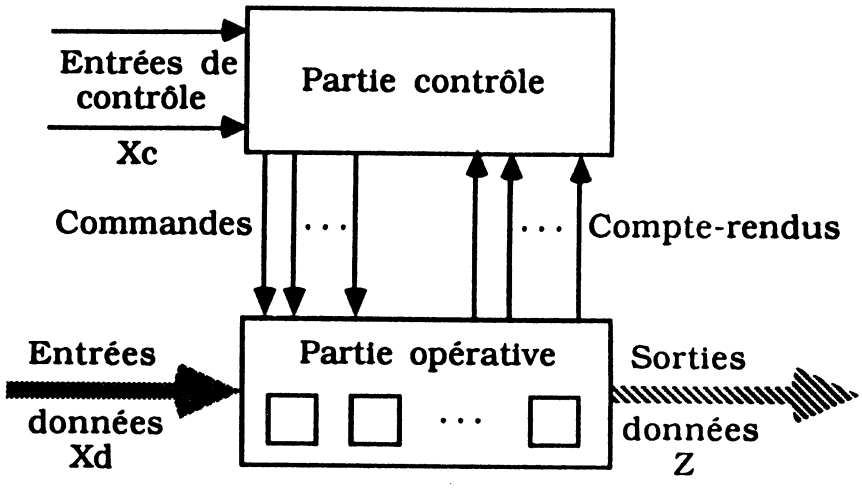


Figure III-8 : Circuit composé d'une partie opérative et d'une partie contrôle.

Nous commençons par généraliser la définition d'un automate par la définition d'un automate généralisé ; le tableau d'états et le graphe d'états associés à un automate généralisé sont décrits et les séquences élémentaires pour établir un test par identification d'un automate généralisé sont définies.

Enfin, la méthode proposée est présentée : il s'agit d'une extension des méthodes classiques de test d'automates pour qu'elles soient applicables aux circuits intégrés plus complexes, composés d'une P.C. et d'une P.O.

II. 1 - Définition d'un automate généralisé

La définition d'un automate généralisé nécessite les définitions suivantes :

Définition :

On appelle ***vecteur d'entrée*** X^i du circuit l'ensemble des valeurs d'entrées de contrôles (commandes) Xc^i et des valeurs d'entrées de données Xd^i à l'instant i (Cf. figure III-8) :

$$X^i = (Xc^i, Xd^i)$$

où $Xc^i \in \{T, F, \varphi\}^k$

T= True, F= False, φ = True ou False (valeur indéterminée)

Une ***séquence de vecteurs d'entrée*** X de longueur m est représentée par une suite : $X^1, X^2, \ldots, X^m$ où X^j est le vecteur d'entrée à l'instant j.

Définition :

L'état de contrôle du circuit à un instant donné est défini par l'état à cet instant de l'automate associé à sa partie contrôle. L'ensemble de tous les états de contrôle possibles d'un circuit Q est :

$$Q = \{q_1, q_2, \ldots, q_n\}$$

Définition :

A chaque module de mémorisation de la P.O. est associée une variable interne. L'état de la P. O. du circuit à un instant donné est défini par le ***vecteur de valeurs*** V_f des variables internes. Ce vecteur de valeurs est défini par l'ensemble des valeurs de toutes les variables internes du circuit (au nombre de p).

$$V_f = (v_1, v_2, \ldots, v_p)$$

Définition :

On appelle ***vecteur de sortie*** l'ensemble de valeurs des sorties du circuit obtenues par l'application d'un vecteur d'entrées à partir d'un état donné.

$$Z_f = (z_1, z_2, \ldots, z_k)$$

où k est le nombre de sorties du circuit.

La définition classique d'un automate par un quintuplet [KOH 70] : M= (*X*, *Z*, *Q*, f, g) ne permet pas de distinguer l'état de la P.C. de l'état de la P.O. Nous définissons un ***automate généralisé*** qui représente un circuit complexe par un septuplet : M= (*X*, *Z*, *Q*, *V*, f, g, h).

où X : est l'ensemble de vecteurs d'entrées.
Z : est l'ensemble de vecteurs de sorties.
Q : est l'ensemble des états de contrôle.
V : est l'ensemble des vecteurs de valeurs.
f : $Q * X * V \rightarrow Q$: est la fonction de transition qui, à un état présent de contrôle, à un vecteur d'entrée et à un vecteur de valeurs, fait correspondre un état suivant de contrôle.
g : $Q * X * V \rightarrow Z$: est la fonction de sortie, (automate de Mealy), qui, à un état de contrôle, à un vecteur d'entrée et à un vecteur de valeurs, fait correspondre un vecteur de sorties.
h : $Q * X * V \rightarrow V$: est la fonction de valeurs internes, (automate de Mealy), qui, à un état de contrôle, à un vecteur d'entrée et à un vecteur de valeurs, fait correspondre un vecteur de valeurs suivant.

L'état du circuit à un instant donné est défini par un couple (état de la partie contrôle, état de la partie opérative) : (q^i, V^i).

II. 1. 1 - Tableau d'états généralisé et Graphe d'états généralisé

La distinction entre P.C. et P.O. d'un circuit complexe nous permet de réduire la complexité en proposant une représentation semi-symbolique. Dans cette représentation, les valeurs des entrées de données ainsi que les valeurs d'un vecteur de valeurs peuvent être des valeurs symboliques. Le vecteur d'entrée défini auparavant devient :

$$X^i = (Xc^i, Xd^i)$$
$$Xci \in \{T, F, \varphi\}^k$$
$$Xd^i = \$Xd^i$$

où T= True, F= False, φ = True ou False (valeur indéterminée) ;
$\$Xd^i$ est une valeur symbolique.

Un tableau d'états généralisé a autant de lignes que d'états de contrôle du circuit et autant de colonnes que de combinaisons possibles de valeurs d'entrées de la partie contrôle. En appliquant un vecteur d'entrées $X_1 \in X$ à

partir d'un état de contrôle du circuit $q_i \in Q$, on obtient l'état suivant de contrôle q_j, le vecteur de valeurs suivant $V_f \in V$ et le vecteur de sorties produites $Z_f \in Z$, tous deux exprimés par des expressions sur les valeurs symboliques du vecteur de valeurs de départ $V_i \in V$.

Le tableau d'états généralisé peut se présenter sous forme de graphe nommé graphe d'états généralisé dans lequel à chaque transition sont associés le vecteur d'entrée/le vecteur de valeurs suivant, le vecteur de sorties produites.

Exemple :

Considérons l'exemple suivant : les entrées/sorties du circuit sont indiquées dans la figure III-13. Le circuit est synchronisé par le front montant du signal d'horloge ck.

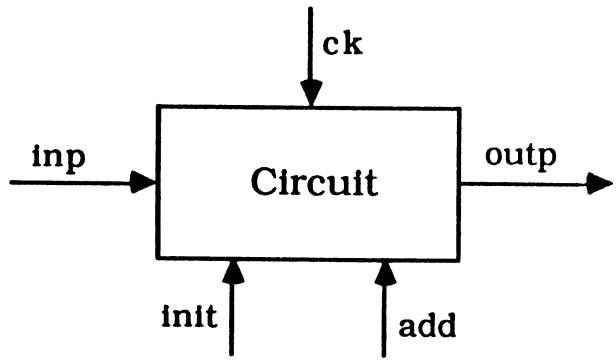


Figure III-13 : Les entrées/sorties du circuit

Les spécifications fonctionnelles du circuit sont exprimées de la façon suivante : quand "init" est faux, le circuit est en état d'oisiveté. Quand "init" est vrai, si "add" est vrai, le circuit effectue l'addition de la valeur actuelle de "inp" avec la valeur suivante à l'instant suivant et mémorise le résultat (quand ck est actif c'est-à-dire quand on a un front montant). Si "add" est faux, le circuit sort la valeur mémorisée et à l'instant suivant les deux registres sont initialisés à zéro. Deux instants d'horloge au moins doivent séparer deux activations.

La structure interne du circuit figure (III-14, a) comporte deux registres ayant un signal de remise à zéro (rs) et un additionneur, le registre reg1 (resp. reg2) est sensible au front montant de clk1 (resp. clk2). Le reg2 a une sortie haute impédance (autorisé par le signal enable). Cette P.O. est

contrôlée par la partie contrôle dont le graphe d'états est donné dans la figure (III-14, b) (automate de type Mealy).

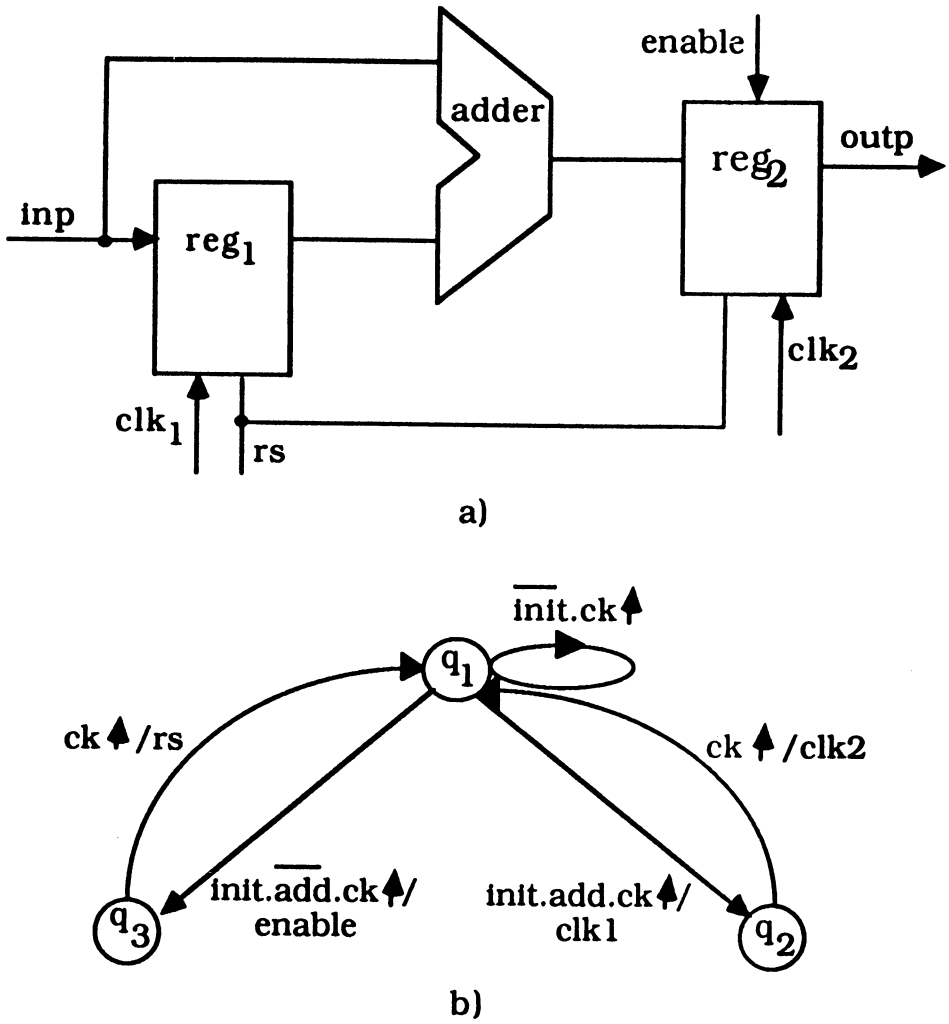


Figure III-14 : a) Partie de données (partie opérative).
b) Partie contrôle du circuit.

La description LUSTRE du circuit est la suivante :

```
node front(clk : bool) returns(trans : bool);
let
trans= clk and (false-> not pre(clk));
tel.
```

```
node automate(init, add : bool) returns(clk1, clk2, enable,rs : bool);
var et1, et2, et3, ck : bool;
let
ck= true-> not pre(ck);
et1= true -> if front(ck) then pre(et2) or pre(et3) or (pre(et1) and not init)
             else pre(et1);
et2= false -> if front(ck) then pre(et1) and init and add
              else pre(et2);
et3= false-> if front(ck) then pre(et1) and init and not add
             else pre(et3);
clk1= false-> if front(ck) then pre(et1) and init and add
              else false;
clk2= false-> if front(ck) then pre(et2)
              else false;
enable= false-> if front(ck) then pre(et1) and init and not add
                else false;
rs= false-> if front(ck) then pre(et3)
            else false;
tel.

node registre(rs, clk : bool; regin : int) returns(regout : int);
let
regout= if rs then 0
        else 0-> pre( if (front(clk) and not rs) then regin
                      else regout );
tel.

node block_level(init, add : bool; inp : int) returns(outp : int);
var clk1, clk2, enable, rs : bool;
    valeur, addition : int;
let
(clk1, clk2, enable,rs)= automate(init, add);
valeur= registre(rs, clk1, inp);
addition= (valeur + inp);
outp= if enable then registre(rs, clk2, addition)
      else 0;
tel.
```

Le nœud "automate" décrit la partie contrôle comme elle est spécifiée par le graphe d'états initial. Les équations sont déterminées en examinant les états antécédents d'un état et les conditions de transition. Le nœud "registre" est utilisé deux fois pour reg_1 et reg_2 en tenant compte du fait que la haute impédance de la sortie est réalisée par la valeur "tout à zéro". On a considéré l'horloge de base comme une horloge interne du nœud principal.

Le tableau d'états généralisé de cet exemple obtenu après la compilation de la description LUSTRE est le suivant :

entrées / PC	init add			
	F F	F T	T F	T T
1	2,($R1,$R2),(0)			
2	1,($R1,$R2),(0)	1,($R1,$R2),(0)	4,($R1,$R2),($R2)	3,($I,$R2),(0)
3	6,($R1,$R2),(0)			
4	5,($R1,$R2),(0)			
5	1,(0,0),(0)			
6	1,($R1,$I+$R1),(0)			

état de contrôle suivant, (vecteur de valeurs suivant), (vecteur de sortie).

Figure III-15 : Tableau généralisé d'états à partir d'une description LUSTRE.

où, pour chaque transition, :

$R1 est la valeur symbolique initiale du registre reg_1 ;

$R2 est la valeur symbolique initiale du registre reg_2 ;

$I est la valeur symbolique de l'entrée "inp".

Le graphe d'états généralisé de cet exemple est montré dans la figure III-16. Il se compose de six états dont trois pseudo-états (les états 1, 3 et 4 correspondant à l'intervalle entre le front descendant et le front montant de l'horloge, le front descendant de l'horloge ne déclenchant aucun changement).

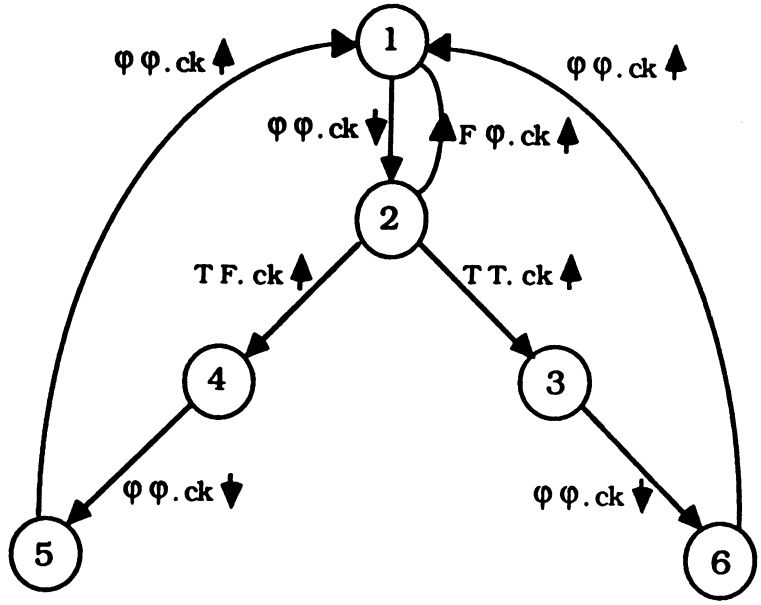


Figure III-16 : Graphe d'états généralisé de cet exemple.

où φ = True ou False (valeur indéterminée).

En éliminant ces pseudo-états, on obtient le graphe d'états généralisé réduit montré dans la figure III-17 et le tableau d'états généralisé réduit montré dans la figure III-18. Les transitions de ce graphe sont synchronisées par le front montant de ck. (I, II, III) est l'ensemble de tous les états de contrôle possibles du circuit.

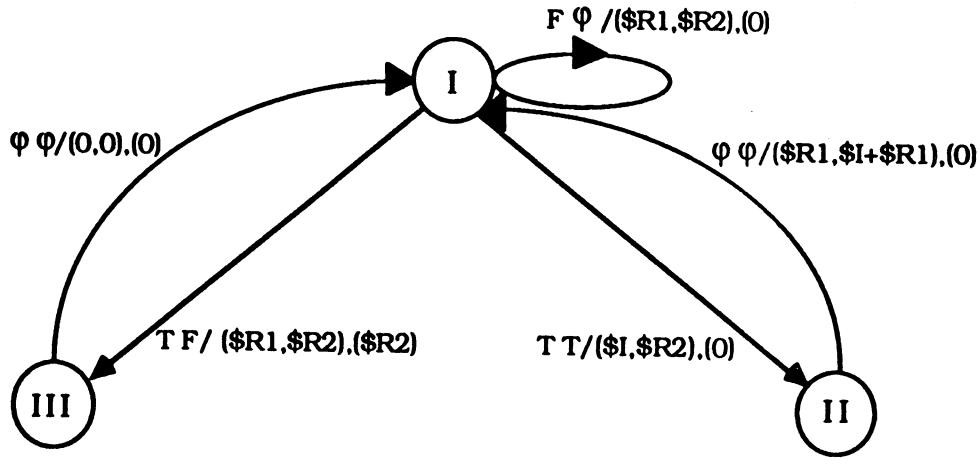


Figure III-17 : Graphe d'états généralisé réduit.

L'application de (init=T, add=F), à partir de l'état de contrôle du circuit I, par exemple, donne III,($R1,$R2),($R2) :

où I est l'état initial de contrôle.
($R1,$R2) est le vecteur de valeurs de départ.
III est l'état suivant de contrôle.
($R1,$R2) est le vecteur de valeurs suivant.
($R2) est le vecteur de sortie produit.

entrées / PC	init add			
	F F	F T	T F	T T
I	I,($R1,$R2),(0)	I,($R1,$R2),(0)	III,($R1,$R2),($R2)	II,($I,$R2),(0)
II	I,($R1,$I+$R1),(0)			
III	I,(0,0),(0)			

Figure III-18 : Tableau d'états généralisé réduit.

Le graphe d'états réduit obtenu a le même nombre d'états que le graphe initial donné dans la figure (III-14, b). La différence est que ce graphe représente la fonction de l'ensemble du circuit, et non la P.C. seulement.

II. 1. 2 - Transitions sous contraintes

L'automate représentant la P.C. d'un circuit complexe peut avoir certaines transitions conditionnées par des comptes-rendus venant de la P.O. Dans notre représentation généralisée semi-symbolique, cela est traduit par des prédicats portant sur les valeurs symboliques des entrées de données ou du vecteur de valeurs de départ d'une transition. On parlera de ***transition sous contraintes***.

La valeur effective de l'entrée de contrôle associé à la transition est complétée par le(s) prédicat(s) sur les valeurs symboliques qui doivent être vérifiés pour l'activation de cette transition.

Exemple :

Soit l'algorithme suivant qui décrit les transitions de l'état q_1 d'un circuit comportant un registre "reg" avec une entrée de donnée "inp" :

q_1 : si not e alors allera q_1
sinon si reg > inp alors (reg:= reg+1 , allera q_2)
sinon allera q_3 ;
...

Soient $R la valeur symbolique initiale du registre "reg" et $I la valeur symbolique de l'entrée "inp". On obtient, après le franchissement de la transition de q_1 à q_2, qui est conditionnée par : e= true et {$R > $I}, le vecteur de valeurs ($R+1) et le vecteur de sortie (0).

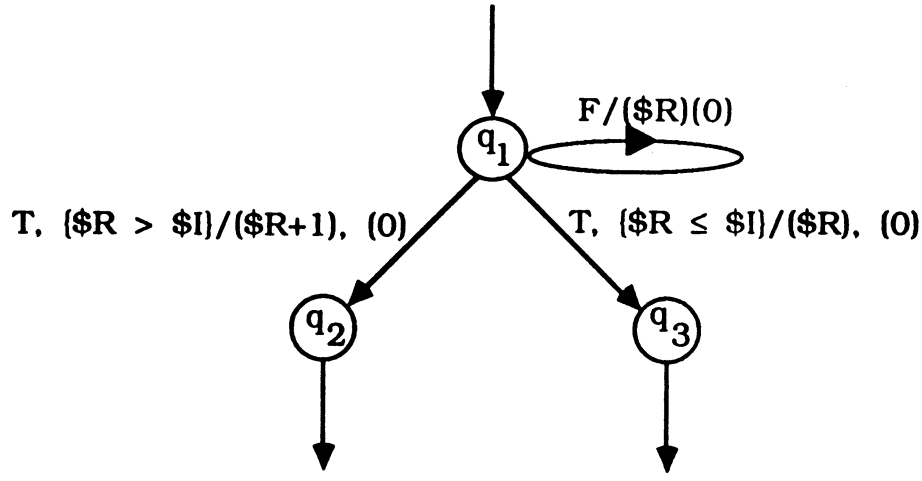


Figure III-19 : Transitions sous contraintes.

C'est sur le graphe d'états généralisé que vont être définies les séquences élémentaires généralisées (synchronisation, positionnement et distinction), qui servent à l'établissement d'une séquence de test, ainsi que les algorithmes de recherche de tels séquences.

II. 2 - Séquence de synchronisation généralisée

II. 2. 1 - Définition de la séquence de synchronisation généralisée

Une ***séquence de synchronisation généralisée*** est une séquence de vecteurs d'entrée qui amène la P.C., quel que soit son état initial de contrôle, au même état final de contrôle q_f et la P.O. à un état spécifique qui ne dépend que des valeurs des entrées de données du circuit.

Une séquence de synchronisation généralisée $S = X^1, X^2, \ldots, X^m$ de longueur m est telle que :

$$\forall\, q_j \in Q\,,\ \forall\, V_j \in V\,,\ f(q_j, S, V_j) = q_f,\ h(q_j, S, V_j) = V_f.$$

où q_j est l'état de départ.
q_f est l'état d'arrivée.
$X^i = (Xc^i, Xd^i)$ est le vecteur d'entrée à l'instant i.
V_j est le vecteur de valeurs de départ.
V_f est le vecteur de valeurs obtenu.

La séquence S est définie par un triplet :

$$S = (Sc, Sd, \{Pi\})$$

où Sc est la séquence des valeurs effectives des entrées de contrôle,
Sd est la séquence des valeurs symboliques des entrées de données et
{Pi} est un ensemble de prédicats sur les valeurs symboliques des entrées de données appliquées au cours de la séquence Sd.

Remarques :

1) Les prédicats associés à une séquence de synchronisation généralisée peuvent être générés soit par le parcours de transitions sous contraintes, soit

par la détermination de valeurs des entrées de données qui permettent de mettre la P.O. dans le même état, quelque soit l'état initial du circuit.

2) La séquence de synchronisation généralisée doit être indépendante du vecteur de valeurs initiales : aucun des prédicats ne doit porter sur des valeurs de ce vecteur.

3) Au cours des prochains paragraphes l'état du circuit désignera l'état de contrôle du circuit, sinon on précisera s'il s'agit de l'état de la P.O.

Il existe deux types de synchronisation généralisée :

a) Synchronisation sans contraintes : quand l'ensemble de prédicats {Pi} est un ensemble vide c'est à dire il existe une séquence de synchronisation sans aucune contrainte sur les valeurs des entrées de donnée du circuit (valeurs libres).

b) Synchronisation avec contraintes : quand certaines valeurs des entrées de donnée sont imposées ou restreintes pour que la séquence d'entrée soit une séquence de synchronisation et on a, donc, un ensemble de prédicats {Pi} non vide.

Exemple :

La séquence S= $(F, \varphi)^1$, $(T, F)^2$, $(\varphi, \varphi)^3$ est une séquence de synchronisation généralisée sans contraintes pour l'exemple précédent, figure III-13.

entrée : init add / PC, PO	F φ	T F	φ φ
I, ($R1,$R2)	I, ($R1,$R2)	III, ($R1,$R2)	I, (0,0)
II, ($R1,$R2)	I, ($R1,$R1+$I1)	III, ($R1,$R1+$I1)	I, (0,0)
III, ($R1,$R2)	I, (0,0)	III, (0,0)	I, (0,0)

Figure III-20 : Succession des états et des vecteurs de valeurs pour la séquence appliquée S= $(F, \varphi)^1$, $(T, F)^2$, $(\varphi, \varphi)^3$.

Cette séquence synchronise la P.C. à l'état I (Cf. graphe d'états généralisé, figure III-17) et la P.O. à un vecteur de valeurs (0, 0).

La séquence S'= (T, T)^1, (T, F)^2, (F, φ)^3 est une séquence de synchronisation généralisée avec contraintes pour le même exemple. Cette séquence synchronise la P.C. à l'état I et la P.O. à un vecteur de valeurs (0, 0) avec les prédicats : \$I^1= 0, \$I^2= 0 (on obtient un vecteur de valeurs égal à (\$I^1, \$I^1+\$I^2), en appliquant cette séquence à partir de l'état I où \$I^1 est la valeur symbolique de l'entrée de donnée à l'instant 1 et \$I^2 est la valeur symbolique de l'entrée de donnée à l'instant 2 ; pour obtenir le même état de la P.O. que dans les autres cas, il faut que \$I^1= \$I^2= 0).

entrée : init add / PC, PO	T T	T F	F φ
I, (\$R1,\$R2)	II, (\$I1,\$R2)	I, (\$I1,\$I1+\$I2)	I, (\$I1,\$I1+\$I2)
II, (\$R1,\$R2)	I, (\$R1,\$R1+\$I1)	III, (\$R1,\$R1+\$I1)	I, (0,0)
III, (\$R1,\$R2)	I, (0,0)	III, (0,0)	I, (0,0)

Figure III-21 : Succession des états et des vecteurs de valeurs pour la séquence appliquée S'= (T, T)^1, (T, F)^2, (F, φ)^3.

II. 2. 2 - Algorithme de recherche de la séquence de synchronisation généralisée

Comme pour le test par identification d'automates simples, la recherche d'une séquence de synchronisation généralisée va utiliser les notions de liste d'états et de X-successeur d'une liste d'états initiale. De plus on ajoute les définitions suivantes :

II. 2. 2. 1 - Définition de la liste de vecteurs de valeurs

On appelle ***Liste de Vecteurs de Valeurs*** "LVV" du circuit la liste formée par les différents vecteurs de valeurs successeurs obtenus par l'application

d'une séquence de vecteurs d'entrée $X = X^1X^2 \ldots X^m$ à partir des différents états d'une liste initiale d'états.

$$V^m = [V_1^m, V_2^m, \ldots, V_n^m] =$$
$$[(v_{1,1}^m, v_{2,1}^m, \ldots, v_{p,1}^m), (v_{1,2}^m, v_{2,2}^m, \ldots, v_{p,2}^m), \ldots, (v_{1,n}^m, v_{2,n}^m, \ldots, v_{p,n}^m)]$$

où V^m représente une LVV.

V_i^m est le vecteur de valeurs obtenu par l'application de X à partir de l'état i.

n est le nombre d'états.

$v_{j,i}^m = v_j$ est la valeur de la $j^{ème}$ variable interne après application de X à partir de l'état i.

La formule suivante définit la ***liste de vecteurs de valeurs initiales*** dont les valeurs sont considérées comme inconnues.

$$V^0 = [V_1^0, V_2^0, \ldots, V_n^0] /$$
$$\forall i \in \{1, 2, \ldots, n\}, \forall s \in \{1, 2, \ldots, p\}, v_{s,i}^0 = x \text{ (valeur inconnue).}$$

Nous définissons une ***Liste de Vecteurs de Valeurs Indépendantes*** "LVVI" par une liste LVV indépendante de la liste de vecteurs de valeurs initiales (ne dépendant que des entrées de données du circuit).

$$LVVI = [V_1^m, V_2^m, \ldots, V_n^m] /$$
$$V_i^m \neq \mathcal{R}(V_i^0), \forall i \in \{1, 2, \ldots, n\}$$

où $\mathcal{R}$ est une fonction de dépendance.

Nous définissons une ***Liste de Vecteurs de Valeurs Indépendantes et Identiques*** "LVVII" par une LVVI qui contient des vecteurs tous identiques, éventuellement en respectant des contraintes sur les entrées de données du circuit.

$$LVVII = [V_1^m, V_2^m, \ldots, V_n^m] /$$
$$\forall i, j \in \{1, 2, \ldots, n\}\ V_i^m \neq \mathcal{R}(V_i^0) \text{ et } V_i^m = V_j^m.$$

II. 2. 2. 2 - Définition de la Liste de Vecteurs de sortie

On appelle ***liste de vecteurs de sortie*** la liste formée par les différents vecteurs de sorties obtenus par l'application d'une séquence de vecteurs d'entrée $X = X^1, X^2, \ldots, X^m$ à partir des différents états d'une liste initiale d'états.

$$Z^m = [Z_1^m, Z_2^m, \ldots, Z_n^m]$$

où Z_i^m est le vecteur de sortie obtenu par l'application de X à partir de l'état i.

La propriété suivante précise les conditions nécessaires et suffisantes pour qu'une séquence de vecteurs d'entrée soit une séquence de synchronisation généralisée.

Propriété 1 :

Une séquence de vecteurs d'entrée S est une séquence de synchronisation généralisée si et seulement si la liste d'états obtenue à partir de la liste initiale d'états en appliquant cette séquence est une LEI (liste d'états identiques) et la liste de vecteurs de valeurs à partir de la liste de valeurs initiales est une LVVII.

Démonstration :

On suppose tout d'abord, que S est une séquence de synchronisation généralisée. Si la liste d'états de contrôle obtenue n'est pas une LEI, ou si la liste de vecteurs de valeurs n'est pas formée de vecteurs de valeurs identiques (ou qui peuvent être rendus identiques) alors S ne positionnerait pas le circuit dans un état global fixé. Si la liste de vecteurs de valeurs obtenue n'est pas indépendante de la liste de vecteurs de valeurs initiales, S ne peut pas être une séquence de synchronisation généralisée parce que l'état de la P.O. final dépend de l'état initial.

Ensuite, on suppose que la liste d'états obtenue par l'application d'une séquence de vecteurs d'entrée du circuit est une LEI et la liste de vecteurs de valeurs est une LVVII. La LEI implique que :

$$\forall\ q_i^0,\ q_j^0 \in Q^0\ ,\ f(q_i^0,\ S,\ V_i^0) = f(q_j^0,\ S,\ V_j^0) = q_f^m.$$

où q_f^m est l'état d'arrivée.

et la liste de vecteurs identiques de valeurs LVVII implique que :

$$\forall\ q_i^0,\ q_j^0 \in Q^0\ \&\ \forall\ V_i^0,\ V_j^0 \in V\ ,\ h(q_i^0,\ S,\ V_i^0) = h(q_j^0,\ S,\ V_j^0) = V_f^m$$

où le vecteur de valeurs V_f^m ne dépend pas de valeurs initiales du circuit.

Donc, quel que soit l'état global initial du circuit, la séquence de vecteurs d'entrée l'amène à un état global fixé. Donc, cette séquence est une séquence de synchronisation généralisée.

II. 2. 2. 3 - Méthode de construction de l'arbre de synchronisation généralisé

L'arbre de synchronisation généralisé est un graphe des successeurs classique modifié pour être appliqué aux circuits qui contiennent une partie contrôle et une partie opérative. Cet arbre est un moyen graphique utilisé pour rechercher s'il existe ou non une séquence de synchronisation pour un circuit. Il est construit de la manière suivante :

La liste d'états et la liste associée de vecteurs de valeurs, obtenu par l'application d'une séquence de vecteurs d'entrée X, sont associées à chaque nœud de l'arbre. On commence par la liste initiale d'états et la liste de vecteurs de valeurs initiales qui sont associées au nœud de niveau 0 (la racine de l'arbre).

A chaque niveau j, on associe à chaque nœud une liste d'états (les états X-successeur obtenus par l'application de $X= X^1, X^2, \ldots, X^j$ à partir des états de la liste initiale d'états) et la liste de vecteurs de valeurs associée (les vecteurs de valeurs successeurs obtenus par l'application de X à partir de la liste de vecteurs de valeurs initiales).

Les niveaux sont numérotés 0, 1, . . ., j, . . . Chaque vecteur d'entrée est associé à un arc de l'arbre. A partir d'un nœud, l'ensemble des arcs est défini par toutes les combinaisons possibles de valeurs des entrées de contrôle Xc. Une séquence de j arcs, à partir de la racine, qui se termine à un nœud au $j^{ème}$ niveau, forme un chemin de longueur j.

Pour synchroniser le circuit, il faut tout d'abord synchroniser la partie contrôle, ensuite la partie opérative. Autrement dit, la procédure pour trouver une séquence de synchronisation généralisée est constituée de deux étapes :

La première étape concerne la synchronisation de la partie contrôle du circuit. Une fois qu'on a synchronisé la partie contrôle du circuit, nous passons à la deuxième étape afin de synchroniser la partie opérative.

Les règles de terminaison pour la première étape sont les suivantes : un nœud du $j^{ème}$ niveau devient terminal lorsqu'un de ces cas aura lieu:

1) au nœud est associée une liste d'états qui est déjà associée à un des nœuds d'un des niveaux précédents ou dans le même niveau avec la même liste de vecteurs de valeurs.
2) la liste d'états associée à un nœud du $j^{ème}$ niveau est une LEI.

La première règle est appelée règle d'échec et la deuxième règle est appelée règle de succès pour cette étape. Si la deuxième règle est validée, alors la partie contrôle est synchronisée et nous passons à la deuxième étape.

La deuxième étape s'applique à partir des nœuds du $j^{ème}$ niveau auxquels sont associées des LEI. Les règles de terminaison pour la deuxième étape sont les suivantes : un nœud dans le $k^{ème}$ niveau devient terminal lorsqu'un de ces cas aura lieu :

1) au nœud est associée une liste d'états qui est déjà associée à un nœud d'un des niveaux précédents ou dans le même niveau avec la même liste de vecteurs de valeurs.
2) la liste d'états associée à un nœud au $k^{ème}$ niveau est une LVVII ou il existe des entrées de données qui peuvent la rendre une LVVII.

Remarque :

Si la deuxième règle de la deuxième étape est validée pour un nœud, les conditions nécessaires et suffisantes de la propriété 1 sont satisfaites et la séquence de vecteurs d'entrée correspondante au chemin parcouru à partir de la racine jusqu'à ce nœud est une séquence de synchronisation généralisée. Cette deuxième règle est appelée règle de succès. Sinon il n'y a pas de séquence de synchronisation généralisée et on cherche une séquence de positionnement généralisée qui sera détaillée par la suite.

Exemple :

La séquence de synchronisation généralisée S= (F, φ)^1, (T, F)^2, (φ, φ)^3 pour l'exemple précédent, figure III-13, peut être déterminée en utilisant l'arbre de synchronisation généralisé. La recherche commence avec la liste initiale d'états (I,II,III) et la liste de vecteurs de valeurs initiales associée $(x_1,x_2)(x_1,x_2)(x_1,x_2)$. x_1 (valeur de reg_1) et x_2 (valeur de reg_2) sont considérées comme des valeurs inconnues.

A un arc est associée un vecteur d'entrée du circuit. Les vecteurs d'entrée à l'instant j sont : a^j, b^j, c^j et ψ^j.

où a^j= F, φ ;
b^j= T, F ;
c^j= T, T ;
ψ^j= φ, φ ;
T= true; F= false ;
φ= true ou false (valeur indéterminée).

La liste d'états et la liste de vecteurs de valeurs obtenus sont associées à chaque nœud dans l'arbre. La figure III-22 montre que la liste d'états (I,I,I) dans le niveau 1 est une LEI. Donc, le nœud correspondant est terminal pour la 1ère étape et la partie contrôle est synchronisée.

La 2ème étape commence avec la LEI (I,I,I) et la liste de vecteurs de valeurs associée : $(x_1,x_2)(x_1,x_1+I^1)(0,0)$ qui sont associées à ce nœud. Le

nœud obtenu par l'application de "a" à l'instant 2 (arc étiqueté a^2) est terminal, parce que la LVV associé est identique à celle du nœud antécédent.

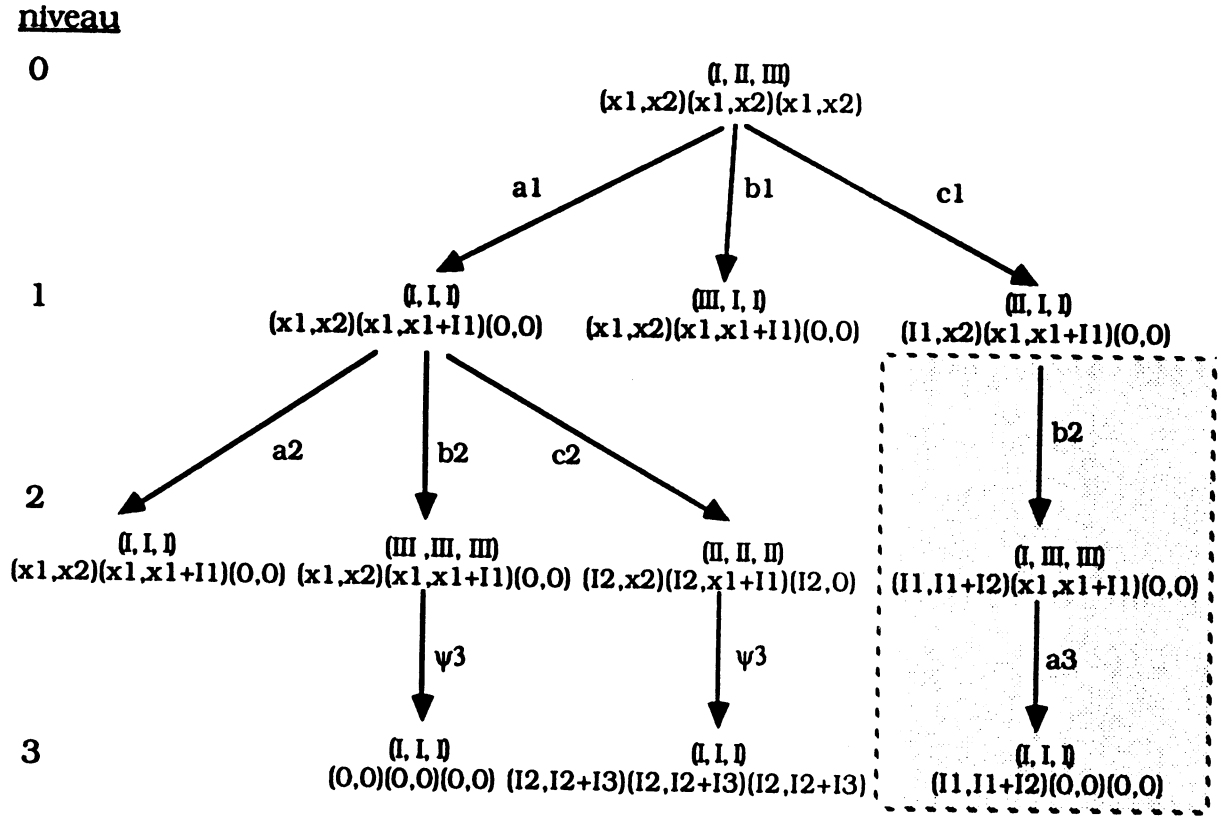


Figure III-22 : Arbre de synchronisation généralisé.

Le circuit de cet exemple possède deux séquences minimales de synchronisation généralisée S= (a^1, b^2, ψ^3), S"= (a^1, c^2, ψ^3). On peut remarquer qu'une séquence de vecteurs d'entrée de longueur 1 est suffisante pour synchroniser la partie contrôle (1ère étape) du circuit alors qu'une séquence de synchronisation généralisée de longueur 3 est nécessaire pour la totalité du circuit.

Remarque :

La partie hachurée à droite illustre la séquence de synchronisation avec contraintes donnée auparavant S'= (c^1, b^2, a^3)= (T, T)^1, (T, F)^2, (F, φ)^3. Cette partie ne serait en fait pas développée par l'algorithme, parce que l'ensemble de la liste d'états (II,I,I) et de la liste de vecteurs de valeurs $(I^1,x_2)(x_1,x_1+I^1)(0,0)$ est associé à un nœud terminal de la première étape.

II. 3 - Séquence de positionnement généralisée

II. 3. 1 - Définition de la séquence de positionnement généralisée

Une ***séquence de positionnement généralisée*** P est une séquence de vecteurs d'entrée qui amène le circuit, quel que soit son état global initial à un état global qui peut être déterminé par observation de la séquence de vecteurs de sortie, indépendamment du vecteur de valeurs initiales.

Cette séquence de positionnement généralisée $P=X^1, X^2, \ldots, X^m$ est définie par un triplet :

$$P = (Pc, Pd, \{C_i\}).$$

où Pc est la séquence des valeurs effectives des entrées de contrôle ;
Pd est la séquence des valeurs symboliques des entrées de données ;
$\{C_i\}$ est un ensemble de prédicats sur les valeurs symboliques des entrées de données appliquées au cours de la séquence P.

Il existe deux types de positionnement généralisé :

1) Positionnement sans contraintes :

Quand l'ensemble de prédicats $\{C_i\}$ est un ensemble vide c'est à dire il existe une séquence de positionnement sans aucune contrainte sur les valeurs des entrées de données Pd du circuit (valeurs libres).

2) Positionnement avec contraintes :

Quand certaines valeurs des entrées de données sont imposées ou restreintes pour que la séquence d'entrée soit une séquence de positionnement et on a, donc, un ensemble de prédicats $\{C_i\}$ non vide.

Exemple :

La séquence P= (T, T)1, (T, F)2, (T, F)3 est une séquence de positionnement généralisée pour l'exemple précédent, figure III-13.

entrée : init add / PC, PO	T T	T F	T F
I, ($R1,$R2)	II, ($I1,$R2)	I,($I1,$I1+$I2)	III,($I1,$I1+$I2)
II, ($R1,$R2)	I,($R1,$R1+$I1)	III,($R1,$R1+$I1)	I, (0,0)
III, ($R1,$R2)	I, (0,0)	III, (0,0)	I, (0,0)

Figure III-23 : Succession des états et des vecteurs de valeurs pour la séquence P= (T, T)1, (T, F)2, (T, F)3.

Cette séquence permet de positionner le circuit à un état de contrôle et à un état de la P.O. connus qui ne dépendent que des entrées de données du circuit, en observant la séquence des vecteurs de sortie qui ne dépend également que des entrées de données du circuit, avec le prédicat $I2+$I1≠ 0.

état initial	séquence de vecteurs de sortie			état final
I	(0)	(0)	($I1+$I2)	III
II	(0)	($R1+$I1)	(0)	I
III	(0)	(0)	(0)	I

Figure III-24 : Succession des vecteurs de sortie pour la séquence appliquée P= (T, T)1, (T, F)2, (T, F)3.

II. 3. 2 - Algorithme de recherche de la séquence de positionnement généralisée

Au lieu d'utiliser la notion de vecteur d'incertitude (Cf. I.2), nous avons utilisé une liste d'états et une liste d'indices associée pour la recherche de la

séquence de positionnement généralisée. La liste d'états est la même liste d'états que celle définie pour construire l'arbre de synchronisation généralisé. La liste d'indices associée permet d'indiquer les états qui sont dans la même liste d'états du vecteur d'incertitude et permet de ne pas effectuer le regroupements d'états dans des listes différentes du vecteur d'incertitude.

II. 3. 2. 1 - Définition de la liste d'indices

On appelle ***liste d'indices*** associée à une liste d'états une liste d'entiers telle que deux états q_i, q_j ont des indices différents si les séquences de vecteurs de sortie obtenus en appliquant $X= X^1, X^2, \ldots, X^m$ à partir de q_i et q_j sont différentes, $\forall$ les valeurs initiales des variables internes.

$$I^m= [I_1{}^m, I_2{}^m, \ldots, I_n{}^m]$$

où I^m représente une liste d'indices ;
$I_i{}^m$ est l'indice associé à l'état i.

On crée autant d'indices qu'il y a de séquences de vecteurs de sortie obtenues distinctes indépendamment des valeurs initiales des variables internes du circuit.

Remarque

La liste d'indices associée à la liste initiale d'états est formée d'une liste dont tous les indices sont identiques.

Une liste d'états à laquelle est associée une liste d'indices, qui sont soit différents pour chaque état de départ, soit identiques mais avec des états d'arrivée identiques, est appelé ***Liste d'Etats Homogène*** "LEH".

Q^m est une LEH si et ssi : $\forall\, q_j{}^m, q_k{}^m \in Q^m$ avec $j \neq k$,
$I_j{}^m \neq I_k{}^m$ ou $q_j{}^m= q_k{}^m$.

Une liste d'états à laquelle est associée une liste d'indices distincts pour chaque état de départ est appelé ***Liste d'Etats Trivial*** "LET".

$\forall\, q_j{}^m, q_k{}^m \in Q^m \Rightarrow I_j{}^m \neq I_k{}^m$ où $j, k \in \{1, 2, \ldots, n\}$.

Une liste d'états trivial est donc une liste d'états homogène dont tous les indices de la liste d'indices associée sont différents.

La propriété suivante précise les conditions nécessaires et suffisantes pour qu'une séquence de vecteurs d'entrée soit une séquence de positionnement généralisée.

Propriété 2

Une séquence de vecteurs d'entrée P est une séquence de positionnement généralisée si et seulement si la liste d'états obtenue à partir de la liste initiale d'états en appliquant cette séquence, est une LEH et la liste de vecteurs de valeurs est une LVVI telle que, si $I_j^m = I_k^m$ alors $V_j^m = V_k^m$.

Démonstration

On suppose tout d'abord, que P est une séquence de positionnement généralisée ; si la liste d'états obtenue par l'application de P n'est pas une LEH alors l'état de contrôle final ne peut pas être déterminé par l'observation de la séquence de vecteurs de sortie. Si la liste de vecteurs de valeurs obtenue a deux vecteurs de valeurs différents pour deux états ayant le même indice, alors l'état de la P.O. final ne peut pas être déterminé par l'observation de la séquence de vecteurs de sortie. Si la liste de vecteurs de valeurs obtenue n'est pas une LVVI alors l'état de la P.O. final dépend de l'état initial de la P.O.

Ensuite, on suppose que la liste d'états obtenue par l'application d'une séquence de vecteurs d'entrée est un LEH et la liste de vecteurs de valeurs est une LVVI telle que si $I_j^m = I_k^m$ alors $V_j^m = V_k^m$.

Donc, à chaque valeur possible des indices, correspondent :

- un état de contrôle, déterminé par l'observation des sorties ;
- un vecteur de valeurs unique indépendant de l'état initial de la P.O., ce qui veut dire que l'état de la P.O. peut être déterminé.

Donc, la séquence de vecteurs d'entrée est une séquence de positionnement généralisée.

II. 3. 2. 2 - Méthode de construction de l'arbre de positionnement généralisé

L'arbre de positionnement généralisé est une extension de l'arbre de positionnement classique. Cet arbre est un moyen graphique utilisé pour trouver une séquence de positionnement généralisée.

A chaque niveau dans l'arbre, on associe à chaque nœud une liste d'états, une liste d'indices et une liste de vecteurs de valeurs. Chaque vecteur d'entrée et le vecteur de sortie obtenu en appliquant ce vecteur d'entrée est associée à un arc dans l'arbre. L'arbre commence par la liste initiale d'états, sa liste d'indices et sa liste de vecteurs de valeurs initiales, associés au nœud de niveau 0.

Chaque liste d'états constitue la X-successeur de la liste initiale d'états où X est la séquence de vecteurs d'entrée correspondante au chemin parcouru. A partir d'un nœud, l'ensemble des arcs est défini par toutes les combinaisons possibles des valeurs des entrées de contrôle Xc.

Les règles de terminaison pour positionner le circuit sont les suivantes : un nœud devient un terminal dans les deux cas suivants :

1) au nœud est associée une liste d'états non homogène qui est déjà associée à un des nœuds dans un des niveaux précédents ou dans le même niveau avec la même liste d'indices et la même liste de vecteurs de valeurs.

2) à un nœud dans le $j^{ème}$ niveau est associée une LET avec une LVVI ou une LEH avec une LVVI telle que si $I_j^m = I_k^m$ alors $V_j^m = V_k^m$.

Remarque :

Si la deuxième règle est validée pour un nœud, les conditions nécessaires et suffisantes de la propriété 2 sont satisfaites et la séquence de vecteurs d'entrée correspondante au chemin parcouru à partir de la racine jusqu'à ce nœud est une séquence de positionnement généralisée. Cette deuxième règle est appelée règle de succès tandis que la première règle est appelée règle de l'échec.

Exemple :

Une séquence de positionnement généralisée, pour l'exemple précédent, figure III-13, est obtenue en utilisant l'arbre de positionnement généralisé, figure III-25. On commence avec la liste initiale d'états (I, II, III), la liste d'indices associée (1, 1, 1) et la liste de vecteurs de valeurs initiales $(x_1,x_2)(x_1,x_2)(x_1,x_2)$. A une transition est associé un vecteur d'entrée du circuit/liste de vecteurs de sortie obtenue en appliquant ce vecteur d'entrée.

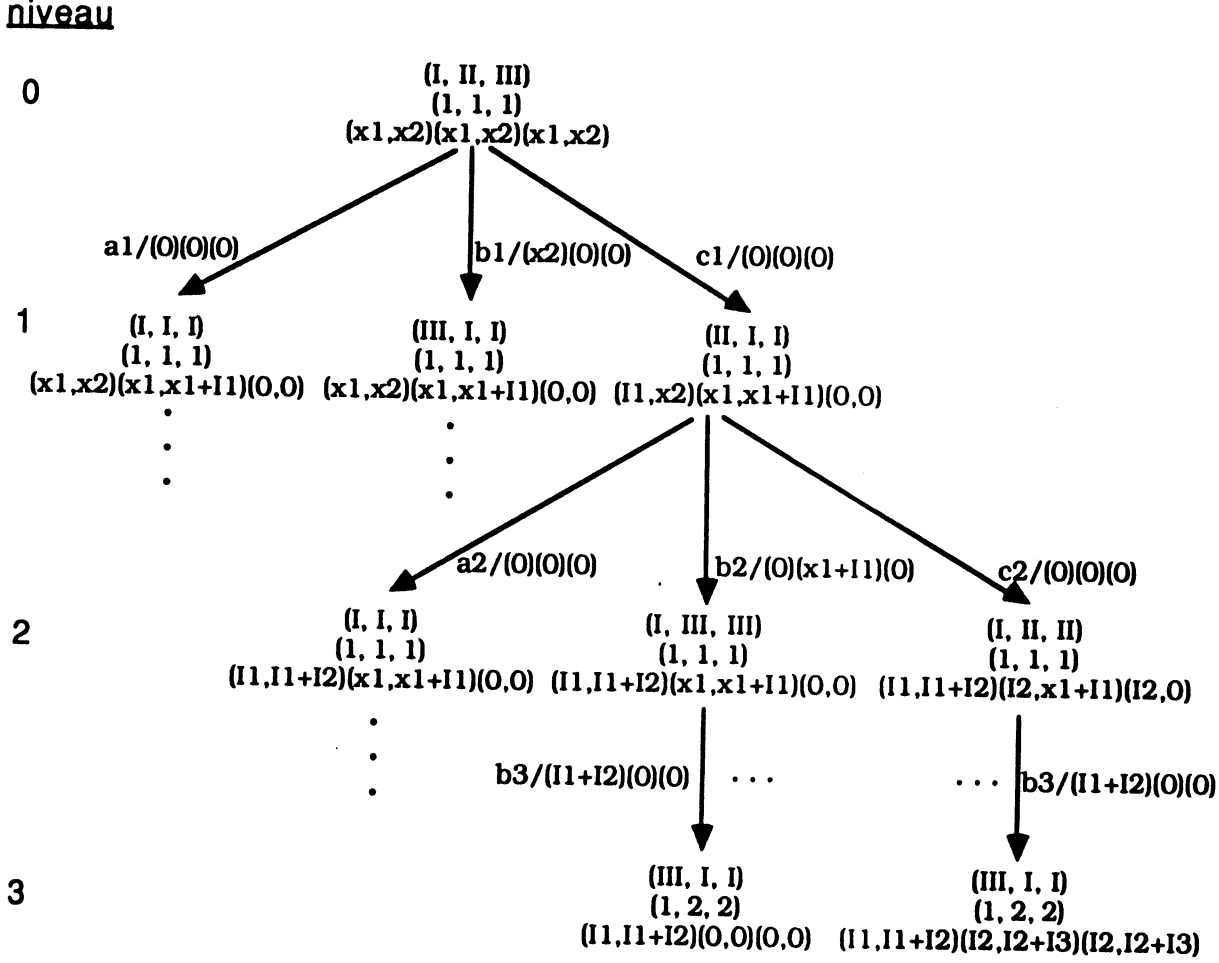


Figure III-25 : Arbre de positionnement généralisé.

Par exemple, le couple : vecteur d'entrée $c1$/liste de vecteurs de sortie (0)(0)(0) est associé à une transition du nœud dans le niveau 0 à un nœud dans le niveau 1. A ce nœud de niveau 1 est associée la liste d'états (II, I, I), la liste d'indices (1, 1, 1) et la liste de vecteurs de valeurs obtenus $(I1,x_2)(x_1,x_1+I1)(0,0)$.

Une séquence de positionnement généralisée est donnée par la séquence de vecteurs d'entrée correspondante au chemin parcouru à partir de la racine jusqu'à l'arrivée au nœud auquel est associé un LEH avec une LVVI associée telle que $I_j^m = I_k^m$ alors $V_j^m = V_k^m$. Par exemple, les séquences P= (c^1, b^2, b^3) et P'= (c^1, c^2, b^3) sont des séquences de positionnement généralisées pour cet exemple.

II. 4 - Séquence de distinction généralisée

II. 4. 1 - Définition de la séquence de distinction généralisée

Une ***séquence de distinction généralisée*** D est une séquence de vecteurs d'entrée du circuit qui produit, pour un état de départ, une séquence de valeurs de vecteurs de sortie différente de celles associées aux autres états de départ indépendamment du vecteur de valeurs initiales.

Note : on cherche à distinguer entre eux les états de contrôle uniquement, puisque notre but est de tester la P.C. à travers la P.O.

Une séquence de distinction généralisée $D = X^1X^2 \ldots X^k \ldots X^m$ est telle que :

$\forall\, q_i^0, q_j^0 \in Q$, avec $i \neq j$, $\forall\, V_i^0, V_j^0 \in V$, $\exists\, k \in \{1, 2, \ldots, m\}$ /
$g(q_i^0, X^1X^2 \ldots X^k, V_i^0) \neq g(q_j^0, X^1X^2 \ldots X^k, V_j^0)$

C'est à dire $Z_i^k \neq Z_j^k$, un au moins des vecteurs de sortie obtenus, qui ne dépend pas du vecteur de valeurs initiales, au cours de l'application de la séquence D est différent suivant l'état initial. (Z_i^k est le vecteur de sortie obtenu au pas k par l'application de la séquence de vecteurs d'entrée $X = X^1X^2 \ldots X^k$ à partir de l'état initial i).

Une séquence de distinction généralisée $D=X^1, X^2, \ldots, X^m$ est définie par un triplet :

$$D = (Dc, Dd, \{C_i\}).$$

où Dc est la séquence des valeurs effectives des entrées de contrôle ;
Dd est la séquence des valeurs symboliques des entrées de données ;
$\{C_i\}$ est un ensemble de prédicats sur les valeurs symboliques des entrées de données appliquées au cours de la séquence D.

Il existe deux types de distinction généralisée :

1) Distinction sans contraintes :

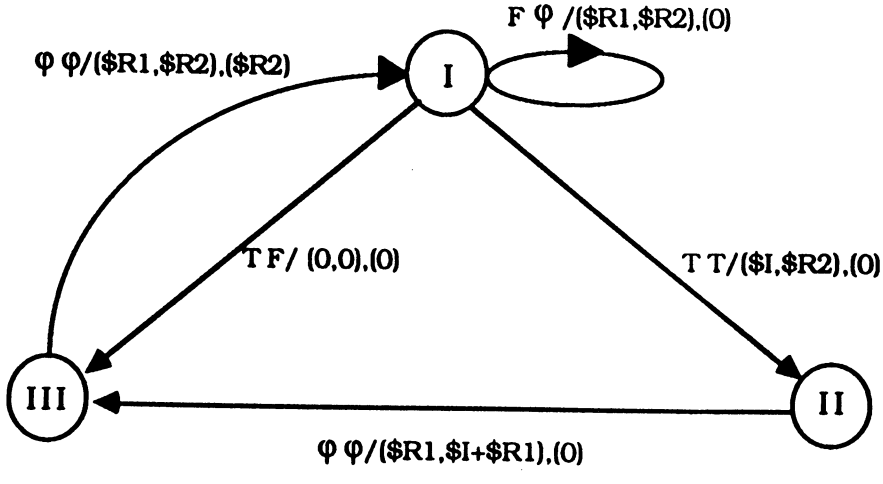
Quand l'ensemble de prédicats $\{C_i\}$ est un ensemble vide c'est à dire il existe une séquence de distinction sans aucune contrainte sur les valeurs des entrées de données Dd du circuit.

2) Distinction avec contraintes :

Quand certaines valeurs des entrées de données sont imposées ou restreintes pour que la séquence d'entrée soit une séquence de distinction généralisée et on a, donc, un ensemble de prédicats $\{C_i\}$ non vide.

Exemple :

Prenons une variante de l'exemple précédent avec les modifications qui sont montrées dans le graphe d'états généralisé, figure (III-26, a) et le tableau d'états généralisé, figure (III-26, b). La séquence D= (T, T)1, (T, T)2, (T, T)3, (T, T)4 est une séquence de distinction généralisée pour cet exemple.



a)

entrées / PC	init add			
	F F	F T	T F	T T
I	I,($R1,$R2),(0)	I,($R1,$R2),(0)	III,(0,0),(0)	II,($I,$R2),(0)
II	III,($R1,$I+$R1),(0)			
III	I,($R1,$R2),($R2)			

b)

Figure III-26 : a) Graphe d'états généralisé ;
b) Tableau d'états généralisé.

La séquence de vecteurs de sortie obtenus pour les trois états initiaux ainsi que l'état final sont montrés dans la figure III-27.

état initial	séquence de vecteurs de sortie				état final
	T T	T T	T T	T T	
I	(0)	(0)	($I1+$I2)	(0)	II
II	(0)	($R1+$I1)	(0)	(0)	III
III	($R2)	(0)	(0)	($I2+$I3)	I

Figure III-27 : Succession des vecteurs de sortie
pour la séquence appliquée D= (T, T)1, (T, T)2, (T, T)3, (T, T)4.

Pour bien distinguer les trois états, il faut que les prédicats suivants sur les valeurs d'entrées de données soient satisfaits : ($I1+$I2 ≠ 0 ; $I2+$I3 ≠ 0). Cette séquence permet, suivant la séquence de vecteurs de sortie, de déterminer l'état initial du circuit.

Une séquence de vecteurs d'entrée X crée une ***convergence*** si l'état final obtenu par l'application de cette séquence à partir de deux états initiaux q_i^0, $q_j^0 \in Q^0$ est le même avec deux vecteurs de valeurs identiques, tout en produisant des séquences de vecteurs de sortie égales (ou qui peuvent être égales) :

$$\exists\ V_i^0, V_j^0 \ /\ f(q_i^0, X, V_i^0) = f(q_j^0, X, V_j^0),\ g(q_i^0, X, V_i^0) = g(q_j^0, X, V_j^0),\ h(q_i^0, X, V_i^0) = h(q_j^0, X, V_j^0).$$

avec $i \neq j$ où V_i^0 et V_j^0 sont les vecteurs de valeurs initiales.

La propriété suivante précise les conditions suffisantes pour avoir une séquence de distinction généralisée.

Propriété 3

Une séquence de vecteurs d'entrée est une séquence de distinction généralisée si elle est une séquence de positionnement généralisée qui ne provoque pas de convergence.

Démonstration

Si on a appliqué une séquence de positionnement généralisée sans convergence à partir de deux états initiaux et on a observé la même séquence de vecteurs de sortie obtenus, le circuit peut être : soit dans le même état final avec le même vecteur de valeurs, soit dans le même état final avec deux vecteurs de valeurs différents, soit dans deux états différents (avec deux vecteurs de valeurs identiques ou différents).

Le premier cas est éliminé parce que la séquence de vecteurs d'entrée est supposée telle qu'elle ne provoque pas de convergence. Le deuxième cas est éliminé aussi si la séquence d'entrée est une séquence de positionnement généralisée suivant la définition de la séquence de positionnement généralisée (la séquence de positionnement généralisée positionne la P.C. et aussi la P.O.). Le troisième cas est éliminé aussi pour la même raison.

Cette contradiction établit que la séquence de vecteurs d'entrée est une séquence de distinction généralisée.

II. 4. 2 - Méthode de construction de l'arbre de distinction généralisé

L'algorithme de recherche de la séquence de distinction généralisée est basé sur la liste d'indices définie auparavant.

Deux états initiaux sont distingués (ont deux indices différents) quand les vecteurs de sortie correspondants sont différents (ou peuvent être différents en respectant des contraintes sur les valeurs des entrées de donnée). Une convergence se produit quand la séquence de vecteurs d'entrée appliquée peut mener le circuit au même état final avec le même indice et le même vecteur de valeurs, pour deux états initiaux différents.

L'arbre de distinction généralisé est une extension de l'arbre de distinction classique. La différence est qu'on associe à chaque nœud dans l'arbre une liste d'états, la liste d'indices et la liste de vecteurs de valeurs.

L'arbre de distinction généralisé commence par la liste initiale d'états avec sa liste d'indices et sa liste de vecteurs de valeurs initiales. Chaque liste d'états constitue la X-successeur de la liste initiale d'états où X est la séquence de vecteurs d'entrée correspondante au chemin parcouru.

Les règles de terminaison sont les suivantes : un nœud devient un terminal dans les trois cas suivants :

1) au nœud est associée une liste d'états non homogène qui est déjà associée à un des nœuds dans un des niveaux précédents avec la même liste d'indices et la même liste de vecteurs de valeurs.

2) au nœud est associée une liste d'états caractérisée par une convergence.

3) à un nœud dans le $j^{ème}$ niveau est associée une LET.

Les deux première règles sont des règles d'échec tandis que la troisième règle est la règle de succès.

Exemple :

La détermination d'une séquence de distinction généralisée, pour l'exemple montré dans la figure III-26, est illustrée dans la figure III-28. La liste d'états obtenue en appliquant c^1, par exemple, est (II, III, I), sa liste d'indices est (1, 1, 1) et la liste de vecteurs de valeurs est

$(I_1,x_2)(x_1,x_1+I^1)(x1,x2)$. L'ensemble de cette liste d'états, la liste d'indices et la liste de vecteurs de valeurs obtenues est associé à un nœud dans le niveau 1.

Une séquence de distinction généralisée est déterminée par la séquence de vecteurs d'entrée correspondant au chemin parcouru à partir de la racine jusqu'à l'arrivée à un LET. Par exemple, la séquence D= (c^1, c^2, c^3, c^4) avec $I^1 + I^2 \neq 0$, $I^2 + I^3 \neq 0$ est une séquence de distinction généralisée pour cet exemple.

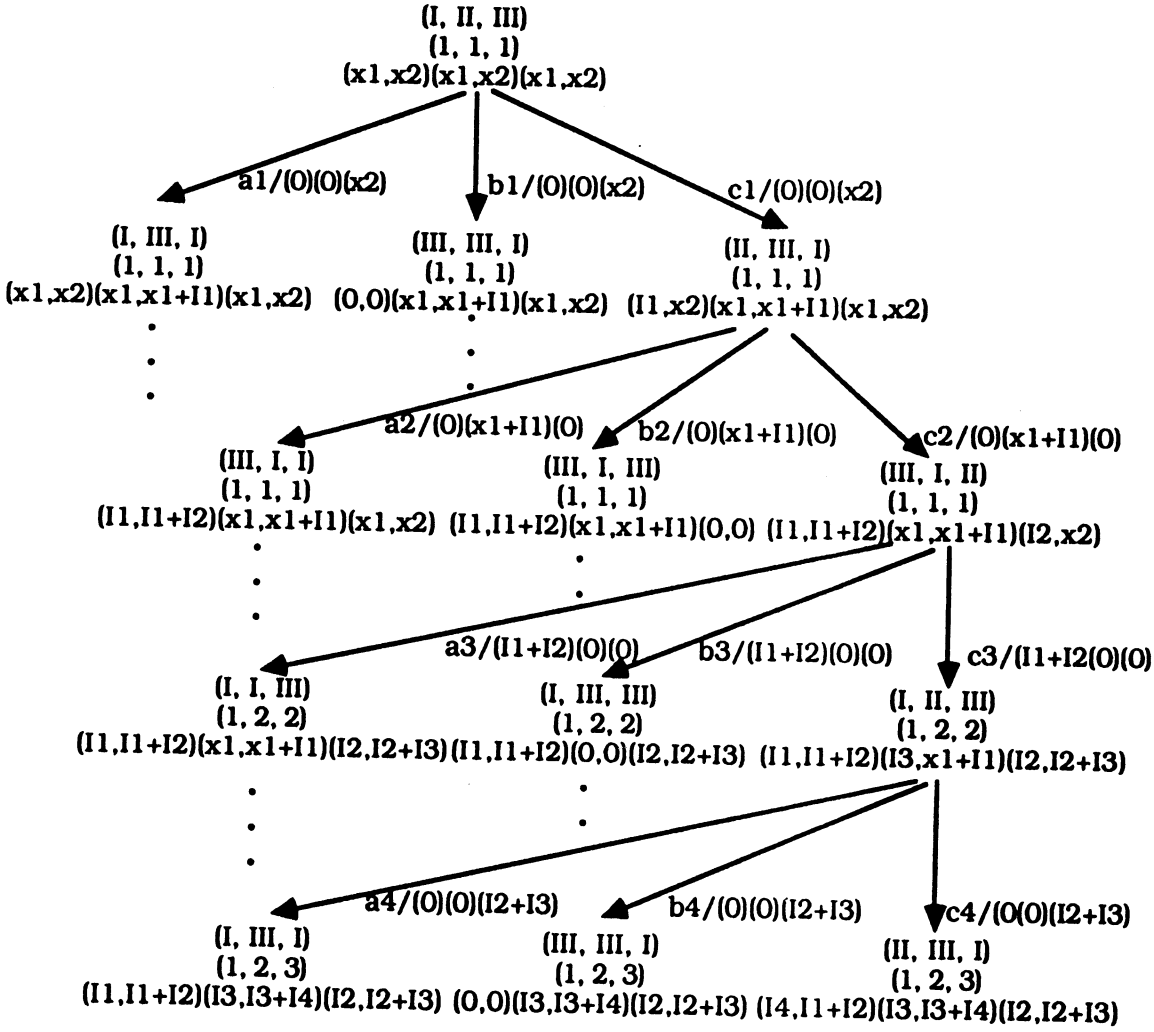


Figure 28 : Arbre de distinction généralisé.

II. 5 - Etablissement d'une séquence de test d'un circuit

Une séquence de test peut être construite en utilisant les méthodes classiques proposées par Lin [LIN 80] et Hennie [HEN 68]. Cette séquence de test se compose de trois parties en utilisant les séquences élémentaires généralisées définies auparavant :

1ère partie : le circuit étant dans un état initial inconnu, on cherche à le positionner dans un état prédéterminé par une séquence de synchronisation généralisée (si le circuit possède une telle séquence) ou par une séquence de positionnement généralisée suivie éventuellement d'une séquence de transfert. Cette première partie devient alors adaptative parce que la séquence de transfert à appliquer sera différente suivant l'état obtenu à l'issue de l'application de la séquence de positionnement généralisée c'est à dire suivant la séquence de vecteurs de sortie obtenue.

2ème partie : distinguer tous les états du circuit en utilisant la séquence de distinction généralisée (si le circuit possède une telle séquence). On applique cette séquence à chaque état de départ et on vérifie le résultat par le vecteur de sortie obtenu. Cette deuxième partie vérifie quelques transitions.

3ème partie : vérifier toutes les transitions qui ne sont pas testées dans la deuxième partie à partir de chaque état de départ. La vérification des transitions a pour but de vérifier la fonction de calcul de l'état suivant (f), la fonction de calcul des sorties (g) et la fonction de calcul des variables internes (h). Cette vérification se fait par l'application du vecteur d'entrée à tester suivie par une séquence de distinction généralisée pour bien identifier l'état d'arrivée d'après le vecteur de sortie jusqu'à ce que toutes les transitions soient vérifiées.

II. 5. 1 - Circuit avec séquence de distinction généralisée

La séquence de test pour un circuit ayant une séquence de distinction généralisée est déterminée en utilisant une extension de la méthode de Hennie de la façon suivante :

Soit $D = X^1X^2 \ldots X^m$ une séquence de distinction généralisée du circuit :

$$q_f = f(q_i, D, V_i).$$

où q_f l'état final du circuit quand on applique la séquence de distinction généralisée D à l'état q_i et V_i est le vecteur de valeurs de départ.

Soit q_1 l'état du circuit après l'application de la 1ère partie du test. La 2ème partie de test est constituée par la séquence suivante :

$$D\ T(q_1, q_2)\ D\ T(q_2, q_3) \ \ldots\ D\ T(q_n, q_1)\ D$$

où la séquence de transfert $T(q_i, q_j)$ mène le circuit de l'état q_i à l'état q_j. Notons que cette séquence vérifie de plus les transitions qui sont utilisées.

Pour chaque transition non encore testée, on applique la séquence suivante :

$$D\ T(q_i, q_j)\ X\ D$$

L'application de D vérifie l'arrivée effective du circuit à l'état q_i. $T(q_i, q_j)$ est un transfert dans l'état q_j à partir duquel on veut vérifier la transition pour le vecteur d'entrée X. (X D) est le test effectif composé de l'application du vecteur d'entrée X et de la séquence de distinction généralisée.

Tous les transferts doivent être effectués en utilisant des transitions déjà testées. Chaque transition dans le tableau d'états généralisé est vérifiée à la fin de cette 3ème partie de test par cette méthode.

Exemple :

En appliquant cette méthode pour l'exemple montré dans la figure III-26, la séquence de synchronisation généralisée $S = (a^1, a^2, b^3)$ constitue la 1ère partie du test. L'application de S mène le circuit à l'état III avec un vecteur de valeurs (0,0). La deuxième partie du test est de vérifier la séquence de vecteurs de sortie obtenu pour chaque état de départ en appliquant la séquence de distinction généralisée. On peut choisir comme séquence de

distinction généralisée la séquence D= (c^1, c^2, c^3, c^4). Pour bien distinguer les trois états, il faut que l'ensemble de prédicats suivants sur les valeurs des entrées de données soit satisfait chaque fois on applique la séquence D : ($\$I^1+\$I^2 \neq 0$; $\$I^2+\$I^3 \neq 0$). La séquence de la deuxième partie de test alors est la suivante :

S D D D D.

La figure III-29 montre l'état suivant, le vecteur de valeurs obtenu à la fin de chaque séquence et les prédicats imposées par les différentes séquences appliquées.

séquence appliquée	état suivant	vecteur de valeurs obtenu	prédicats
S=(a1,a2,b3)	III	(0,0)	
D= (c4,c5,c6,c7)	I	(I5, I5+I6)	I5 + I6 ≠ 0
D= (c8,c9,c10,c11)	II	(I11, I8+I9)	I8 + I9 ≠ 0
D=(c12,c13,c14,c15)	III	(I14, I14+I15)	
D=(c16,c17,c18,c19)	I	(I17, I17+I18)	I17 + I18 ≠ 0

Figure III-29 : L'état suivant, le vecteur de valeurs obtenu et les prédicats imposés pour vérifier tous les états.

Pour la vérification des transitions (3ème partie), il faut concaténer la séquence précédente avec la séquence suivante :

T(I,III) D T(I,II) D T(III,I) D T(II,III) D T(I,I) D

La figure III-30 montre l'état suivant, le vecteur de valeurs obtenu à la fin de chaque séquence et les prédicats demandés.

séquence appliquée	état suivant	vecteur de valeurs obtenu	prédicats
T(I,III)=b20	III	(0,0)	
D= (c21,c22,c23,c24)	I	(I22, I22+I23)	I22 + I23 ≠ 0
T(I,II)=c25	II	(I25,I22+I23)	
D= (c26,c27,c28,c29)	III	(I28, I28+I29)	
T(III,I)= φ30	I	(I28,I28+I29)	
D= (c31,c32,c33,c34)	II	(I34, I31+I32)	I31 + I32 ≠ 0
T(II,III)= φ35	III	(I34,I34+I35)	
D= (c36,c37,c38,c39)	I	(I37, I37+I38)	I37 + I38 ≠ 0
T(I,I)=c40	I	(I37,I37+I38)	
D= (c41,c42,c43,c44)	II	(I44, I41+I42)	I41 + I42 ≠ 0

Figure III-30 : Les prédicats imposés pour tester les différentes transitions.

II. 5. 2 - Circuit sans séquence de distinction généralisée :

La difficulté principale pour le test de circuits qui n'ont pas de séquence de distinction généralisée, est de ne pas pouvoir déterminer l'état du circuit à chaque instant par l'observation des séquences de vecteurs de sortie.

L'approche générale est d'établir une partition des états du circuit afin d'obtenir une séquence de vecteurs d'entrée simple (prédéterminée ou adaptative) qui distingue chaque bloc de la partition.

Un ensemble de séquences de vecteurs d'entrée X, telle que l'ensemble des séquences de vecteurs de sortie produits à partir d'un état q_i de circuit en réponse à ces séquences est différente de l'ensemble des séquences de vecteurs de sortie à partir des autres états avec éventuellement

quelques contraintes concernant les valeurs d'entrées de données, est appelé ***ensemble de séquences de caractérisation généralisée.***

Une partition $\mathcal{Q}_i$ de l'ensemble $\mathcal{Q}$ est telle que :

$$\mathcal{Q}_i \in P(\mathcal{Q}) \text{ et } \mathcal{Q}_i \neq \phi \;/\; \bigcup_i \mathcal{Q}_i = \mathcal{Q} \text{ et } \mathcal{Q}_i \cap \mathcal{Q}_j = \phi \text{ pour } i \neq j$$

où $P(\mathcal{Q})$ est l'ensemble qui contient tous les sous-ensembles de $\mathcal{Q}$.

Une partition des états du circuit est dite induite par une séquence de vecteurs d'entrée X telle que deux états $q_i, q_j, \in \mathcal{Q}$ sont dans le même bloc si et seulement si les séquences de vecteurs de sortie produits par l'application de X à partie de q_i et q_j sont identiques indépendamment des valeurs initiales des variables internes.

On appelle ***0_partition*** induite par une séquence de vecteurs d'entrée X une partition telle qu'il existe un seul membre (singleton) dans chaque bloc. Cette séquence est, alors, une séquence de distinction généralisée.

Le produit de deux partitions P et P' est une partition dans laquelle deux états sont dans le même bloc si et seulement s'ils sont dans le même bloc dans P et dans P'.

L'ensemble de séquences de caractérisation généralisée pour un circuit est un ensemble des séquences de vecteurs d'entrée telles que le produit des partitions induites par ces séquences est une 0_partition.

La ***séquence d'identification généralisée*** (ou séquence de localisation généralisée) pour un état q_i du circuit, est une séquence de vecteurs d'entrée telle que la séquence de vecteurs de sortie obtenue identifie cet état en utilisant l'ensemble de séquences de caractérisation généralisée.

Remarque : Une séquence de distinction généralisée est une séquence d'identification généralisée pour tous les états.

La procédure qui donne la séquence d'identification généralisée est la suivante :

Considérons $\{Y_1, Y_2\}$ l'ensemble de séquences de caractérisation généralisée. On suppose que le circuit est dans l'état initial q_i. $f(Y_1, q_i) = q_j$ où q_j désigne l'état suivant par l'application de Y_1. $T(q_j, q_i)$ désigne la séquence de transfert qui mène le circuit de q_j à q_i. On suppose aussi que (m) états au plus donnent la même séquence de vecteurs de sortie en réponse à la séquence de vecteurs d'entrée Y_1.

Pour vérifier que le circuit est dans l'état q_i, il est nécessaire d'appliquer m+1 fois la séquence $[Y_1\ T(q_j, q_i)]$. La séquence d'identification généralisée L_i pour l'état q_i est définie par l'équation suivante [KOH 70] :

$$L_i = [Y_1\ T(q_j, q_i)]^{m+1}\ Y_2$$

De façon similaire pour les autres états du circuit les séquences d'identification peuvent être dérivées.

On appelle séquence d'identification généralisée du premier degré la séquence qui est constituée d'une seule séquence de caractérisation généralisée, les séquences d'identification généralisée du type $[Y_1\ T(q_j, q_i)\ ]^{m+1}\ Y_2$ sont appelées séquences d'identification généralisée de deuxième degré.

La 2ème partie du test pour ces circuits est constituée d'une séquence d'identification généralisée pour chaque état du circuit. Supposons que L_i est la séquence d'identification généralisée pour l'état s_i et que q_i est l'état d'arrivée après l'application de L_i. La séquence de test est construite par une alternance de séquences d'identification généralisées et des séquences de transfert appropriées :

$$L_1\ T(q_1, q_2)\ L_2\ T(q_2, q_3)\ L_3\ .\ .\ .\ .\ ,\ T(q_{n-1}, q_n)\ L_n$$

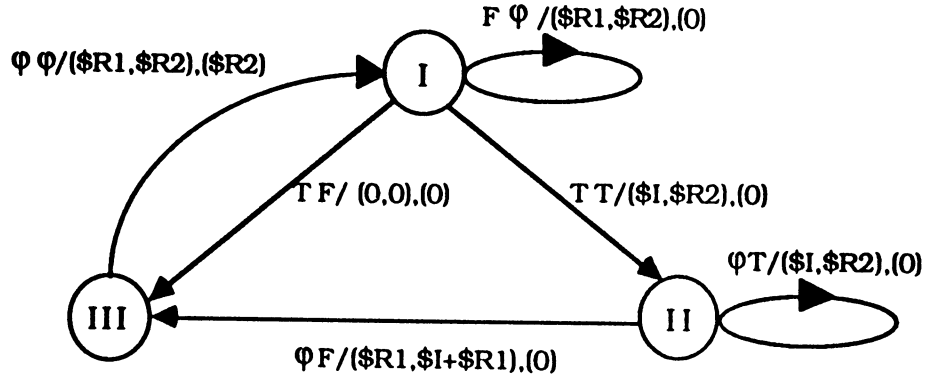
La 3ème partie du test est constituée, pour chaque transition non encore testée, par la séquence suivante :

$$L_m\ T(q_i, q_j)\ X\ L_k.$$

Cette séquence est utilisée pour vérifier la transition de l'état q_j par le vecteur d'entrée X en utilisant les séquences d'identification généralisée L_m et L_k.

Exemple :

Prenons une variante de l'exemple précédent, figure III-26, en ajoutant une transition à partir de l'état II montrées dans le graphe d'états généralisé, figure (III-31, a) et le tableau d'états généralisé, figure (III-31, b).



a)

entrées / PC	init add			
	F F	F T	T F	T T
I	I,($R1,$R2),(0)		III,(0,0),(0)	II,($I,$R2),(0)
II	III,($R1,$I+$R1),(0)	II,($I,$R2),(0)	III,($R1,$I+$R1),(0)	II,($I,$R2),(0)
III	I,($R1,$R2),($R2)			

b)

Figure III-31 : a) Graphe d'états généralisé ;
b) Tableau d'états généralisé.

Il n'y a pas de séquences de distinction généralisée pour cet exemple parce qu'aux nœuds sont associées les listes d'états (I,III,I), (I,II,I) et (II,II,I) qui sont caractérisés par des convergences, figure 32, et le développement de la liste d'états (III,III,I) mène aussi à des nœuds caractérisés par des convergences.

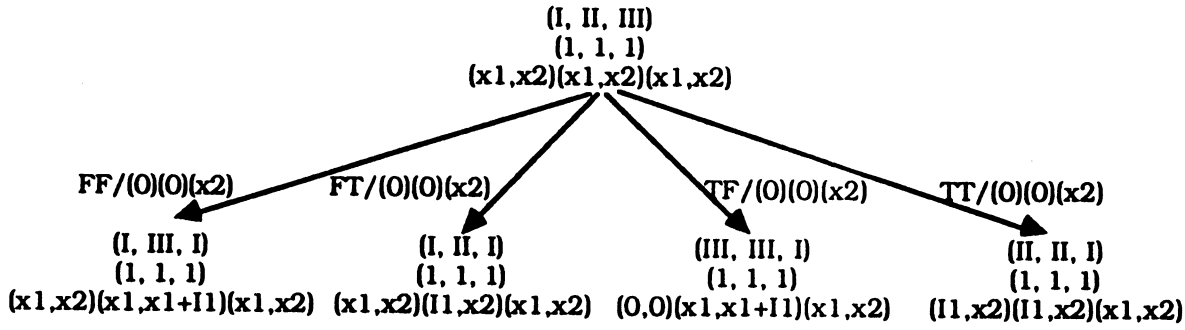


Figure 32 : Arbre de distinction généralisée pour cet exemple.

On cherche des séquences de caractérisation généralisées : l'ensemble $\{Y_1, Y_2\}$ où Y_1= FT, FF, TF et Y_2= TT, FF, TF est un ensemble de séquences de caractérisation généralisée. La séquence de vecteurs de sortie obtenus pour les trois états initiaux ainsi que l'état final en appliquant la séquence Y_1= FT, FF, TF sont montrés dans la figure III-33. La séquence de vecteurs de sortie obtenus en appliquant la séquence Y_2= TT, FF, TF sont montrés dans la figure III-34.

On constate que le produit de $Y_1 * Y_2$ est une 0-partition où la partition des états induite par la séquence Y_1 est {{I, III}, {II}} et la partition des états induite par la séquence Y_2 est {{I, II}, {III}}.

Les séquences d'identification généralisée pour chaque état du circuit sont les suivantes :

$L_{II} = Y_1$; $L_{III} = Y_2$; $L_I = [Y_1\ T(III, I)]^3\ Y_2$.

état initial	séquence de vecteurs de sortie			état final	prédicat
	F T	F F	T F		
I	(0)	(0)	(0)	III	
II	(0)	(0)	($I1+$I2)	I	$I1+$I2≠0
III	($R2)	(0)	(0)	III	

Figure III-33 : Succession des vecteurs de sortie pour la séquence appliquée Y_1= FT, FF, TF.

état initial	séquence de vecteurs de sortie			état final	prédicat
	T T	F F	T F		
I	(0)	(0)	($I1+$I2)	I	$I1+$I2≠0
II	(0)	(0)	($I1+$I2)	I	$I1+$I2≠0
III	($R2)	(0)	(0)	III	

Figure III-34 : Succession des vecteurs de sortie pour la séquence appliquée Y_2= TT, FF, TF.

On remarque que L_{II} et L_{III} sont des séquences d'identification généralisées du premier degré et L_I est une séquence d'identification généralisée de deuxième degré.

La première partie de test est constituée par la séquence de synchronisation généralisée S= FF, FF, TF, par exemple, qui mène le circuit à l'état III. La deuxième partie de test est constituée par la séquence suivante :

$$L_{III}\ T(III, I)\ L_I\ T(I, II)\ L_{II}$$

La troisième partie du test est constituées par la séquence suivante :

$$T(I,I)\ L_I\ T(I,II)\ L_{II}\ T(I,III)\ L_{III}\ T(III,I)\ L_I\ T(I,II)\ T(II,II)\ L_{II}\ T(I,II)\ T(II,III)\ L_{III}$$

II. 6 Logiciel d'aide à la génération des séquences généralisée

Un logiciel de détermination des différentes séquences élémentaires généralisées (synchronisation, positionnement, distinction) a été réalisé en PASCAL sur μVAX VMS.

Les algorithmes de calculs pour ce logiciel sont les suivants : le premier algorithme concerne la séquence de synchronisation généralisée et le deuxième algorithme concerne la séquence de positionnement (respec. distinction) généralisée :

a) Algorithme 1

L'algorithme de calcul suivant précise les étapes nécessaires pour obtenir une séquence de synchronisation généralisée :

1) Création d'un nœud auquel sont associées la liste initiale d'états et la liste de vecteurs de valeurs initiales.
2) Pour chaque nœud dans le graphe construit partiellement et pour toutes les combinaisons possibles de valeurs des entrées de contrôle Xc_i, on détermine la liste d'états Xc_i-successeur de la liste d'états traitée et la liste de vecteurs de valeurs. Si un nœud auquel sont associées les mêmes listes n'existe pas déjà dans le graphe, on ajoute au graphe ce successeur du nœud traité.
3) On répète l'étape 2 jusqu'à l'arrivée à un nœud dont la liste d'états est une LEI (liste d'états identiques).
4) Pour chaque nœud dont la liste d'états est une LEI dans le graphe construit partiellement et pour toutes les combinaisons possibles de valeurs des entrées de contrôle Xc_i, on détermine la liste d'états Xc_i-successeur de la liste d'états traitée et la liste de vecteurs de valeurs ce qui constitue un nœud. Si un tel nœud n'existe pas déjà dans le graphe, on l'ajoute au graphe comme successeur du nœud traité.
5) On répète l'étape 4 jusqu'à l'arrivée à un nœud dont la liste de vecteurs de valeurs est une LVVII (liste de vecteurs de valeurs indépendantes et identiques).
6) On cherche la séquence, qui mène à travers le graphe de la liste initiale d'états à un nœud dont la liste d'états est une LEI et la liste de vecteurs de valeurs est une LVVII, qui est une séquence de synchronisation généralisée.

b) Algorithme 2

L'algorithme suivant précise les étapes nécessaires pour obtenir une séquence de positionnement (respec. de distinction) généralisée :

1) Création d'un nœud auquel sont associées la liste initiale d'états, une liste d'indices identiques pour tous les états et la liste de vecteurs de valeurs initiales.

2) Pour chaque nœud dans le graphe construit partiellement et pour toutes les combinaisons possibles de valeurs des entrées de contrôle Xc_I, on détermine la liste d'états Xc_I-successeur de la liste d'états traitée, la liste d'indices (où deux états ont le même indice quand les deux vecteurs de sortie obtenus sont distincts indépendamment des valeurs initiales des variables internes du circuit) et la liste de vecteurs de valeurs associée ce qui constitue un nœud. Si un tel nœud n'existe pas déjà dans le graphe, (et pour le cas d'une séquence de distinction généralisée : si la liste d'états obtenue n'est pas caractérisée par une convergence) on l'ajoute au graphe comme successeur du nœud traité.
3) On répète l'étape 2 jusqu'à l'arrivée à un nœud dont la liste d'états est une LEH ou LET et la liste de vecteurs de valeurs associée est une LVVI pour le cas d'une séquence de positionnement généralisée (respec. on répète l'étape 2 jusqu'à l'arrivée à un nœud dont la liste d'états est une LEH pour le cas d'une séquence de distinction généralisée).
4) On détermine la séquence qui mène à travers le graphe de la liste initiale d'états au nœud dont la liste d'états est une LEH ou LET et la liste de vecteurs de valeurs associée est une LVVI, qui est une séquence de positionnement généralisée (respec. on détermine la séquence qui mène à travers le graphe de la liste initiale d'états au nœud dont la liste d'états est une LEH, qui est une séquence de distinction généralisée).

Chaque programme est destiné à trouver une séquence élémentaire généralisée (synchronisation, positionnement, distinction) où, après une phase de communication avec l'utilisateur pour déterminer les paramètres de l'exemple traité (nombre des états, nombre des variables internes, nombre des entrées booléennes et nombre des sorties), le programme calcule les listes associées à chaque nœud et compare ces listes avec celles qui existent déjà dans l'arbre.

Pour cela, il génère les valeurs des entrées de contrôle et calcule les informations de chaque nœud en appelant le programme PASCAL obtenu après la compilation d'une source LUSTRE (qui exprime un automate d'états finis) modifié sous forme d'une procédure, figure 35. Puis, il vérifie s'il s'agit d'un succès ou non. Ce logiciel a été appliqué à un ensemble d'exemples avec succès.

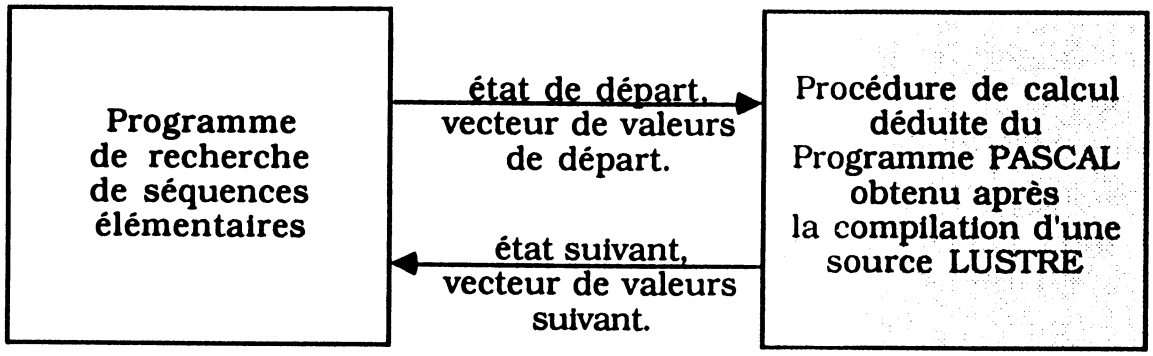


Figure 35 : Organigramme du logiciel d'aide à la génération des séquences élémentaires généralisées

Le temps de calcul augmente considérablement avec le nombre d'états "ne" et le nombre de fils de chaque nœud "nf" ; celui-ci augmente en général exponentiellement avec le nombre de variables testées par la P. C. :

$$nf \leq 2^{(n+m)}$$

où n est le nombre de variables de contrôle ;
m est le nombre des compte rendus venant de la PO ;
2^{n+m} est le nombre de toutes les combinaisons possibles des variables traitées par la P.C.

La figure 36 donne quelques temps de calcul pour des circuits simples. Ces temps sont de l'ordre de quelques secondes en cas de succès, la reconnaissance d'un échec (pas de séquence de synchronisation, exemple 2) étant plus longue. Il faut noter que, lorsqu'on ne considère que la partie contrôle du circuit (c'est à dire quand on génère les séquences classiques d'identification d'automate), le temps de calcul est du même ordre que pour le circuit global.

La recherche du test par identification d'un automate est certes coûteuse en temps CPU. Cependant traiter le test de cet automate à travers la partie opérative qu'il contrôle, et donc générer un test réaliste, ne semble pas modifier outre mesure le temps calcul nécessaire.

Un tel logiciel, après optimisation, pourrait permettre de traiter des cas de moyenne complexité (10-20 états). Notons qu'il s'agit d'un logiciel d'expérimentation ; le calcul symbolique n'a pas été utilisé. Pour distinguer

une variable non encore initialisée (dépendante d'une valeur initiale) on initialise le vecteur de valeurs initiales avec des valeurs assez grandes (2^{20}) et on fournit aux entrées de données des petites valeurs. Cette technique simple a permis de valider ce type de méthode, mais une implantation du calcul symbolique serait nécessaire pour une utilisation réelle.

exemple		synchronisation		positionnement		distinction	
		PC+PO	PC	PC+PO	PC	PC+PO	PC
exemple 1 ne=3 nf= 4	tps (s)	3.53	1.93	10.19	2.59	3.98	2.47
	ls	3	1	3	1	2	1
	nbs	8	1	12	3	2	2
exemple 2 ne=4 nf=2	tps (s)	24.41	2.50	6.46	2.80	2.75	2.45
	ls	échec	échec	4	2	2	1
	nbs			16	4	4	2
exemple 3 ne=4 nf= 4	tps (s)	2.30	1.95	3.96	2.33	2.30	1.90
	ls	2	1	2	1	échec	1
	nbs	2	1	4	1		1
exemple 4 ne=7 nf= 4	tps (s)	735	431	155	3.17	2.11	1.16
	ls	7	5	7	3	4	2
	nbs	4	4	32	32	32	16

tps : temps CPU, ls : longueur des séquences, nbs : nombre des séquences possibles trouvées.

Figure 36 : Temps CPU nécessaire pour des différents exemples

Les exemples 1, 2 et 3 sont des circuits composés d'une PC et d'une PO et peu complexes (l'exemple 1 est l'exemple détaillé dans le paragraphe II. 1. 1). L'exemple 4 décrit un circuit de moyenne complexité (74C922 : circuit de lecture de clavier 16 touches).

conclusion

On a défini une méthode de test par identification de la partie contrôle d'un circuit à travers sa partie opérative, qui s'applique au test des automates générés à partir d'une description d'un circuit par le langage LUSTRE.

Cette méthode est une extension des méthodes classiques de test d'automates, généralisée pour être applicable aux circuits complexes composés d'une partie contrôle et une partie opérative en considérant que seules les entrées/sorties primaires sont accessibles pour le test. Cette méthode a nécessité la généralisation de la définition d'un automate ; cet automate généralisé est défini par un heptuplet.

Pour établir les séquences de test, on a donné la définition des séquences élémentaires généralisées (synchronisation, positionnement, distinction) en définissant la liste de vecteurs de valeurs, la liste de vecteurs de sortie et la liste d'indice nécessaires pour l'algorithme de recherche de ces séquences. Ces séquences sont déterminées en construisant un arbre de successeurs généralisé qui sera spécifique suivant la séquence recherchée et en utilisant les propriétés introduites qui précisent les conditions nécessaires et suffisantes pour qu'une séquence de vecteurs d'entrée soit la séquence recherchée.

Enfin, un logiciel qui permet de déterminer ces séquences élémentaires généralisées a été réalisé et implémenté. Ce logiciel a été appliqué à un ensemble d'exemples simples avec succès. On a pu constater que traiter le test de l'automate à travers la partie opérative qu'il contrôle, et donc générer un test réaliste du circuit, nécessite un temps CPU du même ordre que celui nécessaire pour le test par identification de l'automate seulement.

CONCLUSION ET PERSPECTIVES

Le test des circuits intégrés est un problème de plus en plus difficile à résoudre à cause de la complexité croissante de circuits conçus et l'inefficacité des méthodes existantes de génération de test.

Pour aborder ce problème, outre la conception de circuits facilement testables, la génération du test des circuits à partir d'une description de haut niveau est une voie intéressante. C'est dans ce dernier domaine que se situe le travail présenté ici.

Dans la première partie de cette thèse, nous avons proposé l'utilisation du langage déclaratif synchrone LUSTRE en tant que langage de description de circuits. Le parallélisme implicite du langage LUSTRE, conséquence de sa nature "data-flow", est parfaitement adapté au problème de description de circuits ; de plus son aspect synchrone permet de modéliser à haut niveau les aspects temporels de leurs fonctionnements. Dans la présente étude, les différentes caractéristiques de ce langage ont été analysées tout en mettant en évidence les extensions à effectuer.

Une version pour la description du matériel du langage LUSTRE est actuellement en cours de réalisation. Cette version a comme but de résoudre les problèmes rencontrés lors de l'utilisation de ce langage dans le domaine de description des circuits.

Dans la deuxième partie de cette thèse, une étude d'aide à la génération de test a été faite à partir d'une description LUSTRE. Pour la génération de test, l'utilisation d'un langage de type "data-flow" offre une propriété très intéressante qui est la correspondance directe entre l'occurrence unique d'une variable dans la description et le bloc matériel correspondant.

L'étude présentée concerne une extension du système d'évaluation de testabilité SATAN pour les ASICs décrits dans le langage LUSTRE. Nous avons proposé d'effectuer une traduction automatique et directe des différentes primitives du langage LUSTRE en graphe de testabilité SATAN pour l'évaluation de la testabilité d'un ASIC.

A partir du graphe de testabilité SATAN obtenu, le paramètre du temps est pris en compte et on obtient ainsi un graphe de testabilité temporisé. Cela permet de déterminer les différents instants d'activation du circuit. Un logiciel a été réalisé et les résultats obtenus montrent que cette méthode offre un bon moyen pour la génération de test fonctionnel de tels circuits.

Dans la troisième partie de cette thèse, une méthode de test pour les circuits composés d'une partie contrôle centralisée et d'une partie opérative est proposée. Cette méthode est une extension des méthodes de test d'automates par identification ("checking experiments"), appliquées à des circuits décrits dans le langage LUSTRE : test de la partie contrôle d'un circuit à travers sa partie opérative. Les définitions des différentes séquences de test élémentaires nécessaires à la construction de celui-ci ont été étendues et des algorithmes de recherche de ces séquences ont été définis. Cette méthode a été appliquée aux automates générés à partir d'une description LUSTRE d'un circuit ; un logiciel prototype de détermination de ces séquences a été réalisé et appliqué à un ensemble d'exemples de faible complexité. Ce logiciel a permis de valider cette méthode, mais l'implantation de calcul symbolique serait nécessaire pour obtenir un logiciel plus efficace.

Outre le développement d'une version orientée vers la description de circuits de LUSTRE, la voie de recherche la plus prometteuse semble être l'analyse de testabilité de circuits décrits dans un langage "data-flow".

Dans cette thèse le langage LUSTRE a servi de support à ce travail ; ce langage a été choisi pour deux raisons : la première est la maîtrise de ce langage parce que ce langage a une sémantique simple et une syntaxe réduite permettant de faciliter l'analyse d'une description. La deuxième est la disponibilité de ce langage et de ses formes intermédiaires dès le début de ce travail.

Une étude complémentaire à celle présentée ici est en cours : il s'agit d'étudier la transposition des algorithmes d'analyse de testabilité à une description VHDL, en se limitant au début à des descriptions de modules de types "data-flow". L'étude du langage VHDL est évidemment motivée par l'intérêt industriel de ce langage ; des outils d'analyse de génération de test à partir de description VHDL auraient un large domaine d'application.

BIBLIOGRAPHIE

CHAPITRE 1 :

[ARM 88] : J.R. ARMSTRONG
"Chip-level modeling with VHDL"
Prentice Hall, New Jersey, 1988.

[ASH 85] : E.A. ASHCROFT, W.W. WADGE
"LUCID, the data-flow programming language"
Academic Press, 85.

[AYL 86] : J.H. AYLOR, R. WAXMAN et C. SCARRATT
"VHDL-Feature Description and Analysis"
IEEE Design & Test, Avril 1986.

[BAR 75] : M.R. BARBACCI
"A comparison of Register Transfer Languages for Describing Computers and Digital Systems"
IEEE Trans. on Computers, Fev 1975.

[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.

[BEL 84] : C. BELLON, G. SAUCIER
"CADOC : a system for computer aided functional test"
Int. Test Conference (ITC), Philadelphie, USA, Octobre 84.

[BER 86-a] : J.L. BERGERAND
"LUSTRE : Un langage déclaratif pour le temps réel"
Thèse, INPG, Grenoble, Janvier 86.

[BER 86-b] : J.L. BERGERAND, P. CASPI, N. HALBWACHS,J.A. PLAICE
"Automatic control systems programming using a real-time declarative language".
IFAC/IFIP Symp. SOCOCO 86, Graz, Mai 86.

[BOR 81] : D. BORRIONE
"Langages de description de systèmes logiques"
Thèse d'état, INPG, Grenoble, Juillet 1981.

[BOR 85] : D. BORRIONE, C. LE FAOU
"Overview of the CASCADE multi-level hardware description language and its mixed-mode simulation mechanisms".
Proc. of CHDL 85, Tokyo, Août 1985.

[CAS 87] : P. CASPI, D. PILAUD, N. HALBWACHS, J.A. PLAICE
"Le langage LUSTRE et sa sémantique opérationnelle"
Actes du colloque C3, Angoulème, Mai 87.

[CHU 65] : Y. CHU
"An algol-like computer design langage"
Comm. of the ACM, Vol 8, Octobre 1965.

[COR 81] : W.E. CORY, W.M. VAN CLEEMPUT
"Symbolic simulation for functional verification using ADLIB and SDL"
18th DAC, Nashville, USA, Juin 1981.

[COU 87] : P. COURONNE, J.B. SAINT, J.A. PLAICE
"The LUSTRE-ESTEREL portable format"
Rapport interne, Mai 87.

[CRA 85] : M. CRASTES DE PAULET
"Spécification et simulation fonctionnelles de circuits complexes"
Thèse, INPG, Grenoble, Novembre 85.

[HAL 86] : N. HALBWACHS, A. LONCHAMPT, D. PILAUD
"Describing and designing circuits by means of a synchronous declarative language".
IFIP Working Conference Frome HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble, Septembre 86.

[HIL 79] : D.D. HILL, W. VAN CLEEMPUT
"SABLE : a tool for generation structured, multi-level simulations"
Proc. of the 17th DAC, Mineapolis, Juin 1979.

[HIL 80] : D.D. HILL
"Language and environment for multi-level simulation"
PhD Thesis. Standford University; Mars 1980.

[LIP 86] : R. LIPSETT et M. SHAHDAD
"VHDL-The Language"
IEEE Design & Test, Avril 1986.

[LIS 89] : J.S. LIS et D.D. GAJSKI
"VHDL synthesis using structured modeling"
26th, ACM/IEEE Design Automation Conference, Las Vegas, Juin 1989.

[LON 86] : A. LONCHAMPT
"LUSTRE: Langage de conception de circuit"
Rapport de DEA d'Informatique, Grenoble, Juillet 1986.

[LOU 87] : M. LOUGHZAIL
"VHDL : Tests, Performance measurements and Guidelines"
Comm. Interne, Laboratoire de VLSI, Université de Montréal, Novembre 1987.

[MAH 88] : M. MAHROUS et C. BELLON
"Test generation from a circuit high-level description using the data-flow language LUSTRE"
International conference on microelectronic, (ICM 88), Algérie, Novembre 1988.

[MAH 89] : M. MAHROUS et C. BELLON
"Declarative versus imperative description languages for functional fault-modeling and test generation of VLSI circuits"

COMP EURO 89, Hambourg, RFA, Mai 1989.

[MAR 83] : S. MARINE, F. ANCEAU, K. JAHIDI
"IRENE : un langage pour la description de circuits intégrés logiques"
IMAG, Rapport de recherche N° 356, Grenoble, Mars 83.

[MAR 86] : S. MARINE
"IRENE : un langage pour la description, simulation et synthèse automatique du matériel VLSI".
Thèse, INPG, Grenoble, Février 86.

[MEN 89] : R. MENDES DA COSTA
"System VHDL-1076 : A new age in electronic design automation". VHDL users' group meeting, Washington, DC, Juin 1989.

[MIL 84] : R.E. MILNE
"A tutorial for LTS"
Internal Technical Memorandum, Standard Telecommunication Laboratories, Harlow, UK, janvier 1984.

[MUN 83] : T. MUNTEAN
"Introduction à OCCAM : langage parallèle issu de CSP pour la programmation des systèmes de transinateurs".
IMAG/LGI, RR N° 430, Décembre 1983.

[NAS 86] : J.D. NASH et L.F. SAUNDERS
"VHDL Critique"
IEEE Design & Test, Avril 1986.

[PLT 85] : R. PILOTY et al.
"The CONLAN project, concepts, implementations and applications"
IEEE Computer, Février 1985.

[PIL 86] : D. PILAUD
"Utilisation de la logique temporelle pour la validation de programmes LUSTRE". Rapport technique, Grenoble, 1986.

[PLA 87] : J.A. PLAICE, N. HALBWACHS
"LUSTRE-V2, Users Guide and reference manual, Juin 87.

[PLA 88] : J.A. PLAICE
"Sémantique et compilation de LUSTRE, un langage déclaratif synchrone".
Thèse, INPG, Grenoble, Mai 88.

[SAU 89] : L. SAUNDERS
"IEEE Design Automation Standards Subcommittee"
automation". VHDL users' group meeting, Washington, DC, Juin 1989.

[SHA 85] : M. SHAHDAD et al.
"VHSIC Hardware Description Language"
IEEE Computer, Fev 1985.

[SMI 89] : S.P. SMITH et J. LARSON
"A high performance VHDL simulator with integrated switch and primitive modeling". Proc. of the 9th Symposium on Computer Hardware Description Language and their Applications, Washington, DC, USA, Juin 1989.

[THO 81] : D.E. THOMAS
"The automatic synthesis of digital systems".
Proc. of the IEEE, Vol 69, N° 10, Octobre 1981.

[WAX 86] : R. WAXMAN
"Hardware Design Language for Computer Design and Test"
IEEE Computer, Avril 1986.

CHAPITRE 2 :

[AKE 78] : S.B. AKERS
"Functional testing with binary decision diagrams". International Fault-Tolerant Computing Symposium, Toulouse, Juin 1978.

[ARM 86] : J.R. ARMSTRONG, D.S. BARCLAY
"A heuristic chip-level test generation algorithm"
Design Automation Conference, Las Vegas, Juin 1986.

[ARM 88] : J.R. ARMSTRONG
"Chip-level modeling with HDLs". IEEE Design & Test of Computers, vol. 5, N° 1, Février 1988.

[BEL 84] : C. BELLON
"Le test fonctionnel de circuits intégrés complexes"
Thèse d'état, INPG, Grenoble, Octobre 84.

[BRE 80] : M.A. BREUER et A.D. FRIEDMAN
"Functional level primitives in test generation"
IEEE Trans. on Comp., Vol. C-29, N° 3, Mars 1980.

[BRE 85] : M.A. BREUER et M.S. ABADIR
"A knowledge-based system for designing testable VLSI chips"
IEEE Design & Test, Aout 1985.

[CRA 89] : M. CRASTES DE PAULET, M. KARAM, G. SAUCIER
"Système à base de règles pour la génération assistée de programme de test de cartes"
Comm. interne, Grenoble, 1989.

[DAV 76] : R. DAVID, G. BLANCHET
"About random fault detection of combinational networks"
IEEE Trans. on Computers, Juin 1976.

[FUJ 83] : H. FUJIWARA et T. SHIMONO
"On the acceleration of test generation algorithms"
IEEE Trans. on Comp., Décembre 1983.

[GOE 81] : P. GOEL
"An implicite enumeration algorithm to generate tests for combinational logic circuits".
IEEE Trans. on Computers, vol. C-30, Mars 1981.

[GOU 85] : J.C. GOURDON
"La réalité quotidienne du test des VLSI"
Journée SEE, Octobre 1985.

[KOH 78] : Z. KOHAVI
"Switching and finite automate theory"
Mc. Graw-Hill Book, New-York, 78.

[LAI 83] : K.W. LAI, D.P. SIEWIOREK
"Functional testing of digital systems"
Design Automation Conference, Miami Beach, Juin 1983.

[LEV 82] : Y.H. LEVENDEL, P.R. MENON
"Test generation algorithms for computer hardware description language".
IEEE tans. on Computers, vol. C-31, Juillet 1982.

[LEV 83] : Y.H. LEVENDEL, P.R. MENON
"*-algorithm : critical traces for functions and CHDL constructs"
Fault Tolerant Computing Symposium, Milan, Juin 1983.

[LIN 84] : T. LIN, S.Y.H. SU
"Functional test generation of digital LSI/VLSI systems using machine symbolic execution technique"
IEEE International Test Conference, Octobre 1984.

[LIN 85-a] : T. LIN, S.Y.H. SU
"VLSI functional test pattern generation-a design and implementation". IEEE International Test Conference, Philadelphia, Novembre 1985.

[LIN 85-b] : T. LIN, S.Y. SU
"The S-algorithm : a promising solution for systematic functional test generation". IEEE Trans. on Computer-Aided Design, vol. CAD-4, Juillet 1985.

[NOR 89] : F.E. NORROD
"An automatic test generation algorithm from hardware description language". 26th ACM/IEEE Design Automation Conference, Las Vegas, Juin 1989.

[ONE 89] : M.D. O'NEILL, D.D. JANI, C.H. CHO, J. R. ARMSTRONG
"BTG : A behavioral test generator". Computer hardware description language and their applications. Proc. of the Ninth IFIP Symposium, Washington DC, Juin 1989.

[POA 63] : J.F. POAGE
"The derivation of optimum tests for logic circuits"
Ph.D Thesis, Princeton University, 1963.

[RAR 85] : J. RARIVOMANANA
"Système CADOC : Génération fonctionnelle de test pour les circuits complexes"
Thèse, INPG, Grenoble, Novembre 85.

[ROB 80] : C. ROBACH, G. SAUCIER
"Microprocessor functional testing". IEEE International Test Conference, Philadelphia, Novembre 1980.

[ROB 85] : C. ROBACH, P. MALECHA, G; MICHEL
"Un système d'aide à l'analyse de testabilité : CATA"
Technique et Science Informatique (TSI), Vol. 4, N° 3, 1985.

[ROB 88] : C. ROBACH
"Choix et utilité des outils d'évaluation de testabilité"
Technique et Science Informatique (TSI), Vol. 7, N° 2, 1988.

[ROT 66] : J.P. ROTH
"Diagnosis of automata failures : a calculs and a method". IBM Journal of Research and Development, Vol. 10, Juillet 1966.

[SAU 83] : G. SAUCIER et C. ROBACH
"Le test et la testabilité des circuits intégrés"
Journées d'Electronique de Lausanne, Lausanne, Octobre 1983.

[SU 81] : S.Y.H. SU, Y. HSIEH
"Testing functional faults in digital systems described by register transfer language". IEEE International Test Conference, Philadelphia, Octobre 1981.

[SU 84] : S.Y.H. SU, T. LIN
"Functional testing techniques for digital LSI/VLSI system
21th Design Automation Conference, New Mexico, Juin 1984.

[THA 80] : S.M. THATTE, J.A. ABRAHAM
"Test generation for microprocessors"
IEEE Trans. on Computers, Vol. C-29, Juin 1980.

[WOD 89] : P. WODEY, C. ROBACH
"linking design and test tools : an implementation"
IEEE Trans. on Industrial Electronics, Vol.36, n°2, Mai 1989.

CHAPITRE 3 :

[BEL 84] : C. BELLON
"Le test fonctionnel de circuits intégrés complexes"
Thèse d'état, INPG, Grenoble, Octobre 84.

[FRI 71] : A.D. FRIEDMAN, P.R. MENON
"Fault detection in digital circuits"
Computer applications series. Prentice-Hall, New Jersey, 71.

[HSI 71] : E. HSIEH
"checking experiments for sequential machines"
IEEE Trans. on Computers, vol. C-20, No. 10, Octobre 71.

[KOH 78] : Z. KOHAVI
"Switching and finite automate theory"
Mc. Graw-Hill Book, New-York, 78.

[LIN 80] : G.K. LIN
"An experimental identification method for sequential machines". PhD Thesis, Illinois Institute of Technology, Chicago, Illinois, Mai 80.

[LIN 83] : G.K. LIN, P.R. MENON
"Totally preset checking experiments for sequential machines".
IEEE Trans. on Computers, vol. C-32, No. 2, Février 83.

[MAH 90] : M. MAHROUS et C. BELLON
"Identification test method applied to the complex circuits - test of the control part through the operative part"
International conference on microelectronic, (ICM 90), A paraître en Octobre 1990.

[POA 63] : J.F. POAGE
"The derivation of optimum tests for logic circuits"
Ph.D Thesis, Princeton University, 1963.

[SAU 78] : G. SAUCIER, C. ROBACH
"Dynamic testing of control units"
IEEE Trans. on Computers, vol. C-27, Juillet 78.

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de

- Monsieur LANDRAULT Christian
- Monsieur BORRIONE Dominique

Monsieur AL MAHROUSE Mazen

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité "Microélectronique"

Fait à Grenoble, le 20 Juin 1990

Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
P. VENNEREAU