Génération de test de circuits intégrés fondée sur des modèles fonctionnels

Margot Karam

THESE

présentée par

Margot KARAM

pour obtenir le titre de DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(arrêté ministériel du 23 novembre 1988)

(Spécialité: Signal - Image - Parole)

Génération de test de circuits intégrés fondée sur des modèles fonctionnels

---

Date de soutenance : 23 octobre 1991

Composition du jury :

Monsieur	Jacques MOSSIERE	Président
Messieurs	Alain COSTES	Examinateur
	Michel CRASTES	Examinateur
	Christian JAY	Examinateur
Madame	Gabrièle SAUCIER	Examinateur
Messieurs	Christian LANDRAULT	Rapporteur
	Bernard COURTOIS	Rapporteur

Thèse préparée au sein du Laboratoire Conception de Systèmes Intégrés

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

GRENOBLE 1
UNIVERSITE
JOSEPH FOURIER
SCIENCES. TECHNOLOGIE. MEDECINE

Président de l'Université :
M. NEMOZ Alain

Année universitaire 1988-1990

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GÉOGRAPHIE

PROFESSEURS de 1ère Classe

ADIBA Michel	Informatique
ANTOINE Pierre	Géologie I.R.I.G.M.
ARNAUD Paul	Chimie Organique
ARVIEU Robert	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S.
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARJON Robert	Physique Nucléaire I.S.N.
BARNOUD Fernand	Biochimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CASTAING Bernard	Physique
CAUQUIS Georges	Chimie Organique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FINKE Gerd	Informatique
GAGNAIRE Didier	Chimie Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GERMAIN Jean-Pierre	Mécanique
GIDON Maurice	Géologie
GUITTON Jacques	Chimie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques Pures

JOSELEAU Jean-Paul	Biochimie
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées
LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
LONGEQUEUE Nicole	Physique
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PANNETIER Jean	Chimie
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIER Guy	Géophysique
PIERRE Jean-Louis	Chimie Organique
RENARD Michel	Thermodynamique
RIEDTMANN Christine	Mathématiques
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ARMAND Gilbert	Géographie
ATTANE Pierre	Mécanique
BARET Paul	Chimie
BERTIN José	Mathématiques
BLANCHI J. Pierre	STAPS
BLOCK Marc	Biologie
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BORRIONE Dominique	Automatique Informatique
BOUVET Jean	Biologie
BROSSARD Jean	Mathématiques
BRUANDET Jean-François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
CHOLLET Jean-Pierre	Mécanique
COLOMBEAU Jean-François	Mathématiques (ENSL)
COURT Jean	Chimie
CUNIN Pierre-Yves	Informatique
DAVID Jean	Géographie

DHOUAILLY Danielle	Biologie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
HERINO Roland	Physique
JARDON Pierre	Chimie
KERCKHOVE Claude	Géologie
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
MOREL Alain	Géographie
NEMOZ Alain	Thermodynamique CNRS - CRTBT
NGUYEN HUY Xuong	Informatique
OUDET Bruno	Mathématiques Appliquées
PAUTOU Guy	Biologie
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PELLETIER Guy	Astrophysique
PERRIN Claude	Sciences Nucléaires I.S.N.
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
REGNARD Jean-René	Physique
RICHARD Jean-Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
ROBERT Danielle	Chimie
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VINCENT Gilbert	Physique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L'IUT 1

PROFESSEURS de 1ère Classe

BUISSON Roger	Physique IUT 1
CHEHIKIAN Alain	E.E.A. IUT 1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1
NOUGARET Marcel	Automatique IUT 1
PERARD Jacques	E.E.A. IUT 1

PROFESSEURS de 2ème Classe

BEE Marc	Physique IUT 1
BOUTHINON Michel	E.E.A. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHENAVAS Jean	Physique IUT 1

CHILO Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
FOSTER Panayotis	Chimie IUT 1
GOSSE Jean-Pierre	E.E.A. IUT 1
GROS Yves	Physique IUT 1
HAMAR Roger	Chimie IUT 1
KUHN Gérard, (détaché)	Physique IUT 1
LEVIEL Jean-Louis	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	E.E.A. IUT 1
PERRAUD Robert	Chimie IUT 1
PIERRE Gérard	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
TURGEMAN Sylvain	Génie Civil
VINCENDON Marc	Chimie IUT 1
ZIGONE Michel	Physique IUT 1

PROFESSEURS DE PHARMACIE

AGNIUS-DELORD Claudine	Physique	Faculté La Tronche
ALARY Josette	Chimie Analytique	Faculté La Tronche
BERIEL Hélène	Physiologie et Pharmacologie	Faculté La Tronche
CUSSAC Max	Chimie Thérapeutique	Faculté La Tronche
DEMENGE Pierre	Pharmacodynamie	Faculté La Tronche
FAVIER Alain	Biochimie	C.H.R.G.
JEANNIN Charles	Pharmacie Galénique	Faculté Meylan
LATURAZE Jean	Biochimie	Faculté La Tronche
LUU DUC Cuong	Chimie Générale	Faculté La Tronche
MARIOTTE Anne-Marie	Pharmacognosie	Faculté La Tronche
MARZIN Daniel	Toxicologie	Faculté Meylan
RENAUDET Jacqueline	Bactériologie	Faculté La Tronche
ROCHAT Jacques	Hygiène et Hydrologie	Faculté La Tronche
SEIGLE-MURANDI Françoise	Botanique et Cryptogamie	Faculté Meylan
VERAIN Alice	Pharmacie Galénique	Faculté Meylan

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS Classe Exceptionnelle et 1ère Classe

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puériculture	C.H.R.G.
BEREZ Henri	Orthopédie-Traumatologie	Hôpital Sud
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
	Chirurgie Générale et Digestive	C.H.R.G.
BUTEL Jean	Orthopédie-Traumatologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHAMPETIER Jean	Anatomie Topographique et Appliquée	C.H.R.G.
CHARACHON Robert	O.R.L.	C.H.R.G.
COLOMB Maurice	Immunologie	Hôpital Sud
COUDERC Pierre	Anatomie Pathologique	C.H.R.G.
DELORMAS Pierre	Pneumophtisiologie	C.H.R.G.
DENIS Bernard	Cardiologie	C.H.R.G.
GAVEND Michel	Pharmacologie	Faculté La Merci
HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie-Virologie	C.H.R.G.
MALINAS Yves	Gynécologie et Obstétrique	C.H.R.G.
MALLION Jean-Michel	Médecine du Travail	C.H.R.G.

MICOUD Max	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté La Merci
PARAMELLE Bernard	Pneumologie	C.H.R.G.
PERRET Jean	Neurologie	C.H.R.G.
RACHAIL Michel	Hépato-Gastro-Entérologie	C.H.R.G.
DE ROUGEMONT Jacques	Neurochirurgie	C.H.R.G.
SARRAZIN Roger	Clinique Chirurgicale	C.H.R.G.
STIEGLITZ Paul	Anestésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté La Merci
VIGNAIS Pierre	Biochimie	Faculté La Merci

PROFESSEURS 2ème Classe

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim-Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hôpital Sud
BERNARD Pierre	Gynécologie Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	Abidjan
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hôpital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et informatique médicale	Faculté La Merci
DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.
MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie Obstétrique	Hôpital Sud
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

"... faire que ce qui est juste soit fort,
ou que ce qui est fort soit juste"

Pascal (Pensées, sect. V)

A mon père qui a encouragé mes efforts qu'il y trouve appui et réconfort,
A ma mère en témoignage de mon affection,
A Michel, Fadi, Fadia, Simon, Robert, Catherine, Elie et Jacques pour les remercier de leur soutien.

Remerciements

Je tiens tout d'abord à exprimer ma profonde reconnaissance à Mme Gabrièle Saucier, directrice du laboratoire Conception de Systèmes Intégrés, pour avoir guidé et suivi de près mes travaux de recherche, avoir été présente pour me soutenir et me conseiller et finalement consacré généreusement son temps et sa patience pour m'aider dans la rédaction de ma thèse.

Je remercie également

M. Alain COSTES, directeur du LAAS, d'avoir accepté de faire partie du jury de cette thèse,

M. Bernard COURTOIS, directeur du laboratoire TIM3, d'avoir accepté d'être rapporteur de cette thèse,

M. Michel CRASTES DE PAULET, pour ses conseils et critiques constructives,

M. Christian JAY, responsable du département CAO-test à la Société Compass, de faire partie de ce jury et pour sa collaboration technique,

M. Christian LANDRAULT, directeur de recherche CNRS au LAM, d'avoir accepté d'être rapporteur de cette thèse ainsi que pour ses remarques et suggestions,

et M. Jacques MOSSIERE, directeur de l'ENSIMAG, de me faire l'honneur de présider le jury.

Nous tenons aussi à remercier infiniment Mme Claire RUBAT DU MARC (CIME), Mme Claire NAUTS (Compass) ainsi que M. Daniel GERVAIS et M. GIROUX (GenRad) pour leur collaboration.

Je remercie enfin les membres du CSI pour l'agréable ambiance de travail et tous ceux qui, de près ou de loin, ont contribué aux travaux présentés dans cette thèse. Mes remerciements vont tout particulièrement à Régis Leveugle, Daniel Salber, Laurent Masse-Navette, Xavier Machenaud, Jean-Luc Patry, Xavier Delord, Pierre Abouzeid et toute l'équipe synthèse logique pour leur collaboration.

Résumé

Cette thèse concerne l'utilisation de modèles fonctionnels dans le test de circuits intégrés complexes. Dans la première partie, des vecteurs de test sont générés pour les automates d'états finis à partir de leurs spécifications de synthèse. Un premier ensemble de test consiste à parcourir tous les arcs du graphe de contrôle en optimisant les valeurs d'entrées non spécifiées sur les transitions afin d'accroitre la couverture. Il est montré que ce test a une excellente couverture par rapport à sa longueur. Les fautes résiduelles sont détectées par des méthodes de distinction sur des modèles (machine juste X machine fausse).
La deuxième partie est consacrée au test hiérarchisé de circuits complexes. Les vecteurs de test locaux sont justifiés vers les entrées primaires et propagés vers les sorties primaires en utilisant des variables symboliques et des modèles fonctionnels pour les blocs traversés. Des techniques originales de propagation retardée permettent de restreindre le nombre d'échecs des propagations. Un prototype en Prolog a été expérimenté.

Mots clés

Génération de test, modèle fonctionnel, automates d'états finis, produit cartésien de machines, génération hiérarchisée de test, valeurs symboliques, propagations retardées.

Abstract

This thesis focuses on the use of the functional models for test generation of complex circuits. In a first section, the test generation is performed for finite state machines using the specification used for synthesis. A first phase consists in going through all the arcs of the control graph while optimizing the values of the don't care inputs. The quality of the sequences (fault coverage/ sequence length) is shown to be high. The residual faults are detected using the (good machine X faulty machine) models.

In a second section, hierarchical test generation is addressed. Local test vectors for blocks are justified to the primary inputs and the fault effect are propagated to the primary outputs using symbolic test values and the functional models for the forward and backward propagation. An original concept of delayed propagations is proposed to reduce the propagation failures.

Key words

Test generation, functional models, finite state machine, cartesian machines product, hierarchical test generation, symbolic values, delayed propagations.

Introduction

Cette thèse a comme objectif d'étudier l'utilisation des modèles fonctionnels dans la génération de test de circuits intégrés complexes. L'objectif est de pallier l'explosion combinatoire des méthodes de génération de test fondées sur une description structurelle au niveau logique et sur des hypothèses de défauts définis à ce même niveau.

Dans une première partie on s'intéresse aux contrôleurs ou automates d'états finis. Ce type de bloc existe pratiquement dans tous les circuits dédiés et peut être relativement complexe. Il a comme particularité d'être spécifique au circuit dédié, d'être décrit au niveau comportemental et synthétisé automatiquement à partir d'une telle spécification. En effet des outils automatiques de synthèse de machines d'états finis (FSM Compiler) existent aujourd'hui pratiquement dans tous les outils élaborés de conception de circuits intégrés digitaux. Il semble donc intéressant de tirer profit de cette spécification de haut niveau pour générer des séquences de test. La méthode proposée consiste à passer par tous les arcs du graphe de la machine d'états finis, en évitant délibérément de faire des hypothèses "d'erreurs fonctionnelles" toujours très contestables, puis de mesurer la couverture de la séquence obtenue en termes de fautes considérées au niveau logique. Dans une dernière étape, les défauts ou fautes résiduelles (non couvertes) sont détectées par une méthode de distinction fondée sur les produits cartésiens de machines d'états finis juste et fausses. Ce que l'on cherche à démontrer est qu'une séquence fonctionnelle relativement "stricte", évitant des raisonnements d'identification d'automates [Hen64], [Koh78] ainsi que des hypothèses d'erreur fonctionnelles [Che90], [Red91], a un haut pouvoir de détection (mesuré par le nombre de fautes détectées par vecteurs de test) et qu'il est plus efficace de revenir ensuite à un vrai raisonnement par distinction sur les fautes résiduelles. Cette approche sera justifiée sur des résultats obtenus sur des contrôleurs-test (benchmarks internationaux).

Dans une deuxième partie, on s'intéresse à une méthode de génération de test hiérarchisée fondée sur des modèles fonctionnels des blocs composant le circuit. Un circuit est donc décrit comme une interconnexion de blocs définis à un niveau de transfert de registre (RTL). A chaque bloc est associé un ensemble de vecteurs de test et il s'agit de "justifier" ces vecteurs locaux à partir des entrées primaires et de propager l'effet des fautes vers les sorties primaires du circuit. Un modèle fonctionnel est associé à chaque bloc afin de permettre des techniques efficaces de propagation. Les aspects originaux de cette approche réside en l'utilisation de

données symboliques de test permettant de compacter les données de test ainsi qu'une technique originale de résolution de conflit fondée sur des propagations retardées.

Pour valider cette approche, un logiciel de génération hiérarchisée de test a été écrit en langage Prolog. En effet un prototypage est rapide en Prolog car les différentes heuristiques de retour arrière sont aisément implantées et expérimentées. Ce langage a également permis une modélisation fonctionnelle dense et claire. L'efficacité de l'approche a été testée sur des circuits de type chemins de données pour lesquels les signaux de contrôle sont supposés directement contrôlables.

En conclusion, il semble que pour les deux problèmes traités, la modélisation fonctionnelle a été synonyme d'efficacité tant en temps d'élaboration de programmes de test qu'en qualité des séquences obtenues.

Chapitre 1: Contexte général et terminologie

Les circuits intégrés sont testés au cours de différentes étapes. Le "test en fin de conception" met en évidence d'éventuelles erreurs de la phase de conception. Le "test de fabrication" détecte les défauts de fabrication et trie les circuits en éliminant ceux qui sont défectueux. Le troisième est le test usager; celui-ci à la réception du produit, peut tester la conformité de son circuit par rapport à son cahier de charge, grâce au "test en réception".
Dans cette thèse, nous nous intéressons au test en fin de fabrication.
En général un test d'un circuit consiste à appliquer aux entrées du circuit des vecteurs binaires appelés stimulis de test ou vecteurs d'entrée de test et comparer la réponse du circuit avec la réponse attendue (juste). Les stimulis et leur réponse forment les vecteurs de test ou séquence de vecteurs de test ou séquence de test tout court.

I Test de fin de fabrication

On distingue deux types de test de fin de fabrication à savoir le test aléatoire et le test avec vecteurs prédéterminés.

I. 1 Test aléatoire

Le test aléatoire consiste à générer des vecteurs d'entrée aléatoires, au moment du test effectif du circuit. Il suppose que l'on possède une référence et nécessite un testeur qui compare les sorties du circuit à tester avec celles du circuit de référence et détermine ainsi si le circuit est correct ou non, par rapport à cette référence.
La longueur de la séquence de test peut être approximée. Cette approximation est fondée sur la théorie probabiliste. L'objectif d'une telle approximation est d'estimer la longueur de la séquence qui permet d'avoir une faible probabilité d'accepter comme bon un circuit défectueux [Dav76], [She77], [Par75].

I. 2 Test avec des vecteurs prédéterminés

Ce test est effectué avec une séquence de test précalculée et stockée dans l'équipement de test. L'équipement de test effectue la comparaison entre les réponses du circuit sous test et les réponses attendues qui sont également stockées dans l'équipement de test.
Les vecteurs de test peuvent être générés par une méthode aléatoire ou calculés par des méthodes déterministes. Ces méthodes travaillent sur des hypothèses de fautes.

I. 2. 1 Génération aléatoire de vecteurs de test

Cette méthode consiste à générer des vecteurs aléatoirement et à sélectionner parmi les vecteurs générés ceux qui détectent les fautes référenciées dans un dictionnaire de fautes. Cette sélection peut se faire à l'aide un simulateur de fautes [Sch75b].

Plusieurs méthodes ont été proposées pour augmenter l'efficacité de cette méthode [Agr76], [Agr81], [Par76], [Lis87].

I. 2. 2 Génération déterministe de vecteurs de test

Il existe deux méthodes de génération déterministe de vecteurs à savoir la génération par distinction et la génération par identification.

Génération de test par identification

La méthode de génération des vecteurs de test par identification est une recherche de la conformité du circuit par rapport à une référence. C'est le cas par exemple de la génération des vecteurs exhaustifs des circuits combinatoires ou de la génération de test d'un contrôleur qui consiste à passer par tous les arcs du graphe des états du contrôleur.

La génération des vecteurs de test est en général indépendante des hypothèses de fautes définies à un niveau structurel. Toutefois cette séquence est évaluée en calculant le nombre de fautes qu'elle détecte. Ceci est généralement fait par un simulateur de fautes.

Génération de test par distinction

Dans la méthode par distinction, des hypothèses de fautes sont faites et l'on essaie de trouver un vecteur ou une séquence de vecteurs de test qui permettrait de distinguer le circuit juste du circuit faux pour chacune de ces fautes. Dans le cas où elle est applicable, la méthode de génération de test déterministe par distinction permet d'obtenir une séquence de test compacte et performante.

I. 2. 3 Génération mixte de vecteurs de test

La génération mixte des vecteurs de test est très utilisée. Elle consiste à générer des vecteurs de test en combinant plusieurs méthodes parmi celles qui sont présentées ci-dessus (aléatoire, par identification, par distinction).

II Génération de vecteurs prédéterminés de test de circuits

Le but de la génération de test est d'obtenir une séquence de test de "bonne" qualité avec un coût "abordable".

La qualité d'une séquence est donnée par la "couverture de fautes" qu'elle garantit. Cette couverture est le quotient du nombre des fautes détectées par la séquence sur le nombre des fautes modélisées.

Deux coûts très différents peuvent être associés à une séquence de test. Le premier est le temps de génération de la séquence. Le second est le temps nécessaire au test lui même. Ce temps est supposé proportionnel au nombre des vecteurs de test de la séquence [Agr88].

II. 1 Génération de test des circuits combinatoires

Les méthodes les plus utilisées pour la génération de test des circuits combinatoires sont des méthodes par distinction qui travaillent sur des hypothèses de fautes logiques. La génération aléatoire est moins utilisée car il a été démontré [Agr88] que cette méthode génère des vecteurs de test pour un grand nombre de fautes 65 % à 85 % mais que la génération des vecteurs de test pour les fautes "résistantes" devient très longue, voire impossible.

Les méthodes de génération de test par distinction utilisent des algorithmes du type chemin sensible tels le D-algorithme [Rot66] et Podem [Goe81]. Les modèles de fautes que l'on cherche à couvrir sont les collages à 0 et à 1 des entrées et sorties des portes logiques.

Dans [Iba75] les auteurs ont démontré que la génération des vecteurs de test d'un circuit combinatoire par des méthodes algorithmiques est un problème NP-complet. L'utilisation d'heuristiques réduit la complexité de ces méthodes qui varie alors entre n**2 et n**3 où n est le nombre de portes logiques du circuit [Goe80], [Wil82].

II. 1. 1 Notions de base dans un circuit combinatoire

Le collage à 0 (1) d'une connexion suppose que le niveau 0 (1) est permanent sur cette connexion.

Un collage à 0 est noté s-0 et un collage à 1 est noté s-1.

Manifester une faute s-0 (s-1) sur une connexion consiste à lui appliquer une valeur différente de celle que lui impose la faute. Pour manifester un collage à 0 sur une connexion, cette connexion doit être mise à 1.

Les entrées primaires qui permettent de manifester une faute sur une connexion définissent le *vecteur de manifestation de la faute.*

La valeur D (Dbar) sur une connection désigne que la valeur juste de cette connexion est 1 (0) et la fausse est 0 (1).

Un *chemin de propagation* (sensibilisation) de l'effet d'une faute est un chemin dont la sortie prend deux valeurs différentes suivant que la faute est présente ou absente. Il est calculé par un processus de propagation. Le chemin de propagation est dit chemin de *propagation simple* lorsque l'effet de la faute est propagé sur un seul chemin sinon il est dit chemin de *propagation multiple.*

Un *chemin de justification* d'une valeur d'une connexion est un chemin qui permet d'amener la valeur souhaitée depuis les entrées primaires du circuit jusqu'à cette connexion.

Le *processus d'implication* (avant) est proche de la simulation. Il consiste à calculer les conséquences logiques d'une affectation de valeurs à certaines connexions.

Une *faute redondante* est une faute indétectable.

II. 1. 2 Le D-algorithme

Etant donné une faute sur l'entrée d'un bloc le D-algorithme cherche un chemin de propagation de cette faute et justifie toutes les valeurs requises pour l'établissement de ce chemin.

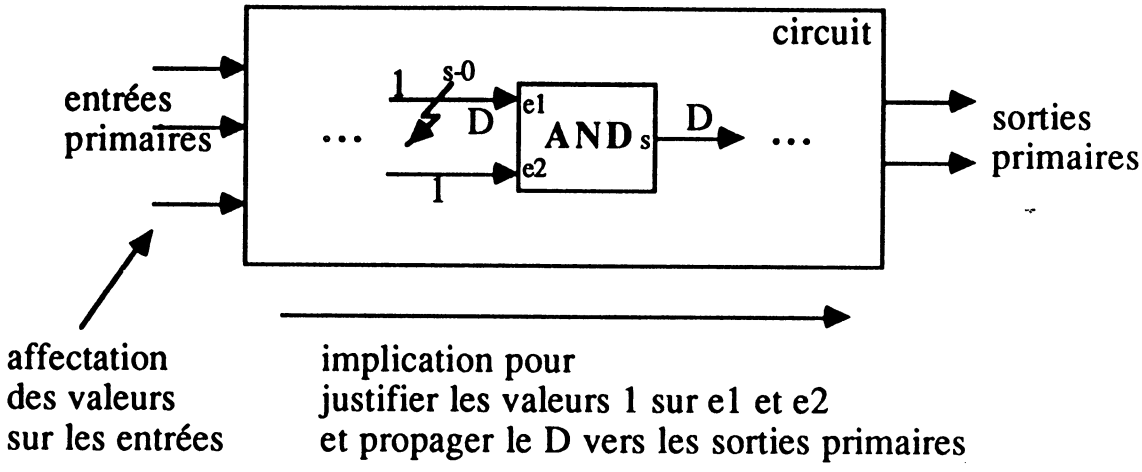


Figure 1. Propagation et justification dans le D-algorithme

Etant donné une faute sur l'entrée d'une porte, le processus de propagation consiste à propager l'effet de cette faute D (Dbar) depuis l'entrée du bloc jusqu'aux sorties primaires du circuit. Dans le cas d'un bloc à sortance multiple, la valeur du D (Dbar) est propagée sur toutes les sortances. C'est un chemin de propagation multiple (sensibilisation multiple).

Le processus de justification consiste à justifier les valeurs 0 et 1 depuis les entrées des portes jusqu'aux entrées primaires du circuit.

Le D-algorithme travaille avec 5 valeurs des signaux qui sont 0, 1, D, Dbar, X où X est la valeur indifférente et utilise les chemins de propagation simples et multiples.

Certaines méthodes de génération de test sont réduites à la recherche des chemins de propagation simple [Cha70]. Il a été démontré plus tard que certains types de problèmes, qui ont nécessité une propagation multiple dans le cadre du D-algorithme peuvent être résolus si on tient compte des deux circuits juste et faux tout au long de la propagation simple [Che88a]. Ainsi une représentation de 9 valeurs remplace celle à 5 valeurs du D-algorithme. Ces valeurs donnent la valeur juste et fausse sur une connexion (0/0, 1/1, 1/0, 0/1, X/X, 0/X, X/0, 1/X, X/1).

II. 1. 3 L'algorithme Podem

Cet algorithme cherche à affecter des valeurs aux entrées primaires du circuit qui permettent de manifester une faute et de propager son effet jusqu'aux sorties primaires du circuit. Cet algorithme ne se sert pas de la propagation arrière pour

justifier des valeurs nécessaires pour la manifestation de la faute et la propagation de son effet. La manifestation d'une faute et la propagation de son effet se font par implication uniquement. La difficulté réside dans le choix des valeurs des entrées qui permettent la manifestation de la faute et la propagation de son effet.

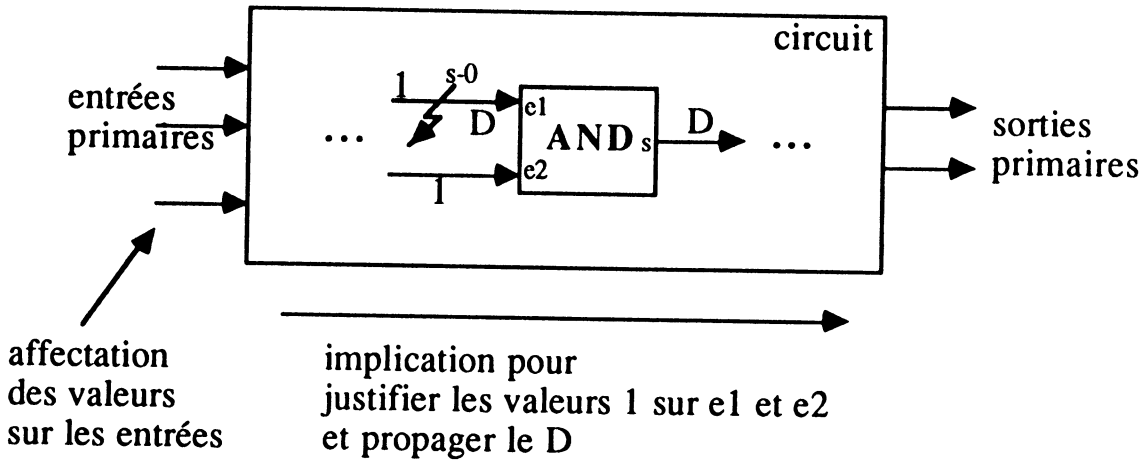


Figure 2. Affectation et implication dans Podem

Pour accélérer le choix des entrées à affecter [Goe81] propose une procédure de *trace arrière* (backtrace) qui consiste à choisir un chemin entre les entrées primaires du circuit et le bloc où se manifeste la faute.

Dans la plupart des approches proches du Podem, au cours des processus de propagation de l'effet d'une faute et de trace arrière, la recherche des chemins est guidée par des heuristiques dont les critères de choix sont la mesure de la contrôlabilité des sorties et l'observabilité des entrées des portes logiques du circuit. Cette mesure peut être calculée de différentes façons et la littérature est abondante sur ce sujet [Brg84], [Gol80], [Ben84], [Tom83], [Set86], [Sav84], [Agr85]. Une évaluation de ces mesures est donnée dans [Cha89] et [Pat86].

Podem est capable de traiter les circuits comportant des portes xor pour lesquels le D-algorithme est inefficace. De plus il est plus rapide que le D-algorithme puisqu'il réduit le nombre de retours arrière. Cette réduction est liée à l'utilisation des heuristiques de recherche de chemin. Comme le D-algorithme, Podem travaille sur 5 valeurs.

De nombreuses méthodes ont proposé des heuristiques d'amélioration de la recherche des vecteurs de test. [Fuj83], [Abr85], [Sch87], [Sch88], [Che87] proposent des heuristiques pour guider le processus d'affectation des valeurs des entrées. [Pat90] propose d'utiliser plusieurs (16) processeurs en parallèle pour diminuer le temps de la génération.

Il est intéressant de signaler qu'une version de Podem travaillant sur des valeurs symboliques a été implantée dans [Su90]. Cette méthode n'est pas pratique pour les gros circuits combinatoires vu le grand nombre de fonctions symboliques booléennes à évaluer.

II. 2 Génération de test de circuits séquentiels n'utilisant pas le modèle comportemental

Les blocs de logique séquentielle implantés dans les circuits à la demande sont essentiellement de deux types. Les premiers sont réalisés sous forme de logique séquentielle dite aléatoire. Cette logique est constituée de portions de logique combinatoire séparées par des barrières temporelles qui sont les points mémoires et les registres. Le deuxième type implante des automates d'états finis et constitue l'essentiel des séquenceurs ou contrôleurs. Dans ce deuxième cas, les outils du type compilateur de silicium permettent de réaliser cette logique automatiquement à partir d'une spécification fonctionnelle sous forme de graphe d'états. Ces outils réalisent un codage optimisé des états et implémentent cette logique sur des cellules standards ou réseaux programmables avec des points de mémorisation de différents types (JK, RS, D).

Il est reconnu que la génération de test des circuits séquentiels est une tâche difficile [Mic83], [Bou71], [Hen64], [Bre86]. C'est pourquoi plusieurs approches sont utilisées pour améliorer la testabilité de ces circuits. Elles proposent soit d'insérer des points de test par insertion de multiplexeurs [Sam89] ou de points de mémorisations [Hay73], [Hay74], [Mor89] sur les connexions, soit de monter les points de mémorisation en registre à décalage (scan path) commandable et observable à partir des plots du circuit. Ce registre peut contenir tous les points de mémorisation du circuit [Eic77], [Eic78], [Agr84], ou certains de ces points. Pour les automates d'états finis complexes il est judicieux de rendre contrôlable et observable tout le registre d'état. Dans le cas d'une logique séquentielle aléatoire, il est courant de chercher quels sont les points de mémorisation qu'il est souhaitable de monter en registre à décalage. De nombreuses études théoriques peuvent être inventoriées pour la recherche d'un ensemble optimal de ces points [Gup90]. L'outil industiel de génération de test "Intelligen" s'appuie sur des techniques d'estimation de testabilité [Gol79], [Gol80] pour définir les points de mémorisation à monter en registre à décalage et proposer un ensemble minimal de points de test complémentaires à insérer [Mor89].

Nous allons rappeler brièvement l'extension du D-algorithme au cas des automates d'états finis car ces notions nous seront utiles dans le chapitre suivant. On donnera

ensuite des considérations générales sur la complexité de la génération du test pour les circuits séquentiels.

II. 2. 1 Dérivés du D-algorithme pour les automates d'états finis

Un automate d'états finis (figure 3a) peut être représenté par une structure itérative (figure 3b) en coupant la boucle de retour à la sortie des bascules d'états. Cette représentation est formée de plusieurs blocs combinatoires aux instants successifs. Chaque bloc représente une copie de la partie combinatoire et des bascules du circuit séquentiel.

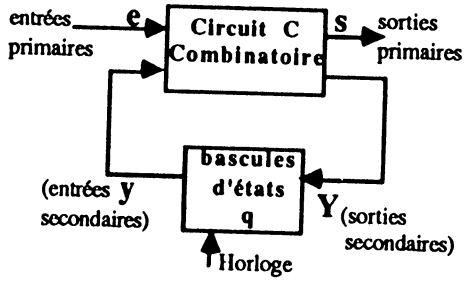


Figure 3a. Circuit séquentiel synchrone

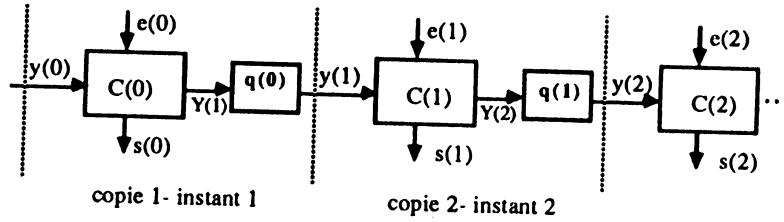


Figure 3b. Circuit itératif équivalent

Etant donné une faute dans le circuit combinatoire C(i), la génération du vecteur ou de la séquence de test est effectuée en deux pas.

pas1. propager l'effet de cette faute sur les sorties s(i) et Y(i+1) du circuit combinatoire C(i).

Si la faute est manifestée sur s(i) alors la phase de propagation a réussi et on passe au pas suivant.

Si la faute est manifestée sur Y(i+1) alors on passe à l'instant suivant (bloc suivant). La sortie y(i+1) des bascules devient égale à Y(i+1) et on recommence le pas1 pour essayer de propager la faute sur la sortie du circuit C(i+1).

pas2. justifier la valeur qui manifeste la faute par une propagation arrière à travers C(i) jusqu'aux entrées e(i) et y(i).

Si y(i) est l'état d'initialisation du circuit ou si la valeur de y(i) est indifférente alors le pas2 se termine avec succès. Sinon y(i) est propagé en arrière à travers la cellule C(i-1) puis C(i-2), ... jusqu'à obtenir une solution.

Remarquons que les processus de propagation et de justification dans le temps, doivent tenir compte de la présence permanente de la faute.

II. 2. 2 Problème de la complexité et heuristiques

Il est clair que les méthodes de génération de test fondées sur le D-algorithme sont explosifs en complexité et surtout dans le cas de circuits séquentiels. En effet la génération de test d'un circuit séquentiel revient à traiter un circuit combinatoire N

fois plus complexe si on utilise N copies dans le temps comme on vient de le montrer dans § II.2.1. Les circuits industriels étant en majorité de la logique séquentielle aléatoire, les études et réalisations portent sur la mise au point d'heuristiques appropriées.

Les variations dans les différentes heuristiques concernent essentiellement l'utilisation de chemins de propagation simple ou multiple et l'exhaustivité de l'exploration de l'ensemble des solutions.

Utilisation de chemins de propagation simple ou multiple

Etant donné une faute sur une connexion, un chemin simple de propagation propage l'effet de la faute suivant une seule équipotentielle et inhibe cet effet dans les différentes copies [Put71]. Ceci peut entrainer un échec lors de la justification des valeurs du chemin. C'est le cas de l'exemple de la figure 4 qui illustre une propagation de l'effet d'un collage à 0 sur la première entrée (e1) de la porte "et" à deux entrées. Dans Copie 1 la valeur D de e1 est propagée alors que dans copie 2, la faute est inhibée puisque l'entrée e1 de la porte "et" à deux entrées est mise à la valeur 0. La propagation du D à travers la porte "et" à trois entrées dans la deuxième copie nécessite la justification de la valeur 1 sur sa première entrée. Cette justification échoue puisqu'un conflit se manifeste sur la connexion collée à 0 où l'effet de la faute a été inhibé.

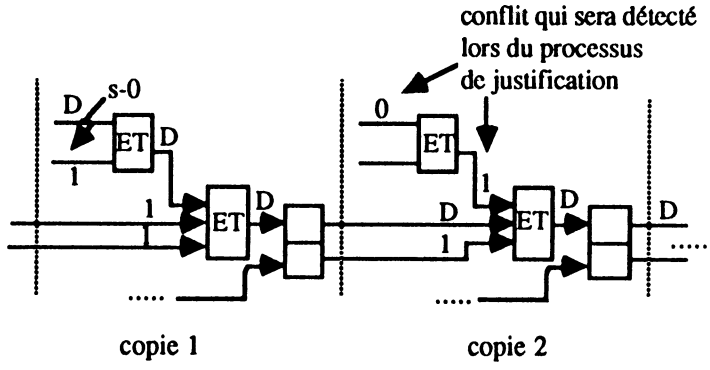


Figure 4. Inhibition de l'effet de la faute dans copie 2

Le problème que pose le chemin à propagation simple peut être résolu par l'utilisation du chemin à propagation multiple. Une solution à ce problème a été proposée dans [Mut76]. Elle consiste à appliquer le processus de propagation simple et avoir recours à la propagation multiple pendant la justification lorsque c'est nécessaire. La solution du problème posé sur la figure 4 est illustré sur les figures 5a et 5b. La figure 5a illustre une propagation simple de la valeur D sur l'entrée e1 de la porte "et". Un conflit apparaît lors de la justification de la valeur 1 sur la première entrée de la porte "et" à trois entrées. Ce conflit entraine la suspension de

la procédure de justification. La valeur D qui manifeste l'effet de la faute sur l'entrée e1 de la porte "et" à deux entrées est aussi propagée dans copie 2 (figure 5b).

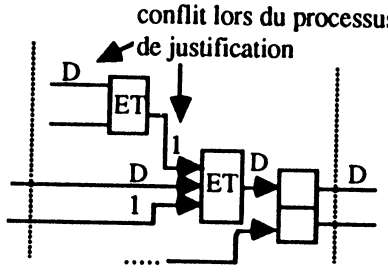


Figure 5a. Propagation simple de la valeur D

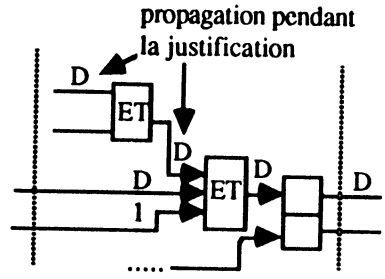


Figure 5b. Propagation multiple de D

Exhaustivité de l'exploration de l'ensemble des solutions

Les différentes méthodes existantes effectuent des retours arrières plus ou moins exhaustifs en cas d'échec. On peut chercher exhaustivement une solution dans une copie courante avant de propager vers une autre copie [Put71].

Il faut noter que tout retour arrière intensif se paie par un espace mémoire coûteux. Certains auteurs se sont penchés sur l'optimisation de cet espace mémoire [Che88b]. Dans certains outils industriels des restrictions sérieuses sur le nombre des retours arrières sont appliquées. L'exploration large de l'espace des solutions est remplacée par une technique de choix plus sophistiquée d'une solution à laquelle on se tient. Ainsi dans "Intelligen" [Mar89], [Mar78] un seul chemin de propagation est choisi à partir des mesures de la testabilité proposée dans [Gol79], [Gol80].

II. 2. 3 Autres méthodes n'utilisant pas le modèle fonctionnel ou comportemental

Ce sont des méthodes assistées par un simulateur de fautes. On distingue celles qui sont purement aléatoires [Nit85], [Sch75a], [Wil77] et celles qui sont mixtes utilisant une combinaison des approches aléatoire et déterministe classique.

[Cho91] propose une méthode de génération aléatoire détectant un premier ensemble de fautes. Pour les fautes résiduelles il utilise une approche déterministe classique (dérivé d'un D-algorithme) dans une des copies. La propagation de l'effet de ces fautes sur les différentes copies se fait ultérieurement de façon aléatoire également. La justification à travers les différentes copies se fait par une approche déterministe.

Dans [Agr87], [Agr88b], [Che87], [Che90] l'analyse des résultats de la simulation d'une séquence aléatoire permet de définir les points difficiles à contrôler et à

observer du circuit. Ces renseignements servent comme critères de choix dans la recherche des chemins de propagation des fautes dans une approche déterministe classique (dérivé de Podem).
Dans [Pat91] les renseignements sur le circuit juste sont également obtenus et stockés par simulation. Pour justifier un état permettant de manifester une faute, il explore d'abord la séquence simulée pour en extraire une séquence qui permet d'atteindre cet état dans le circuit juste. La séquence extraite est ensuite vérifiée par simulation pour s'assurer qu'elle est valide aussi dans le circuit faux. Dans le cas d'échec la justification de l'état est effectuée par un algorithme classique dérivé de Podem. Pour les fautes résistantes [Pat91] analyse également les résultats du fichier de simulation en reprenant l'étude de la copie où la propagation a avorté.

Chapitre 2: Génération de test des machines d'états finis

I Définitions, terminologie et état de l'art

I. 1 Description d'une machine d'états finis

Une machine d'états finis est définie par un ensemble fini d'états et de transitions entre les états. On distingue deux types de machines d'états finis à savoir les machines de Moore (figure 1a) et de Mealy (figure 1b). Une machine d'états finis comporte une partie combinatoire et les bascules d'états. La partie combinatoire contient la logique de calcul des sorties et des états suivants. Pour les machines de Moore les sorties dépendent uniquement de l'état courant alors que pour les machines de Mealy les sorties dépendent aussi des entrées de la machine.

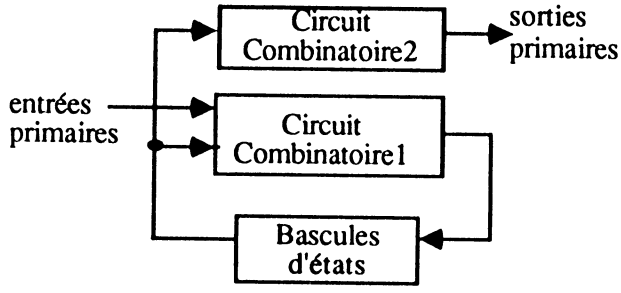


Figure 1a. Machine Moore

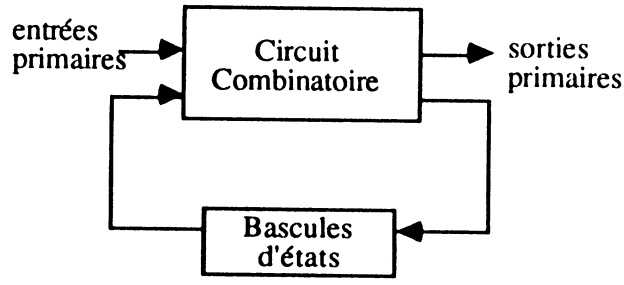


Figure 1b. Machine Mealy

Une machine d'états finis (FSM) est représentée formellement par un 6-tuplet FSM = (Σ, O, S, S0, ∂, λ) où

Σ est l'ensemble des entrées primaires de la machine
O est l'ensemble des sorties primaires de la machine
S est l'ensemble des états de la machine
S0 est l'état initial de la machine
∂ est la fonction de calcul de l'état suivant de la machine.

$$\partial : S \times \Sigma \rightarrow S$$

λ est la fonction de calcul des sorties de la machine.

$$\lambda : S \rightarrow O \text{ dans une machine de Moore}$$

$$\lambda : S \times \Sigma \rightarrow O \text{ dans une machine de Mealy}$$

Une transition est définie par 4 paramètres (entrée, état courant, état suivant, sortie). A partir d'un état courant et d'une entrée, la transition met la machine dans l'état suivant et délivre la sortie.

Une machine d'état fini peut être représentée par un graphe d'états ou par un tableau des états. Elle peut être aussi décrite dans un langage algorithmique classique de type Pascal ou C, ou donnée dans un format normalisé.

I. 1. 1 Représentation par un graphe d'états

Une machine d'états finis peut être représentée par un graphe d'états G(N, A). Dans ce graphe

- N est l'ensemble des nœuds associé bijectivement à l'ensemble des états de la machine.

- A est l'ensemble des arcs correspondant aux transitions entre les états de la machine. Un arc est étiqueté par le vecteur d'entrée de la transition. Une variable d'entrée prend l'une des 3 valeurs (0, 1, ø).

Dans le cas d'une machine Mealy les valeurs des sorties sont affectées aux arcs; dans une machine Moore elles sont affectées aux nœuds.

Une machine d'états finis à 3 états, 2 entrées et 2 sorties est représentée par un graphe d'états de Moore et un graphe d'états de Mealy sur les figures 2a et 2b respectivement.

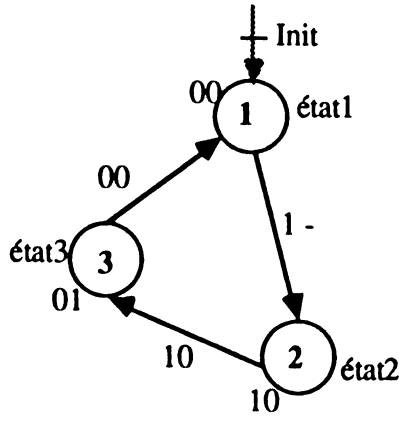


Figure 2a. Graphe d'états de la machine de Moore

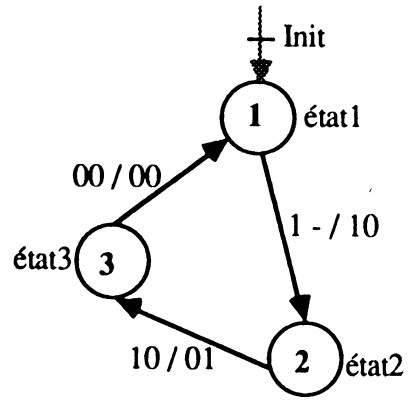


Figure 2b. Graphe d'états de la machine de Mealy

Dans cette figure les vecteurs d'entrée 1-, 10 et 00 sont associés respectivement aux arcs (1, 2), (2, 3) et (3, 1).

I. 1. 2 Représentation par un tableau des états

Le tableau des états associé à une machine d'états finis est un tableau à deux entrées (état courant, entrée). Dans chaque case est représenté le couple (état suivant, sortie).

Les tableaux des états des machines de Moore et de Mealy représentées sur les figures 2a et 2b, sont donnés sur les tableaux 1a et 1b respectivement.

entrées -> / état courant	11	10	00
1	(2, 10)	(2, 10)	-
2	-	(3, 01)	-
3	-	-	(1, 00)

Tableau 1a. Tableau des états de la machine de Moore

entrées -> / état courant	11	10	00
1	(2, 00)	(2, 00)	-
2	-	(3, 10)	-
3	-	-	(1, 01)

Tableau 1b. Tableau des états de la machine de Mealy

I. 1. 3 Format normalisé

Il s'agit de la liste des transitions de la machine donnée dans le format suivant: (valeur des entrées, état courant, état suivant, valeur des sorties). Parmi les formats standards existants on cite le format "kiss" qui est utilisé dans les échanges par réseau informatique des modèles des machines d'états finis.

Le format "kiss" de description de la machine de Mealy décrite par son graphe d'états sur la figure 2b est donné comme suit:

```
i 2;        /* nombre des entrées */
o 2;        /* nombre des sorties */
s 3;        /* nombre des états */

(1-, 1, 2, 10)
(10, 2, 3, 01)
(00, 3, 1, 00)
```

I. 2 Machine d'états finis fausse

Une machine d'états finis "fausse" est associée à une machine "juste" en présence d'une faute. Lorsqu'une faute existe dans une machine d'états finis (collage sur une connexion), elle induit un fonctionnement faux différent de celui de la machine juste. La machine fausse est représentée par le 6-tuplet $FSM_f = (\Sigma, O_f, S_f, S0, \partial_f, \lambda_f)$. L'ensemble des entrées Σ de la machine fausse est le même que celui de la machine juste. L'état d'initialisation S0 est supposé robuste par rapport aux fautes. S0 est donc l'état d'initialisation des machines juste et fausses. Les ensembles O_f et S_f peuvent être les mêmes où différents des ensembles O et S de la machine juste. ∂_f et λ_f définissent les fonctions de calcul de l'état suivant et de la sortie de la machine en présence de la faute.

Le fonctionnement de la machine fausse diffère de celui de la machine juste dans un des trois cas suivants. Les cas ci-dessous traitent une machine de Mealy toutefois ils sont aussi valables pour une machine de Moore.

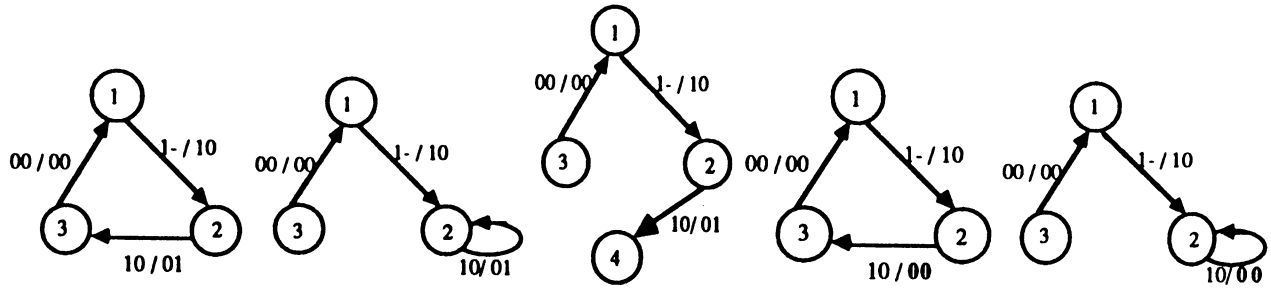
Cas1. A partir d'un état courant s et d'une valeur i aux entrées, l'état suivant de la machine fausse est différent de celui de la machine juste $\partial(s, i) \neq \partial_f(s, i)$ mais la valeur de sortie de la machine fausse est égale à celle de la machine juste $\lambda(s, i) = \lambda_f(s, i)$. Dans le cas où la machine juste n'est pas spécifiée complètement, l'état suivant de la machine fausse peut être un état qui n'appartient pas à l'ensemble S des états de la machine juste $S_f \neq S$ et $\partial_f(s, i) \notin S$.

Cas2. A partir d'un état courant s et d'une valeur i aux entrées, la valeur de sortie de la machine fausse est différente de celle de la machine juste mais l'état suivant de la machine fausse est égal à celui de la machine juste. Ceci se traduit par $\lambda(s, i) \neq \lambda_f(s, i)$ et $\partial(s, i) = \partial_f(s, i)$.

Cas3. Ce cas regroupe les deux cas précédents. Il se traduit par un état suivant et une valeur de sortie différents de ceux de la machine juste $\partial(s, i) \neq \partial_f(s, i)$ et $\lambda(s, i) \neq \lambda_f(s, i)$.

Exemple

Soit l'exemple de la machine juste M représentée par son graphe d'états sur la figure 3a. Soit M1, M2, M3 et M4 quatre machines fausses de M représentrées par leur graphe sur les figures 3b, 3c, 3d et 3e respectivement.



a. Machine juste M | b. Cas1- Machine fausse M1 | c. Cas1- Machine fausse M2 | d. Cas2- Machine fausse M3 | e. Cas3- Machine fausse M4

Figure 3. Fonctionnement de la machine juste différent de celui des 4 machines fausses

Le fonctionnement des machines M1, M2, M3 et M4 est différent de celui de la machine M. La transition juste de l'état 2 vers l'état 3 étiquetée par le couple entrée/sortie 10/01 devient une transition de l'état 2 vers l'état 2 dans le modèle de la machine fausse M1, une transition de l'état 2 vers l'état 4 dans le modèle de la machine fausse M2, une transition de l'état 2 vers l'état 3 étiquetée par le couple 10/00 dans le modèle de la machine fausse M3 et une transition de l'état 2 vers l'état 2 étiquetée par le couple 10/00 dans le modèle de la machine fausse M4. Les

graphes 3b et 3c montrent qu'à partir de l'état 2 et de l'entrée 10, les machines M1 et M2 se trouvent dans un état différent de l'état 3 de la machine juste M. La machine M1 se trouve dans l'état 2 qui est un état de l'ensemble S = {1, 2, 3} des états de M alors que la machine M2 se trouve dans l'état 4 qui n'appartient pas à S.

I. 3 Terminologie

Dans ce paragraphe nous définissons les termes que nous emploirons tout au long de ce chapitre.

Définition 1. L'*état d'initialisation* d'une machine est l'état de départ S0 dans lequel elle se trouve après avoir reçu une commande d'initialisation.

Définition 2. Une *séquence de justification d'un état* s d'une machine d'états finis est une séquence qui, à partir de l'état d'initialisation, permet de mettre la machine dans l'état s souhaité.

Définition 3. Etant donné une faute dans une copie du circuit combinatoire d'une machine d'états finis, les entrées secondaires et primaires de cette copie qui permettent de justifier la valeur de manifestation de la faute définissent respectivement l'*état de manifestation* et l'*entrée de manifestation* de la faute.

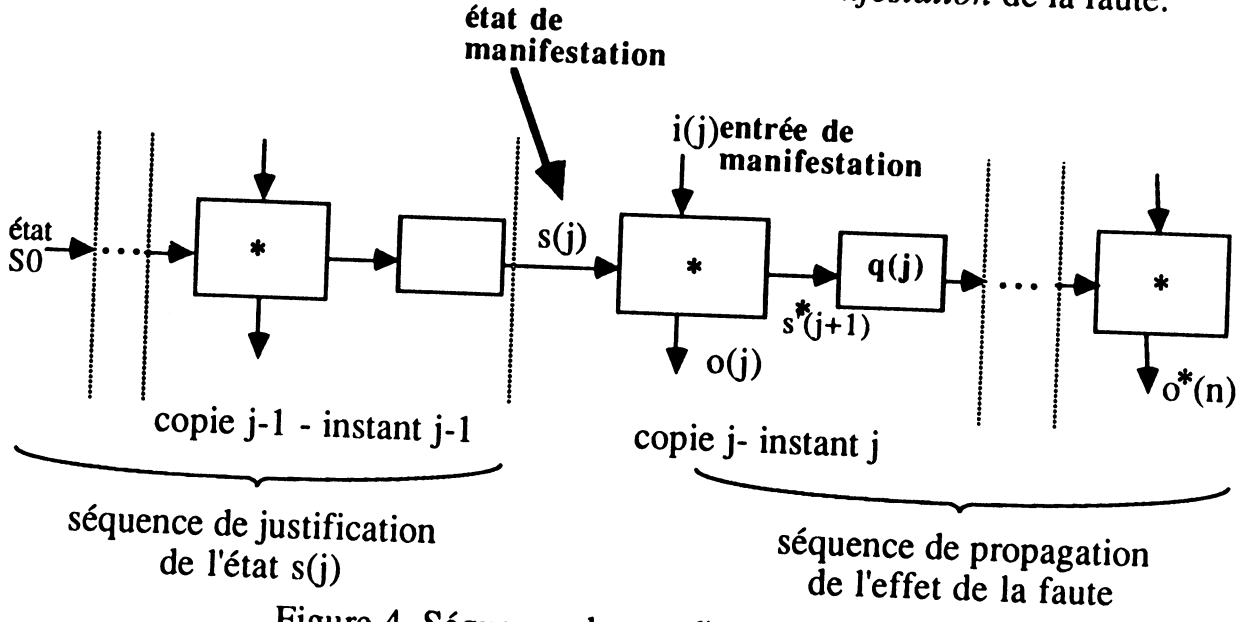


Figure 4. Séquence de test d'une faute

Définition 4. Une transition (i, s, s_f', o_f) d'une machine d'états finis est dite *transition contaminée* par une faute si elle permet à la faute de se propager sur les sorties $\lambda(s, i) \neq \lambda_f(s, i)$ ou sur l'état suivant $\partial(s, i) \neq \partial_f(s, i)$. L'état suivant est alors dit *état contaminé* par la faute.

Définition 5. Une faute indétectable est une faute pour laquelle il n'existe pas de séquence de test.

Définition 6. Etant donné deux machines d'états finis M1=(Σ, O1, S1, S0, ∂1, λ1) et M2=(Σ, O2, S2, S0, ∂2, λ2) ayant le même ensemble d'entrée Σ et le même état d'initialisation S0, le *produit cartésien* M = M1*M2 de ces deux machines est une machine d'états finis définie par le 6-uplet (Σ, O, S,(S0,S0), ∂, λ) où
O = O1 x O2, S = S1 x S2, avec
∂[(s1, s2), i]= [∂1(s1, i), ∂2(s2, i)] et

λ[(s1,s2)]= [λ1(s1), λ2(s2)] pour une machine de Moore
λ[(s1,s2), i]= [λ1(s1, i), λ2(s2, i)] pour une machine de Mealy
Le produit cartésien entre une machine juste et une machine fausse sera utilisé pour générer la séquence qui permet de distinguer la machine juste de la machine fausse.

I. 4 Etat de l'art sur la génération de test des machines d'états finis fondée sur la description fonctionnelle

Remarquons tout d'abord qu'il ne faut en aucun cas confondre les approches d'identification d'automates présentées dans [Moo56], [Gin58], [Gil61], [Hen64], [Hsi71], [Fri73], [Koh78] avec les approches de génération de test pour les automates d'états finis. En effet dans les approches d'identification on cherche à vérifier si une machine séquentielle a un comportement conforme à une spécification abstraite de type automate d'états finis et ceci en l'absence de toute faute. Cette équivalence n'est vérifiée précisément qu'en absence de fautes. En effet la séquence dite "homing" [Moo56] qui met la machine dans un état connu et la séquence de "distinction" [Moo56] qui permet de distinguer entre deux états d'une machine juste, sont calculées sur le modèle de la machine juste et ne sont pas valides dans les machines fausses.
Historiquement une des méthodes les plus anciennes utilisant le modèle fonctionnel est celle de Poage [Poa63]. Cette méthode part de la machine juste et de l'ensemble des machines fausses et suppose que l'état d'initialisation est robuste par rapport aux fautes. Les fautes considérées sont du type collage dans le circuit combinatoire. Cette méthode consiste à calculer le produit cartésien de la machine juste avec chacune des machines fausses afin de générer les séquences qui permettent de distinguer entre la machine juste et chacune des machines fausses. Les séquences résultantes sont des séquences de test valides puisqu'elles ont été générées sur le modèle des machines juste et fausses.
Cependant le stockage des machines fausses a l'inconvénient de nécessiter un grand espace de mémoire [Poa63]. Pour pallier ce problème des méthodes de test récentes [Ma87], [Dev87], [Gho89], [Ash90] proposent de générer la séquence de justification de l'état de manifestation d'une faute en utilisant le modèle de la machine juste et de

valider cette séquence ultérieurement par simulation de fautes. Par ailleurs [Ma87] confirme que dans la plupart des exemples traités, la séquence de justification calculée sur le modèle de la machine juste est, soit une séquence de justification valide aussi pour la machine fausse, soit une séquence de test de la faute.
Les méthodes de génération de test considèrent principalement les fautes logiques du type collage. Dans [Che90] et [Pom91] des modèles fonctionnels de fautes qui sont des fautes induisant de fausses transitions sont considérés. La méthode de [Che90] travaille uniquement sur le modèle de la machine juste. Elle suppose qu'une seule transition est fausse à la fois et propose de chercher les séquences définies dans [Hen64] qui permettent d'identifier l'état final parmi tous les autres états de la machine juste. L'hypothèse qu'une seule transition est fausse à la fois ne semble pas réaliste. En effet la présence d'une faute dans le circuit contamine normalement plusieurs transitions (δ et λ deviennent δ_f et λ_f). De plus la séquence d'identification d'un état n'est plus valide puisqu'elle est calculée sur le modèle de la machine juste. L'autre inconvénient est qu'une séquence d'identification n'existe pas toujours [Koh78]. On précise que la séquence générée par cette méthode a l'avantage de fournir une très bonne couverture de fautes au niveau logique (96% en moyenne sur les exemples traités) avec un temps de génération beaucoup plus petit mais une longueur de séquence plus grande (1,5 à 2 fois plus grande) que celle générée par une approche classique comme le D-algorithme ou l'une de ses dérivés et ceci pour des exemples de référence internationale. La couverture obtenue est dûe précisément à la longueur de la séquence générée qui se rapproche de celle d'une séquence aléatoire.

II Méthode mixte proposée pour le test d'automate d'états finis

Dans cette thèse nous proposons une méthode mixte de génération de test [Kar91a], [Kar91d], [Jay90b]. C'est une combinaison de deux approches à savoir une génération de test par identification suivie d'une génération de test par distinction.
La phase de génération de test par identification est indépendante des modèles de fautes. Elle consiste à parcourir le graphe d'états de la machine juste en passant par tous les arcs du graphe. La séquence obtenue est simulée par un simulateur de fautes qui indique les fautes qui ne sont pas couvertes.
La phase de génération de test par distinction prend en compte les machine juste et fausses représentées par leur graphe d'états. Les machines fausses sont celles qui sont associées aux fautes non couvertes par la première phase. Cette phase est fondée sur l'approche de Poage [Poa63]. Elle utilise les produits cartésiens entre la machine juste et chacune des machines fausses et elle est assistée d'un simulateur de

fautes. Elle suppose que l'état d'initialisation est robuste par rapport aux fautes. Elle consiste à calculer les séquences qui permettent de distinguer la machine juste de chacune des machines fausses.
Dans le paragraphe suivant §III nous faisons un rappel sur la théorie des graphes qui nous sera utile dans le paragraphe §IV. Le paragraphe §IV présente la première phase de la génération de test qui consiste à parcourir le graphe de la machine juste. L'évaluation de la séquence générée par cette première phase est effectuée dans le paragraphe §V. Le paragraphe §VI expose la deuxième phase de génération de test fondée sur le produit cartésien de la machine juste et des machines fausses.

III Rappels sur la théorie des graphes

Soit G(N, A) un graphe orienté connexe, N= {ni} l'ensemble des nœuds et A= {ai} l'ensemble des arcs.
Nous désignons par

d^-(ni) le nombre d'arcs entrants d'un nœud ni
d^+(ni) le nombre d'arcs sortants d'un nœud ni
N0 le nœud correspondant à l'état d'initialisation

III. 1 Définitions générales sur les graphes

Définition 1. Dans un graphe orienté, un *chemin* est une séquence d'arcs [a1, a2, ...an] dans laquelle l'extrémité terminale d'un arc ai est l'extrémité initiale de l'arc ai+1. Le nœud initial (terminal) de ai est dit *nœud prédécesseur* (*nœud successeur*) du nœud terminal (initial). L'arc ai est appelé *arc successeur* (*arc prédécesseur*) de son nœud initial (terminal).
Définition 2. Dans un graphe orienté une *chaîne* est une séquence d'arcs [a1, a2,... ai...aq] dans laquelle les arcs ai et ai+1 ont une extrémité commune.
Définition 3. Une *chaîne élémentaire* est une chaîne tel qu'en la parcourant on ne rencontre pas deux fois le même nœud.
Soit l'exemple du graphe de la figure 5. La séquence [a1, a4, a5, a6] constitue le chemin de N0 à n3. Les séquences [a1, a4, a5, a6] et [a1, a2, a3] sont les deux chaînes élémentaires de N0 à n3.

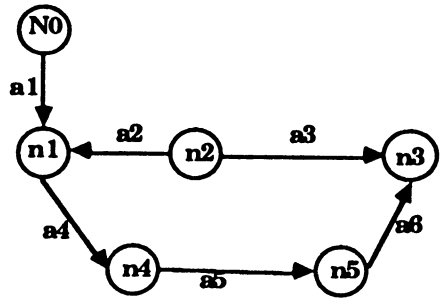


Figure 5. Graphe orienté

Définition 4. Le *degré* D(ni) d'un nœud ni est la différence entre le nombre d'arcs entrants à un nœud ni et le nombre des arcs qui en sortent: $D(ni) = d^-(ni) - d^+(ni)$.

Définition 5. Un graphe est dit *fortement connexe* si étant donnés deux nœuds quelconques ni et nj, il existe un chemin d'extrémité initiale ni et d'extrémité terminale nj.

Définition 6. Un *graphe partiel d'un graphe* G est un graphe comprenant tous nœuds de G et un sous-ensemble des arcs de G.

Définition 7. Une *arborescence* de racine $r \in N$ est un graphe orienté G(N, A) connexe sans cycle et tel que pour tout nœud ni de N il existe un chemin allant de r à ni.

Définition 8. Une *arborescence complète d'un graphe* G est un graphe orienté partiel de G connexe et sans cycle.

III. 2 Rappels sur les graphes Eulériens [Gon85], [Gib85]

Définition 9. Un *chemin Eulérien* est un chemin qui passe une fois et une seule par chacun des arcs du graphe.

Définition 10. Un *circuit Eulérien* est un chemin Eulérien pour lequel les nœuds de départ et d'arrivée sont identiques.

Définition 11. Un *graphe Eulérien* est un graphe qui admet un chemin Eulérien ou un circuit Eulérien.

Définition 12. Dans un graphe connexe G(N, A), un *parcours du postier chinois* est un parcours qui passe au moins une fois par tous les arcs du graphe et dont la longueur totale est minimale.

Théorème 1. Un graphe connexe admet un circuit Eulérien si et seulement si tous les nœuds du graphe ont un degré nul [Gib85].

Théorème 2. Un graphe connexe G admet un chemin Eulérien (N0, Nf) de G si et seulement si

$D(N0) = -1$,

$D(Nf) = 1$,

$D(ni) = 0$ pour tout nœud ni du graphe tel que $ni \neq N0$ et $ni \neq Nf$.

Théorème 3. Dans un graphe non Eulérien un parcours du postier chinois existe si et seulement si le graphe est fortement connexe [Gib85].

III. 3 Rappels sur la théorie du flot

Définition 13. Dans un graphe G(N, A) une *source* S est un nœud tel que $d^+(S)>0$ et un *puits* P est un nœud tel que $d^-(P)<0$.

Définition 14. Soit un graphe G(N, A) qui comprend un nœud source S et un nœud puits P. Un *graphe étiqueté par un flot de S à P* est un graphe orienté G(N, A) tel qu'à tout arc ai est associée bijectivement une quantité f(ai) ≥ 0 tel que

1. pour tout nœud ni du graphe (ni ≠ S et ni ≠ P), les arcs prédécesseurs a' et les arcs successeurs a" vérifient:

$$\sum_{a' \in \{\text{arcs prédécesseurs de ni}\}} f(a') = \sum_{a'' \in \{\text{arcs successeurs de ni}\}} f(a'')$$

2. pour tout arc a' successeur de S et pour tout arc a" prédécesseur de P on a:

$$\sum_{a' \in \{\text{arcs successeurs de S}\}} f(a') = \sum_{a'' \in \{\text{arcs prédécesseurs de P}\}} f(a'') = F$$

f(a) est appelé *quantité de flot* sur l'arc a et F est appelé *valeur du flot* sur le graphe.

Définition 15. Un *réseau de transport* est un graphe connexe G(N, A) étiqueté par un flot dans lequel chaque arc a est muni d'un nombre c(a) ≥ 0 appelé *capacité de l'arc*. La capacité c(a) indique la limite supérieure de la quantité de flot f(a) admissible sur l'arc a. $0 \leq f(a) \leq c(a)$

Définition 16. Dans un graphe G(N, A) étiqueté par un flot, une *chaîne non saturée* de S à P est une chaîne élémentaire (S, P) telle que pour tout arc (ni, nj) de la chaîne on a:

si (ni, nj) est un arc direct alors
c(ni, nj) - f(ni, nj) > 0

si (ni, nj) est un arc inverse alors
f(nj, ni) > 0

La *capacité résiduelle* de l'arc direct (ni, nj) est égale à c(ni, nj) - f(ni, nj). Dans le cas où (ni, nj) est un arc inverse, on définit la capacité résiduelle de l'arc (nj, ni) égale à f(nj, ni).

Définition 17. Dans un graphe G(N, A) étiqueté par un flot de S à P, la *capacité résiduelle Δ d'une chaîne non saturée* (S, P) est l'accroissement maximal de la valeur du flot pouvant être associé à la chaîne.
Δ = min Δ(a) où Δ(a) est la capacité résiduelle des arcs de la chaîne.

Théorème 4. Etant donné un réseau de transport G(N, A) étiqueté par un flot de S à P de valeur F sur G. Si une chaîne non saturée (S, P) existe alors on peut construire un autre flot de S à P de valeur F' égale à la valeur du flot de départ F augmentée de la capacité Δ de la chaîne F' = F + Δ.

IV Première phase de test consistant à parcourir tous les arcs du graphe

Cette phase consiste à passer par tous les arcs du graphe d'états de la machine juste. Une première approche présentée dans [Jay89] consiste à transformer ce problème en un problème de recherche de chemins passant par les nœuds du graphe dual. Le graphe dual G* est un graphe dans lequel les nœuds sont les arcs du graphe initial G. Un arc (ni, nj) dans le graphe G* correspond à une transition ni suivie de la transition nj dans G. La recherche d'un chemin minimal passant par les nœuds du graphe G* est un problème NP-complet. L'algorithme appliqué est fondé sur une heuristique de complexité polynômiale.
Dans notre approche on considère le graphe initial et on cherche à parcourir directement les arcs du graphe initial. L'intérêt de revenir au problème initial est l'existence d'algorithme de complexité polynômiale pour le résoudre.

D'après la définition 9 (§III.2) le parcours recherché est un parcours du postier chinois. Dans le cas d'un graphe Eulérien le parcours du postier chinois est un chemin Eulérien ou un circuit Eulérien. C'est le parcours optimal passant par tous les arcs une seule fois.
Pour un graphe non Eulérien un parcours optimal existe si le graphe est fortement connexe (théorème 3 §III.2). Pour les machines d'états finis les graphes d'états sont des graphes connexes. Ces graphes peuvent être rendus fortement connexes en rajoutant des arcs d'initialisation asynchrone. Rappelons en effet que dans tout circuit les bascules peuvent en principe être initialisées à zéro de façon asynchrone (commande Preset). Il n'est pas nécessaire dans cette approche d'ajouter un arc d'initialisation à la sortie de chaque nœud du graphe. Il suffit d'ajouter un arc d'initialisation sortant des composantes fortement connexes du graphe. Un algorithme minimisant le nombre des arcs d'initialisation à ajouter est proposé. Il est de complexité O(max(|A|, |N|)).

L'algorithme de recherche d'un parcours passant au moins une fois par chaque arc du graphe est le suivant:
Si le graphe est Eulérien **alors** recherche d'un parcours Eulérien pour graphe Eulérien,
fin si
Si le graphe n'est pas Eulérien **alors**
    Si le graphe est fortement connexe **alors**
        recherche d'un parcours Eulérien pour graphe non Eulérien

fin_si
Si le graphe n'est pas fortement connexe **alors**
rendre le graphe fortement connexe,
recherche d'un parcours Eulérien pour graphe non Eulérien
fin_si
fin_si

IV. 1 Recherche d'un parcours Eulérien dans un graphe Eulérien

Cette procédure se fait en deux phases et le parcours résultant est un chemin Eulérien ou un circuit eulérien. Dans la suite on parle plus généralement de chemin. La première phase consiste à extraire une arborescence complète T du graphe. T est calculé par un algorithme d'exploration en profondeur d'abord (deep search first) ou par un algorithme d'exploration en largeur d'abord (breadth search first). Ce sont des algorithmes simples de complexité polynômiale. Dans ce travail un algorithme de parcours par profondeur a été implanté. Il est de complexité O(|A|).

La deuxième phase est dédiée à la recherche du chemin Eulérien lui même. Ce chemin est construit progressivement en partant du nœud terminal du chemin jusqu'à arriver au nœud initial. Le chemin est donné sous la forme d'une liste d'arcs traversés. A chaque pas on part d'un nœud courant ni et on cherche un nouveau arc à traverser qui soit un arc prédécesseur de ni et qui n'appartienne pas au chemin. Les arcs qui appartiennent à l'arborescence T sont les derniers arcs traversés.

Exemple

Considérons l'exemple de la machine Regshift représentée par le graphe d'états de la figure 6. Cette machine comporte 8 états N0, ..., n7 et 17 arcs. Les prédicats de la machine ne sont pas indiqués par souci de lisibilité.

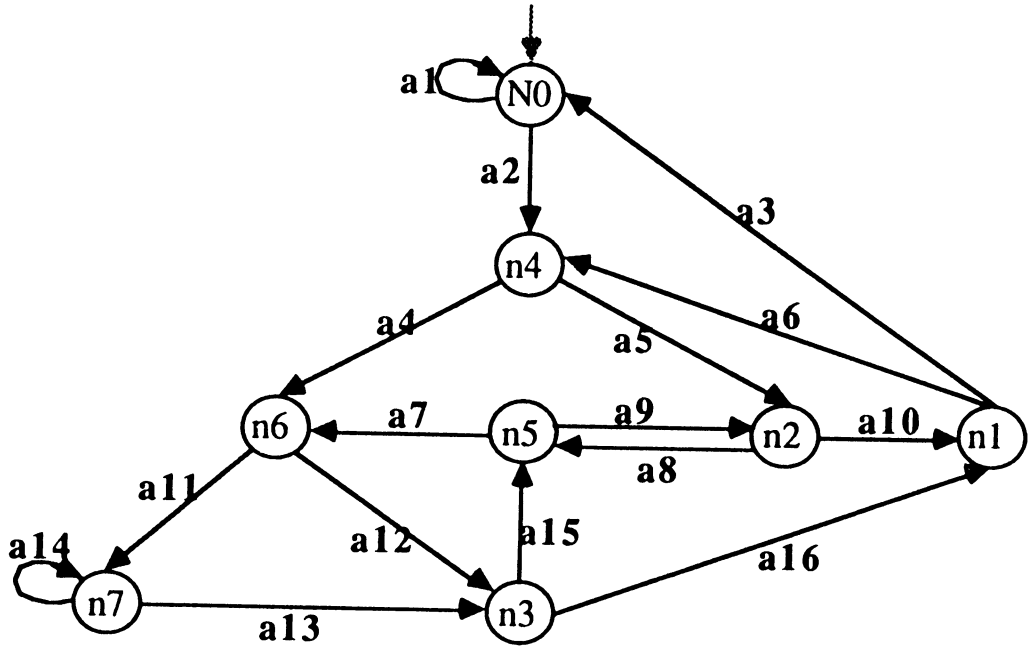
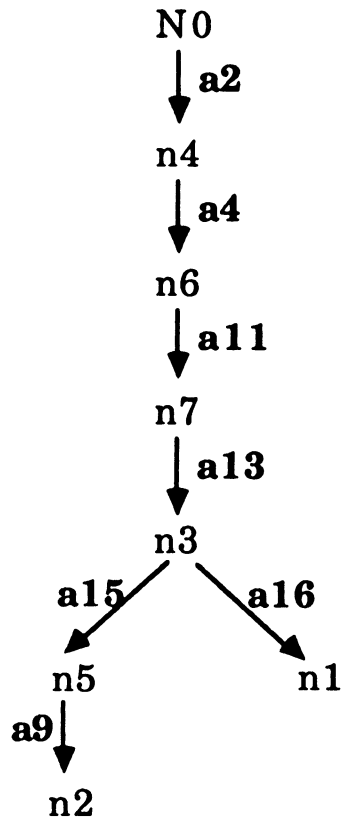


Figure 6. Exemple d'un graphe Eulérien

Ce graphe est connexe. Les nœuds ni de ce graphe vérifient la propriété suivante d+(ni) = d-(ni). Il est Eulérien et contient un circuit Eulérien. La recherche du circuit Eulérien commence par l'extraction d'une arborescence complète du graphe ayant le nœud d'initialisation comme racine. L'algorithme d'exploration en profondeur d'abord est appliqué. Le résultat est l'arborescence T illustré sur la figure 7a.

L'étape suivante associe à chaque nœud ni du graphe deux listes à savoir la liste des nœuds prédécesseurs et des arcs prédécesseurs de ni. L'arc prédécesseur appartenant à l'arborescence T et le nœud prédécesseur correspondant sont les derniers éléments des listes. Pendant la phase de recherche d'un circuit Eulérien, l'arc et le nœud prédécesseurs d'un nœud courant sont choisis parmi les éléments des deux listes et dans l'ordre de ces listes. Ainsi les éléments prédécesseurs d'un nœud et appartenant à l'arborescence T sont les derniers à être choisis.

Pour l'exemple de la figure 6, les listes des arcs A_{ni} et des nœuds B_{ni} prédécesseurs des nœuds ni du graphe figurent sur la figure 7b. Considérons par exemple les listes A_{n2} et B_{n2} associées au nœud n2. La première contient les nœuds prédécesseurs n4 et n5 et la seconde contient les arcs a5 et a9. Le nœud n5 et l'arc correspondant a9 sont les derniers éléments des listes A_{n2} et B_{n2} puisque l'arc a9 appartient à l'arborescence T.



(a) Une arborescence complète T de G

A_{n0} = [n1, N0]
B_{n0} = [a3 , a1]
A_{n1} = [n2, n3]
B_{n1} = [a10, a16]
A_{n2} = [n4, n5]
B_{n2} = [a5 , a9]
A_{n3} = [n6, n7]
B_{n3} = [a12, a13]
A_{n4} = [n1, N0]
B_{n4} = [a6 , a2]
A_{n5} = [n2, n3]
B_{n5} = [a8 , a15]
A_{n6} = [n5, n4]
B_{n6} = [a7 , a4]
A_{n7} = [n7, n6]
B_{n7} = [a14, a11]

(b) Listes A_{ni} et Bni

Figure 7. Une arborescence complète du graphe et listes des prédécesseurs

Le nœud N0 est l'extrémité initiale et finale du circuit Eulérien recherché.
A la première itération l'arc a3 adjacent à N0 n'ayant pas encore été sélectionné, est traversé. Le nœud n1 devient le nœud courant et une nouvelle itération commence. L'algorithme s'arrête lorsque tous les arcs sont traversés. Le circuit Eulérien résultant en termes d'arcs est le suivant: [a1, a2, a4, a11, a14, a13, a15, a9, a8, a7, a12, a16, a6, a5, a10, a3] et en termes de nœuds [N0, N0, n4, n6, n7, n7, n3, n5, n2, n5, n6, n3, n1, n4, n2, n1, N0].

IV. 2 Recherche d'un parcours Eulérien dans un graphe non Eulérien

Dans un graphe non Eulérien G, il existe des nœuds ni de degré D(ni) non nul. Un parcours passant par tous les arcs du graphe nécessite la répétition des arcs sortants de ni de degré D(ni)>0 et des arcs entrants à nj de degré D(nj)<0 et parfois la répétition en nombre égal des arcs entrants et sortants à nk de degré D(nk)=0.

Dans un graphe non Eulérien G(N, A), la solution consiste à rajouter des arcs afin d'obtenir un graphe dans lequel les nœuds sont de degré nul tout en minimisant le nombre des arcs répliqués.
Il a été démontré [Gon85] que les arcs à répéter dans G forment un ensemble de chaînes allant des nœuds de degré positif (D>0 dans G) aux nœuds de degré négatif (D<0 dans G) et passant par des nœuds de degré nul (D=0).

La solution proposée est fondée sur la théorie des flots et comprend les quatre étapes suivantes. L'algorithme implantant ces étapes est de complexité O(max(|A|, |N|)).

étape1. Adjonction de deux nœuds au graphe
Un nœud source S et un nœud puits P sont ajoutés au graphe. On relie S aux nœuds ni de G tels que D(ni)>0 et P aux nœuds nj de G tels que D(nj)<0. A chacun des arcs du graphe ainsi construit, est affectée une capacité. Aux arcs reliant S aux ni on affecte une capacité égale à D(ni) et à ceux reliant les nj à P on affecte une capacité égale à -D(nj). Aux arcs restants est affectée une capacité de valeur infinie.

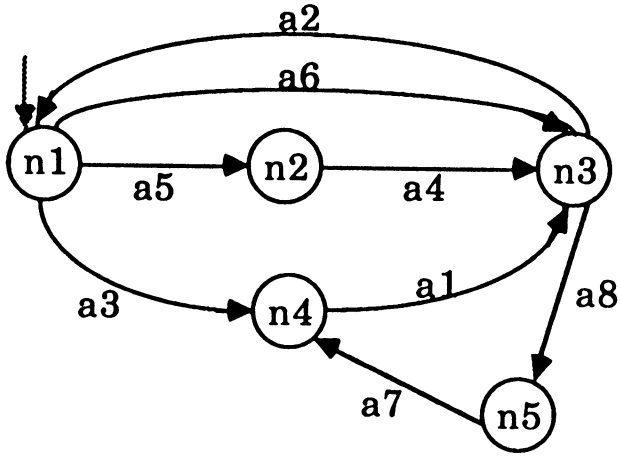
étape2. Recherche d'un flot maximal de S à P.
A chacun des arcs du graphe on affecte une valeur f qui correspond au nombre de sa réplication. Cette valeur est initialisée à zéro.
Le nombre de réplication des arcs définit un flot de S à P. Soit F la valeur de ce flot $F = \sum_{(S, u) \text{successeur de } S} f(S, u)$. On cherche à rendre maximale la valeur F pour atteindre $F_{max} = \sum_{(S, u) \text{successeur de } S} c(S, u)$.
A l'initialisation on part avec une valeur du flot nulle F=0 que l'on augmente par la suite jusqu'à atteindre la valeur maximale $F = F_{max}$ (théorème 4 §III.3). Cette augmentation se fait pas à pas. A chaque pas on cherche la plus petite chaîne non saturée entre S et P. Sur cette chaîne la valeur du flot augmente d'une quantité égale à la capacité résiduelle Δ de la chaîne. La quantité de flot des arcs directs se trouve augmentée alors que celle des arcs inverses se trouve diminuée de la quantité Δ. Ainsi pour les arcs (ni, nj) de la chaîne:
f(ni, nj) <-- f(ni, nj)+Δ si (ni, nj) est un arc direct
f(nj, ni) <-- f(nj, ni)-Δ si (ni, nj) est un arc inverse
et la valeur flot sur G est augmentée de Δ: F<--F+Δ. La valeur du flot F atteint son maximum F_{max} lorsqu'aucune chaîne non saturée n'existe plus de S à P.

étape3. Construction d'un graphe Eulérien
Le graphe Eulérien est déduit en supprimant la source S et le puits P et tous leurs arcs successeurs et prédécesseurs. Les arcs du graphe G sont répliqués un nombre de fois égale à leur quantité de flot respective.

étape4. Recherche d'un circuit Eulérien
G est maintenant un graphe Eulérien pour lequel un circuit Eulérien est trouvé suivant l'algorithme donné en (§IV.1).

Exemple

Considérons l'exemple de l'automate d'états finis de la figure 8. Cette machine comporte 5 états n1... n5 et 8 arcs a1... a8. Le graphe G correspondant à cette machine est un graphe fortement connexe. G est un graphe qui ne vérifie aucune des deux conditions nécessaires pour qu'un graphe soit Eulérien (théorème 1 et théorème 2 §III.2). G est donc un graphe non Eulérien.



ni	D(ni)
n1	-2
n2	0
n3	+1
n4	+1
n5	0

(a) Graphe G (b) Degré des nœuds

Figure 8. Graphe fortement connexe et non Eulérien

L'algorithme de transformation d'un graphe non Eulérien en graphe Eulérien est appliqué sur le graphe G. Ceci se fait en 3 étapes. La première étape consiste à rajouter une source S et un puits P et de définir les capacités sur les arcs. La deuxième étape calcule un flot maximal. La troisième étape construit le graphe Eulérien.

Recherche d'un flot maximal

La valeur maximale F_{max} du flot à chercher est égale à la somme des capacités des arcs successeurs de S et est donc égale à 2. La figure 9a montre le graphe après la

procédure d'initialisation de l'algorithme de la recherche d'un flot maximal. Pour chacun des arcs du graphe, la valeur encadrée représente la quantité de flot de l'arc et la valeur non encadrée sa capacité. A l'initialisation la quantité de flot des arcs est nulle et par suite la valeur du flot F sur G est elle aussi nulle.

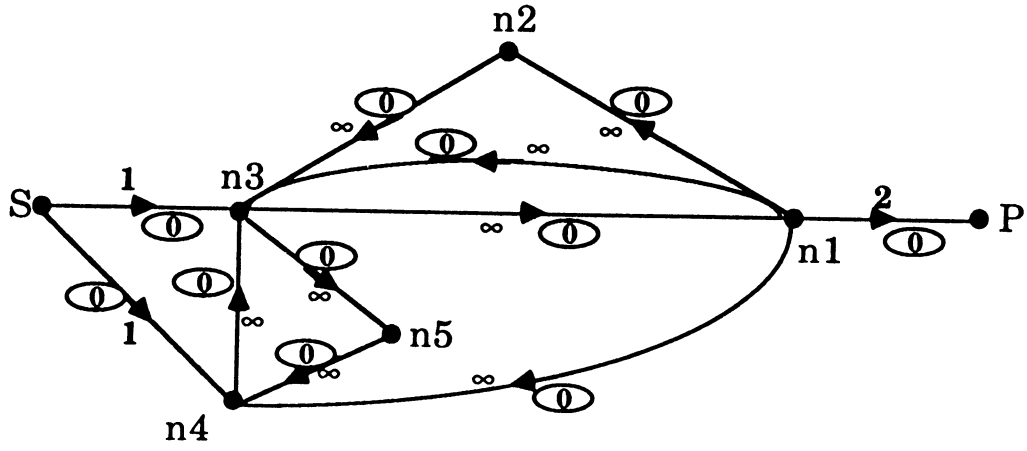


Figure 9a. Initialisation

La figure 9b résume la première itération de l'algorithme. Dans cette figure la plus petite chaîne non saturée de S à P est [(S, n3), (n3, n1), (n1, P)]. La capacité résiduelle de cette chaîne est égale à la valeur minimale des capacités résiduelles de ses arcs.

$\Delta = \min(\Delta(S, n3), \Delta(n3, n1), \Delta(n1, P)) = \min(1, \infty, 2) = 1$. Les arcs de la chaîne sont tous des arcs directs. Leur quantité de flot est alors augmentée de 1.

La valeur du flot $F = f(S, n3) + f(S, n4)$ du graphe est aussi augmentée de 1. Elle devient égale à 1.

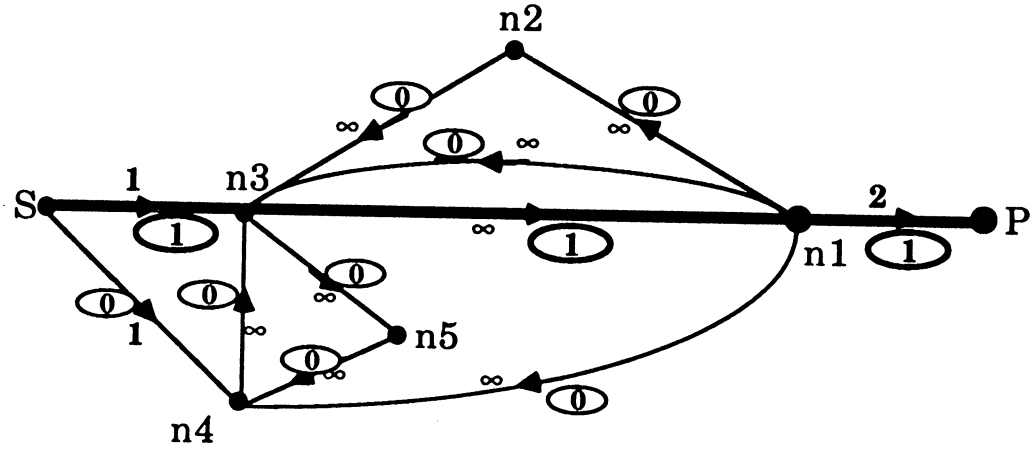


Figure 9b. Itération 1: Recherche de la plus petite chaîne non saturée de S à P

Puisque la valeur du flot F est égale à 1 et donc n'a pas encore atteint sa valeur maximale $F_{max} = 2$, une autre itération est nécessaire.

La figure 9c résume la deuxième itération de l'algorithme. La plus petite chaîne existante non saturée de S à P est [(S, n4), (n4, n3), (n3, n1), (n1, P)]. La capacité résiduelle de cette chaîne est égale à

$\Delta = \min(\Delta(S, n4), \Delta(n4, n3), \Delta(n3, n1), \Delta(n1, P)) = \min(1, \infty, \infty, 1) = 1$. Les arcs de la chaîne sont tous des arcs directs. Leur quantité de flot est alors augmentée de 1.

La valeur du flot $F = f(S, n3) + f(S, n4)$ sur le graphe est elle aussi augmentée de 1 et devient donc égale à 2.

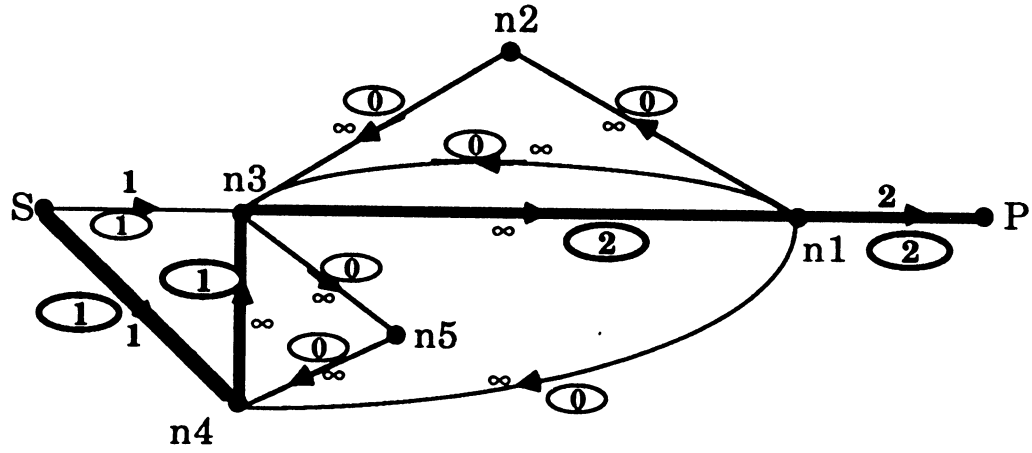


Figure 9c. Itération 2 : Recherche de la plus petite chaîne non saturée de S à P

L'algorithme s'arrête puisque la valeur du flot F a atteint sa valeur maximale $F_{max} = 2$.

Construction du graphe Eulérien

Le graphe Eulérien (figure 9d) est obtenu

en supprimant la source S et le puits P ainsi que les arcs (S, n3), (S, n4) et (n1, P) qui leur sont attachés et

en dupliquant une fois l'arc a1 de flot égale à 1 et deux fois l'arc a2 de flot égale à 2.

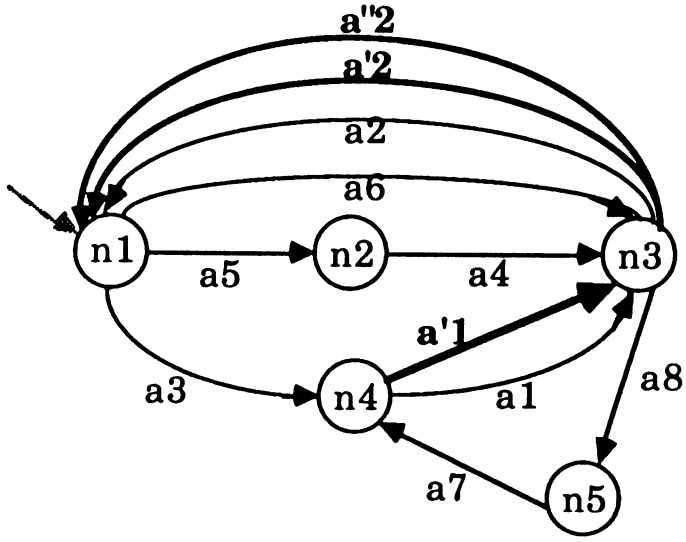


Figure 9d. Graphe Eulérien

Le circuit Eulérien donné en terme d'arcs est [a5, a4, a8, a7, a1, a"2, a3, a'1, a'2, a6, a2].

IV. 3 Implantation et résultats

Un logiciel de recherche d'un chemin eulérien dans un graphe orienté a été implanté en langage C. Il comporte près de 450 lignes. Ce logiciel accepte, en entrée, une description en format normalisé "kiss" de machines d'états finis (utilisée pour les transferts à travers réseau informatique) et fournit, en sortie, le chemin eulérien sous forme d'une liste.

Ce logiciel a tourné sur plusieurs exemples de circuits de référence internationale MCNC sur une station de travail SUN4. Les résultats sont récapitulés dans les tableaux 2 et 3 ci-dessous. Le tableau 2 montre aussi les résultats donnés par l'algorithme de recherche de chemin en passant par le graphe dual. Cet algorithme est noté "ancienne méthode" alors que l'algorithme correspondant au parcours eulérien est référencié par "nouvelle méthode".

Dans les tableaux 2 et 3, une machine est caractérisée par le nombre de ses états, transitions, entrées et sorties. On note si le graphe correspondant est eulérien ou non. Dans le cas où il n'est pas eulérien on donne le nombre des arcs à dupliquer. La longueur du chemin en est déduite. Le temps de calcul est exprimé en seconde.

	nb. états	nb. arcs	nb. entrées	nb. sorties	graphe eulérien ?	nombre des arcs dupliqués		longueur du chemin		Temps de calcul (secondes)	
						ancienne méthode	nouvelle méthode	ancienne méthode	nouvelle méthode	ancienne méthode	nouvelle méthode
u802b	15	39	10	16	non	85	25	124	63	87	0.1
flammand	20	31	11	9	non	21	11	52	42	43	0.06
mouse	3	4	4	4	non	6	1	10	5	15	10^{-6}
big3	8	18	4	1	non	7	2	25	20	13.8	10^{-6}
traffic	4	9	5	5	oui	2	0	11	9	10.8	$< 10^{-6}$
s1	20	108	10	6	non	273	60	383	167	357.6	0.16
planet	48	116	9	19	non	613	168	729	284	279.6	0.5
hyeti	175	455	29	71	non	2416	746	2871	1201	2638	493

Tableau 2. Résultats de l'algorithme de parcours de graphe

Le tableau 2 montre que la nouvelle méthode donne une séquence de test plus courte que l'ancienne. De plus le temps de la génération du chemin est nettement plus réduit dans la nouvelle méthode que dans l'ancienne.
Le tableau 3 illustre des résultats sur d'autres machines de référence internationale. Ces machines serviront dans la suite à des fins de comparaisons entre notre méthode et celles de [Che90] et [Dev87].

	nb. états	nb. arcs	nb. entrées	nb. sorties	graphe eulérien	nombre des arcs dupliqués	longueur du chemin	Temps de calcul (secondes)
bbara	10	60	6	2	non	30	90	0.06
cse	16	91	9	7	non	116	207	0.29
dk14	7	56	5	5	non	28	84	0.06
dk15	4	32	5	5	non	12	44	0.03
dk16	27	108	4	3	non	51	159	0.26
dk17	8	32	4	3	non	31	63	0.04
sand	32	184	13	9	non	124	308	0.22
styr	30	166	11	10	non	175	341	0.59
tav	4	49	6	4	non	15	64	0.02

Tableau 3. Résultats de l'algorithme de parcours de graphe

V Evaluation de la première phase

La première phase fournit en sortie une première séquence de vecteurs en termes de prédicats et valeurs correspondantes des sorties. Ceci détermine la première séquence de test. Un moyen d'évaluer cette séquence consiste à dresser une liste de fautes que l'on voudrait détecter dans le circuit et à calculer le nombre des fautes couvertes par cette séquence. On rappelle que cette première séquence de test est générée indépendamment de la structure du circuit et des hypothèses de fautes. Ainsi on ne peut pas déterminer à priori les fautes qu'elle couvre au niveau structurel.
Le problème de la couverture des fautes est résolu à l'aide d'un simulateur de fautes qui, à partir d'une description du circuit, d'une liste de fautes à couvrir et de la séquence de test, fournit le taux de couverture et délivre la liste des fautes non détectées par cette séquence. Le taux de couverture des fautes est défini comme étant le pourcentage des fautes détectées sur le nombre total des fautes injectées et simulées.

$$\text{Taux de couverture} = \frac{\text{nombre des fautes détectées}}{\text{nombre des fautes injectées}} \times 100$$

Toutefois le taux de couverture ne permet pas à lui seul d'évaluer complètement une séquence de test puisqu'il ne tient pas compte de la longueur de la séquence. Dans l'exemple d'une porte "and" à deux entrées, une séquence exhaustive (quatre vecteurs) ainsi que la séquence formée des trois vecteurs 11 01 10 permettent de tester à 100% le collage à 0 et 1 des entrées et sortie de la porte.
Pour prendre la longueur de la séquence en compte on introduit un deuxième paramètre d'évaluation qui est la longueur normalisée. C'est le pourcentage de la longueur de la séquence générée sur le nombre des arcs du graphe.

$$\text{Longueur normalisée} = \frac{\text{longueur de la séquence générée}}{\text{nombre des arcs du graphe}} \times 100.$$

Le facteur de qualité d'une séquence est défini à partir de ces deux paramètres. Ce facteur est proportionel au taux de couverture et inversement proportionnel à la longueur normalisée de la séquence de test. Il est défini comme étant le rapport du taux de couverture sur la longueur normalisée.

$$\text{Facteur de qualité} = \frac{\text{taux de couverture}}{\text{longueur normalisée}}$$

La séquence de test fournie au simulateur est définie à partir de la séquence de prédicats calculée par la première phase, en fixant les entrées indéterminées à "0" ou "1". Cette affectation à l'une ou à l'autre de ces valeurs a une influence considérable sur le taux de couverture. D'où l'idée de choisir ces valeurs de manière à augmenter le taux de couverture.

V. 1 Affectation des valeurs indéterminées

Deux heuristiques sont proposées pour fixer les valeurs indéterminées.
La première heuristique consiste à fixer systématiquement les valeurs indéterminées à la valeur binaire "0".
Dans la deuxième heuristique les valeurs indéterminées sont fixées suivant les deux critères suivants.
Critère 1. Chacune des entrées doit prendre les deux valeurs binaires "0" et "1".
Critère 2. Les vecteurs de test doivent être les "plus distincts possible". En d'autres termes ces vecteurs doivent parcourir le plus grand sous-ensemble de l'ensemble des valeurs de l'entrée.

Un algorithme itératif correspondant à la deuxième heuristique a été implanté. A chaque itération un vecteur appartenant à la séquence de test est considéré comme le vecteur courant pour lequel on cherche à fixer les valeurs indéterminées des entrées. Une liste correspondant à l'espace de valeurs que peut prendre un vecteur est stockée dans un tableau T. Une itération correspond aux trois pas suivants.

pas a- **tant que** le vecteur de test courant comporte encore une entrée indéterminée qui n'a pas pris les deux valeurs 0 et 1 dans un des vecteurs déjà fixés au cours des itérations précédentes **faire**
affecter la valeur indéterminée à la valeur non encore prise.
enregistrer la valeur choisie de l'entrée.
fait

pas b- **Si** le vecteur de test courant est complètement déterminé **alors** le vecteur du tableau qui lui est égal devient le dernier élément du tableau. Fin de l'itération.

pas c- **Si** le vecteur courant contient des valeurs indéterminées non encore fixées **alors**
on compare le vecteur de test courant avec le premier vecteur du tableau. Si les deux vecteurs sont identiques alors les valeurs indéterminées du vecteur courant sont fixées de manière à avoir deux vecteurs égaux et le premier vecteur du tableau est mis à la fin du

tableau. Si les deux vecteurs ne sont pas égaux alors on recommence le pas c avec le vecteur suivant du tableau. Fin de l'itération.

Il est important de signaler deux remarques concernant le (pas c) de cet algorithme. La première est qu'à la fin de ce pas le vecteur de test courant est égal à un vecteur du tableau. Ce vecteur est mis à la fin du tableau pour permettre au vecteur de test de l'itération suivante de se comparer avec un vecteur différent et avoir éventuellement une valeur différente des vecteurs de test précédents. La deuxième remarque est lorsque tout l'espace des vecteurs est parcouru et qu'il reste encore des vecteurs de test à variables indéterminées non fixées, on recommence à parcourir l'espace des valeurs des entrées de nouveau. En effet les vecteurs recherchés du tableau ne sont pas effacés mais mis tout simplement à la fin du tableau.

Exemple

Soit une machine d'états finis à trois entrées et soit la séquence de test suivante dans laquelle on veut fixer les valeurs indéterminées Xi.

vecteur 1 :	0	1	X1
vecteur 2 :	X2	0	X3
vecteur 3 :	X4	0	1
...			

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

Tableau 4. Tableau T à trois entrées

On construit le tableau T (tableau 4) qui couvre l'espace des valeurs possibles d'un vecteur à trois entrées. A la première itération le vecteur courant est "vecteur 1" de la séquence de test. Dans ce vecteur, la valeur X1 est fixée arbitrairement à la valeur "0" et le premier vecteur de test devient 010. Dans le tableau T le vecteur 010 est déplacé de la troisième jusqu'à la dernière position du tableau.
Dans la seconde itération le vecteur courant est le second vecteur de test "vecteur 2". Dans ce vecteur les valeurs X2 et X3 prennent la valeur "1" conformément au critère1. Le vecteur 2 devient égal à 101. Le vecteur 101 est déplacé depuis la cinquième position jusqu'à la fin du tableau.

Dans la troisième itération le vecteur courant est "vecteur 3". La première entrée a déjà pris la valeur "0" (dans "vecteur 1") et la valeur "1" (dans "vecteur 2"). On cherche à fixer la valeur de X4 de manière à ce que le vecteur résultant "vecteur 3" soit différent de "vecteur 1" et "vecteur 2" selon le critère 2. Une comparaison entre le premier élément du tableau 0 0 0 et le vecteur courant X4 0 1 montre que ces deux vecteurs ne sont pas identiques puisque la valeur de la troisième entrée est différente dans les deux vecteurs. Une comparaison entre le deuxième vecteur du tableau 001 et le vecteur courant montre une équivalence entre ces deux vecteurs si X4 est égal à "0". Ainsi X4 est fixé à la valeur "0" et "vecteur 3" devient 001. Le vecteur 001 est mis en dernière position dans le tableau.

V. 2 Implantation, simulation et résultats

Un logiciel correspondant à chacune des deux heuristiques a été implanté en langage C. Il comporte près de 250 lignes. Il fournit en sortie un fichier d'entrée au simulateur de fautes HIFAULT de GenRad installé au CIME (Centre Inter-universitaire de MicroEléctronique). Toutefois une option existe pour fournir un fichier d'entrée au simulateur Mach 1000.

La phase de simulation nécessite trois fichiers d'entrée au simulateur qui sont le fichier de description du circuit (netlist), le fichier définissant les fautes à injecter et le fichier de la séquence de test.

V. 2. 1 Fichier de description du circuit

Le schématique correspondant aux machines que l'on a traitées (tableaux 2 et 3) a été généré automatiquement par le générateur automatique de machines d'états finis de l'outil CAO de compilation de silicium de VLSI Technology. Ce compilateur prend la description d'une machine en format kiss en entrée et génère le schématique du circuit à base de cellules standard. La netlist du circuit généré est traduite ensuite en une autre netlist que le système HIFAULT ou Mach 1000 peut traiter. Cette traduction est effectuée par un utilitaire du système VLSI.

V. 2. 2 Fichier de fautes à injecter

C'est un fichier qui spécifie les fautes à injecter au circuit au niveau structurel. Le système HIFAULT peut traiter les fautes du type collage ("stuck fault") et circuit ouvert ("open fault"). Pour les circuits traités, on a considéré les fautes de collage à 0 et à 1.

V. 2. 3 Séquence de test

La séquence de test à envoyer au simulateur est celle qui a été générée par la première phase et dans laquelle les valeurs indéterminées ont été instanciées.

V. 2. 4 Comparaison de la qualité des séquences générées par l'ancienne et la nouvelle méthode

Le tableau 5 donne la longueur normalisée de la séquence de test fournie par l'ancienne méthode de parcours de graphe (passage par le graphe dual) et celle fournie par la nouvelle méthode (recherche d'un chemin eulérien sur le graphe initial). De plus il illustre les résultats fournis par le simulateur de fautes Mach 1000 et ce pour trois séquences de test différentes. La première est celle fournie par l'ancienne méthode dans laquelle les valeurs indéterminées sont fixées aléatoirement à "0" ou "1". Les deuxième et troisième séquences correspondent aux séquences de test fournies par la nouvelle méthode dans laquelle les valeurs indéterminées ont été fixées selon l'heuristique 1 et l'heuristique 2 respectivement. Pour ces trois séquences de test on donne le taux de couverture ainsi que le facteur de qualité.

	longueur normalisée		couverture de fautes en %			facteur de qualité de la séquence		
	ancienne méthode	nouvelle méthode	ancienne méthode	heuristique 1	heuristique 2	ancienne méthode	heuristique 1	heuristique 2
u802b	318	161	84.5	85.3	87.6	0.27	0.53	0.54
flammand	168	135	85.7	83.1	87.3	0.51	0.62	0.65
mouse	250	125	82	78	84	0.33	0.62	0.67
big3	139	111	88.8	67.5	72.5	0.64	0.61	0.65
traffic	122	100	92.6	80.8	86.3	0.76	0.81	0.86
s1	355	155	85.3	82.8	86.6	0.24	0.53	0.56
planet	628	245	92.3	90.1	95.2	0.15	0.37	0.39
hyeti	631	264	92.3			0.15		

Tableau 5. Longueur normalisée, taux de couverture et facteur de qualité

Les résultats montrent que pour toutes les machines considérées la couverture de fautes de la séquence donnée par l'heuristique 2 est meilleure que celle de la séquence donnée par l'heuristique1. Le tableau montre aussi que pour la plupart des machines le taux de couverture de la séquence générée par l'heuristique 2 est comparable à celui de la séquence fournie par l'ancienne méthode bien que cette dernière séquence soit plus longue que la précédente. Le facteur de qualité de ces séquences appuie cette conclusion. En effet le facteur de qualité de la séquence fournie par l'heuristique 2 est meilleur que celui de la séquence fournie par la première méthode. Ceci est vrai pour toutes les machines traitées sans exception.

V. 2. 5 Comparaison avec une séquence aléatoire de même longueur

Le but de cette comparaison est de montrer que notre séquence est meilleure qu'une séquence aléatoire et donc notre méthode ne peut être assimilée à une méthode de génération aléatoire.

Pour chacune des machines du tableau 3, une simulation de fautes a été effectuée par le simulateur HIFAULT et ce pour les deux séquences suivantes: la séquence générée par le parcours de graphe dans laquelle les entrées à valeurs indéterminées ont été fixées selon l'heuristique 2 (référenciée par Asyl) et une séquence aléatoire ayant la même longueur. Les résultats sont illustrés dans le tableau 6. Dans ce tableau on rappelle la longueur de la séquence de test générée par l'algorithme de parcours de graphe, le nombre total des fautes à injecter, le nombre des fautes équivalentes simulées, la couverture de fautes et le nombre des fautes non couvertes par chacune des deux séquences simulées.

				séquence aléatoire		séquence donnée par ASYL	
	longueur séquence	nombre de fautes	fautes équivalentes	couverture en %	fautes non détectées	couverture en %	fautes non détectées
bbara	90	342	87	46.5	48	97.4	3
cse	207	842	186	79.3	57	99.2	2
dk14	84	476	118	97.1	5	99.4	1
dk15	44	278	75	98.6	2	99.3	1
dk16	159	1244	235	98.6	7	99.4	2
dk17	63	286	69	95.1	5	99.0	1
sand	308	2810	495	90.4	92	97.0	35
styr	341	1846	336	88.4	78	98.9	8
tav	64	114	44	98.2	1	98.2	1
s1	167	1558	307	95.4	27	98.3	10
planet	284	2604	476	98.0	18	98.8	9

Tableau 6. Comparaison avec les résultats de simulation d'une séquence aléatoire

Le tableau 6 montre que la couverture d'une séquence générée en parcourant le graphe d'états d'une machine d'états finis est toujours meilleure que celle d'une séquence aléatoire de même longueur. On peut néanmoins constater la bonne qualité des séquences aléatoires. Il conviendrait donc de prendre une telle séquence comme référence dans toutes les études de génération de vecteurs de test.

VI Deuxième phase de la génération de test

Cette phase consiste à déterminer une deuxième séquence de test couvrant les fautes non détectées par la première séquence. L'approche proposée utilise le graphe des états de la machine juste et des machines fausses et doit être assistée d'un simulateur de fautes.

Notre méthode consiste à trouver une séquence de test capable de distinguer la machine juste de chacune des machines fausses et ceci pour toutes les fautes résiduelles. Cette approche est fondée sur la méthode de Poage. Cette méthode travaille sur le produit cartésien de la machine juste avec chacune des machines fausses où une machine fausse correspond à une faute résiduelle. L'avantage de la méthode de Poage est une bonne minimisation de la longueur de la séquence de test. Un inconvénient majeur de cette méthode est la grande place mémoire dûe à la taille des contrôleurs et au nombre des fautes considérées. En effet il faut stocker le modèle de la machine juste, celui de chacune des machines fausses et les produits cartésiens de la machine juste avec chacune des machines fausses. Pour pallier ce problème deux heuristiques sont étudiées et présentées dans §VI.1.2. Ces heuristiques réduisent considérablement la place mémoire nécessaire mais ne garantissent pas une minimisation aussi efficace de la longueur de la séquence de test.

Les paragraphes qui suivent considèrent le modèle de la machine de Mealy. Toutefois le modèle de la machine de Moore est aussi valable.

VI. 1 Approche de Poage

Elle consiste à générer des séquences de test qui distinguent la machine juste de l'ensemble des machines fausses.

Soit M la machine juste. Pour chacune des fautes résiduelles i, le modèle de la machine fausse Mi est construit. Cette construction est simple si les expressions des variables internes et des sorties primaires sont données pour le circuit en présence de la faute. La construction du modèle des machines fausses peut être aussi effectuée par simulation après injection des fautes au circuit.

Le produit cartésien est construit entre la machine juste M et chacune des machines fausses machine Mi. On rappelle que le produit cartésien d'une machine $M=(\Sigma, O, S, S0, \partial, \lambda)$ et d'une machine $Mi=(\Sigma, Oi, Si, S0, \partial i, \lambda i)$ est une machine définie par $M*Mi = (\Sigma, Opi, Spi,(S0,S0), \partial pi, \lambda pi)$ où $Opi = O \times Oi$, $Spi = S \times Si$ et

$\delta pi: Spi \times \Sigma \rightarrow Spi$ avec $\delta pi[(s, si), e] = (\delta(s, e), \delta i(si, e))$

$\lambda pi: Spi \times \Sigma \rightarrow Opi$ avec $\lambda pi[(s, si), e] = (\lambda(s, e), \lambda i(si, e))$

Une machine juste M est distinguée d'une machine fausse Mi s'il existe une sortie produit de M*Mi dans laquelle la sortie de M est différente de la sortie de Mi. En d'autres termes M est distinguable de Mi s'il existe un état courant produit spi $\in$ Spi et une entrée e $\in \Sigma$ tel que
$\lambda pi[spi, e] = \lambda pi[(s, si), e] = (\lambda(s, e), \lambda i(si, e))$ avec $\lambda(s, e) \neq \lambda i(si, e)$.

En fait le produit cartésien M*Mi n'est pas construit exhaustivement. La construction s'arrête lorsqu'une sortie produit permet de distinguer M de Mi.

Le produit cartésien général est construit exhaustivement à partir des produits cartésiens des machines M*Mi. Le produit cartésien général (M*M1)*(M*M2)* ... (M*Mn) des machines M*Mi (Σ, Opi, Spi,(S0,S0), ∂pi, λpi) est représenté par le 6-uplet (Σ, OP, SP,(S0,S0,...S0), δP, λP) où OP = Op1xOp2 ...Opn, SP = Sp1xSp2 ...Spn et tel que

δP: SP x $\Sigma \rightarrow$ SP avec

$\delta P[(sp1, sp2, ...spn), e] = (\delta p1(sp1, e), \delta p2(sp2, e), ... \delta pn(spn, e))$ et

λP: SP x $\Sigma \rightarrow$ OP avec

$\lambda P[(sp1, sp2, ...spn), e] = (\lambda p1(sp1, e), \lambda p2(sp2, e), ... \lambda pn(spn, e))$

Sur le produit général des machines, Poage cherche une séquence d'entrée minimale qui distingue la machine juste de chacune des machines fausses en utilisant une approche de minimisation de fonction booléenne.

Exemple

Soit une machine M à deux états, une variable d'entrée x et une variable de sortie [Cha68]. Soit quatre fautes résiduelles non détectées par la première phase. A chacune des fautes i correspond une machine fausse Mi.
Soit T, T1, T2, T3 et T4 les tableaux des états de la machine juste et des quatre machines fausses respectivement (tableau 7). Un tableau Ti représente les fonctions δi et λi. A partir d'un état courant (première colonne du tableau) et d'une valeur des entrées (première ligne du tableau), le terme état suivant / sortie est calculé. Pour chacune des machines Mi le calcul des termes du tableau Ti est effectué par un simulateur après injection de la faute i dans la structure du circuit.

T	x=0	x=1
0	0/0	1/0
1	1/0	0/1

(a) Machine M

T1	x=0	x=1
0	0/0	0/1

(b) Machine M1

T2	x=0	x=1
0	1/0	1/0
1	1/0	0/1

(c) Machine M2

T3	x=0	x=1
0	0/0	1/0
1	1/0	1/1

(d) Machine M3

T4	x=0	x=1
0	0/0	1/0
1	0/0	0/1

(e) Machine M4

Tableau 7. Tableaux des états des machines juste et fausses

A partir des tableaux des états ainsi obtenus Poage effectue les produits M*Mi de la machine juste M avec chacune des machines fausses Mi. Ceci est représenté par les tableaux des états produit T*Ti. Ces tableaux produit sont donnés dans le tableau 8. Un tableau produit T*Ti représente les fonctions produit δpi et λpi. La sortie produit est soulignée dans le cas où la sortie de la machine juste est différente de celle de la machine fausse.

T*T1	x=0	x=1
00	00/00	10/<u>01</u>

T*T2	x=0	x=1
00	01/00	11/00
01	01/00	10/<u>01</u>
11	11/00	00/11

T*T3	x=0	x=1
00	00/00	11/00
11	11/00	01/11
01	01/00	11/<u>01</u>

T*T4	x=0	x=1
00	00/00	11/00
11	10/00	00/11
10	10/00	01/<u>10</u>

Tableau 8. Produits cartésiens entre M et chacune des machines M1, M2, M3 et M4

Afin de simplifier l'écriture, on note A, B, C et D les états produit 00, 01, 11 et 10 respectivement.
Le tableau produit général est donné dans le tableau 9.

T*T1	T*T2	T*T3	T*T4	x = 0	x = 1
A	A	A	A	A B A A	* C C C
A	B	A	A	A B A A	* * C C
*	C	C	C	* C C D	* A B A
*	C	C	D	* C C D	* A B *
*	A	B	A	* B B A	* C * C
*	B	B	A	* B B A	* * * C

Tableau 9. Tableau produit général des tableaux produit T*Ti

Dans ce tableau un état produit général est une concaténation d'états "locaux" où le $i^{ème}$ état local est un état produit de M*Mi. Dans un état suivant général, l'état local

d'ordre i est remplacé par une étoile si la sortie correspondante dans M*Mi manifeste une différence entre la sortie de M et celle de Mi. Dans la première ligne de cette table par exemple, à l'état courant général AAAA et l'entrée x=1 correspond l'état suivant général *CCC. L'étoile rappelle que dans T*T1 l'état courant A et l'entrée x=1 permettent de manifester une différence dans la sortie produit de M*M1.
Sur ce tableau général on cherche la séquence minimale qui permet de distinguer la machine juste de chacune des machines fausses. Cela revient à chercher une séquence minimale de vecteurs d'entrée qui permet qu'une étoile associée à chacune des machines fausses se manifeste sur au moins un état général. On trouve la séquence minimale de test suivante.

vecteur 1 : 0
vecteur 2 : 1 M1 et M2 sont distinguées de M
vecteur 3 : 0
vecteur 4 : 1 M4 est distinguée de M
vecteur 5 : 1 M3 est distinguée de M

VI. 2 Méthode proposée

Elle a le même principe de base que celle de Poage. Au lieu de calculer exhaustivement les modèles des machines fausses, des machines produit et de la machine produit général et chercher dans un second temps une séquence de test sur le modèle produit général, notre méthode consiste à construire progressivement le produit cartésien de la machine juste avec l'ensemble des machines fausses au fur et à mesure du calcul de la séquence de test recherchée. Cette construction est effectuée par simulation parallèle de fautes. Elle n'est pas exhaustive. Elle s'arrête lorsque toutes les machines fausses sont distinguées de la machine juste.

Le produit cartésien d'une machine juste $M=(\Sigma, O, S, S0, \partial, \lambda)$ avec n machines fausses M1, M2,..Mi, ...Mn où $Mi=(\Sigma, Oi, Si, S0, \partial i, \lambda i)$, est noté M*M1*M2*...Mi*...Mn. Il est représenté par le 6-uplet

$(\Sigma, OP, SP, (S0,S0,...S0), \delta P, \lambda P)$ où

$OP = OxO1xO2 ...On,$

$SP = SxS1xS2 ...Sn$ et tel que

$\delta P: SP \times \Sigma \rightarrow SP$ avec

$\delta P[(s, s1, s2, ...sn), e] = (\delta(s, e), \delta 1(s1, e), \delta 2(s2, e), ... \delta n(sn, e))$ et

λP: SP x $\Sigma \rightarrow$ OP avec

$\lambda P[(s, s1, s2, ...sn), e] = (\lambda(s, e), \lambda 1(s1, e), \lambda 2(s2, e), ... \lambda n(sn, e))$.

Comme précédemment une machine juste M est distinguée de la machine Mi s'il existe un terme produit dans lequel la sortie de M est différente de celle de Mi. En d'autres termes M est distinguable de Mi s'il existe un état (s, s1, s2, ...sn) $\in$ SP et une entrée e $\in \Sigma$ tel que:

$\lambda P[(s, s1, s2, ...sn), e] = (\lambda(s, e), \lambda 1(s1, e), \lambda 2(s2, e), ... \lambda i(si, e), ...)$ avec
$\lambda(s, e) \neq \lambda i(si, e)$.

La recherche de la machine produit se fait pas à pas. Chaque pas détermine une transition de la machine produit définie par (entrée, état courant produit, état suivant produit, sortie produit). Le calcul de l'état suivant produit et de la sortie produit se fait par simulation parallèle après injection des fautes simultanément dans le circuit.
A partir d'une transition courante de la machine produit, la transition suivante est construite avec comme origine (état courant produit)

- soit le même état origine de la transition courante; on parlera de recherche en largeur.
- soit l'état destination (état suivant produit) de la transition courante; on parlera de recherche par profondeur.

Deux heuristiques ont été développées selon cette méthode. Elles emploient les deux types de recherche par largeur et par profondeur et fournissent en sortie la séquence de test. Lorsqu'un état produit est choisi comme état courant pour une nouvelle transition, il est marqué état visité. A chaque état courant visité est associée la liste des entrées visitées qui sont les entrées appliquées lors d'une recherche par largeur à partir de cet état courant.
On dit que l'exploration d'un état est saturée si toutes les transitions possibles à partir de cet état ont été explorées ce qui veut dire que la liste des entrées visitées associées à cet état est complète.

VI. 2. 1 Heuristique 1

Etant donné une transition ayant permis de distinguer la machine juste d'une machine fausse, le but de cette heuristique est de repartir de l'état destination de la transition, pour chercher des nouvelles entrées qui permettent de distinguer les machines fausses résiduelles de la machine juste.

A partir de l'état initial l'algorithme s'effectue comme suit.

(1) calcul d'une transition. C'est la transition courante.
(2) **Si** aucune différence n'apparaît sur la sortie de la transition courante entre la sortie de la machine juste et celle d'au moins une des machines fausses **alors**
l'état courant de la transition courante devient l'état courant de la transition suivante,
aller à (4) pour continuer la recherche par largeur.
fin_si
(3) **Sinon** une différence apparaît **alors**
le vecteur d'entrée de la transition courante est un vecteur de test,
l'état destination de la transition devient l'état courant de la transition suivante,
aller à (4) pour commencer une recherche par largeur.
fin_sinon

(4) **Si** l'exploration de l'état courant n'est pas saturée **alors**
aller à (1) pour continuer (si on vient de (2)) ou commencer (si on vient de (3)) la recherche en largeur à partir de l'état courant.
fin_si
Sinon l'exploration de l'état courant est saturée **alors**
(5) **Si** existe un état parmi les états destination des transitions explorées en largeur à partir de l'état courant, pour lequel l'exploration n'est pas saturée **alors**
cet état est choisi pour être l'état courant de la transition suivante
aller à (1).
fin_si
(6) **Sinon** on ne peut plus appliquer la recherche en largeur ni à partir de l'état courant ni à partir d'aucun de ses états suivants **alors**
un retour arrière est nécessaire pour changer un choix de l'état courant effectuer au pas (5) ou pour continuer une recherche par largeur arrêtée au pas (3).
fin_sinon
fin_sinon

Au cours d'un retour arrière la séquence des vecteurs d'entrée qui n'a permis aucune distinction entre la machine juste et une des machines fausses est abandonnée.

Exemple

Prenons l'exemple de la machine M ayant 2 états, une entrée x et une sortie représentée par son tableau des états donné dans le tableau 7a. Dans la suite les lettres a et b désignent respectivement les états 0 et 1 de la machine.

Soit f1, f2, f3 et f4 les quatre fautes résiduelles de la première phase de test. La machine produit M*M1*M2*M3*M4 est construite en dix pas (figure 10). Sur cette figure on note les états produit. Les sorties produit ne sont pas notées explicitement. Lorsqu'une étoile apparaît sous un état de la machine fausse Mi ceci veut dire que la faute i est manifestée sur la sortie ce qui permet de distinguer Mi de M.

L'algorithme commence par définir un fichier de fautes à donner au simulateur. Ce fichier contient la liste des quatre fautes. On lance une première simulation avec le vecteur d'entrée d'initialisation "init". On est dans l'état initial a a a a a. La liste des vecteurs de test comporte le vecteur d'initialisation "init".

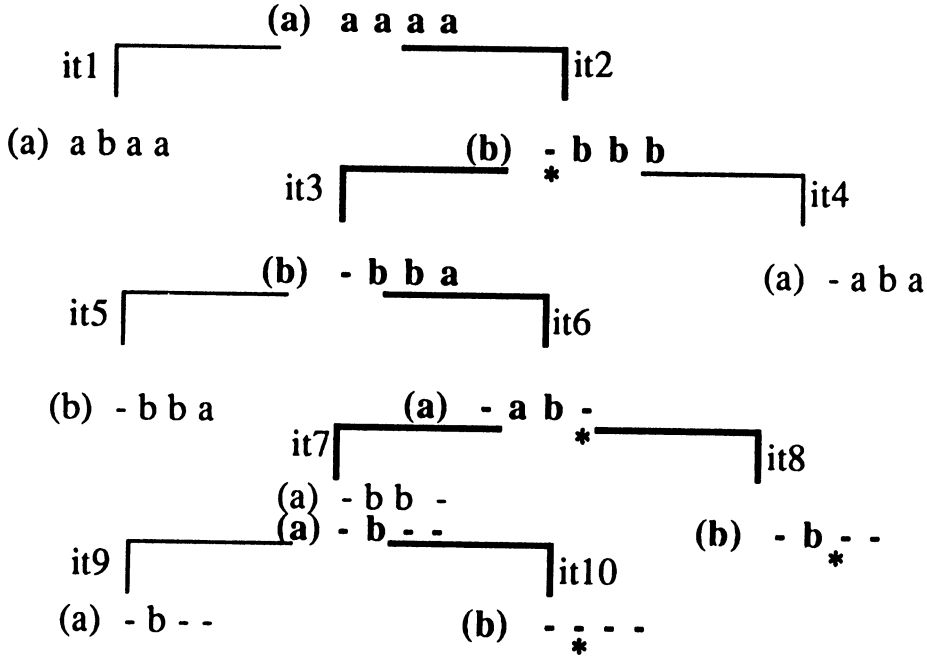


Figure 10. Machine produit M*M1*M2*M3*M4 de l'heuristique 1.

Au premier pas on applique l'entrée x=0 à partir de l'état courant produit a a a a a qui est marqué état visité. L'état suivant produit est a a b a a.

Puisque aucune machine n'est distinguée le deuxième pas est effectué à partir du même état courant a a a a a en appliquant l'entrée x=1. L'état suivant produit est b - b b b et M1 est distinguée. Dans la suite de l'algorithme la machine M1 n'est plus simulée. Pratiquement ceci se traduit par une soustraction de la faute f1 du fichier des fautes à donner au simulateur. Sur la figure 10 ceci est illustré par la présence du tiret (-) dans l'état produit b - b b b.

Puisque le pas 2 a permis de manifester une faute sur la sortie, l'état suivant produit (non encore visité) est pris comme état courant produit du pas 3. Il est marqué état visité.
Le pas 4 est effectué à partir du même état courant produit que celui du pas 3 car ce dernier ne permet de distinguer aucune machine fausse. L'état suivant produit du pas 4 ne permet lui non plus aucune distinction. Le pas 5 choisit son état courant produit parmi les états [b - b b a, a - a b a] suivants produit non visités des pas 3 et 4. Soit b - b b a l'état suivant produit choisi.
Le pas 5 est effectué suivi du pas 6 à partir du même état courant produit b - b b a qui est marqué état visité.
Le pas 6 permet de distinguer la machine M4. La faute f4 est retranchée du fichier des fautes du simulateur.
L'état suivant produit du pas 6 a - a b - (non encore visité) est choisi pour être l'état courant produit des pas 7 et 8.
Le pas 8 distingue la machine M3. La faute f3 est retranchée du fichier des fautes du simulateur. L'état suivant produit b - b - - ne peut pas être pris comme état courant produit du pas suivant car son exploration est déjà saturée ainsi que ses états suivants [b - b - - , a - a - -].
Puisque la recherche en largeur correspondant à l'état courant a - a b - est terminée le pas 9 choisit, par retour arrière, l'état suivant produit du pas 7 (qui est un état non encore visité) son état courant produit. Il est marqué état visité. Sur le graphe l'état courant produit comporte un tiret à la place de l'état de la machine 3 puisqu'elle a été déjà distinguée.
Le pas 9 est effectué. Il est suivi du pas 10 qui permet de distinguer la machine M2. Comme toutes les machines ont été distinguées l'algorithme s'arrête. La séquence de test trouvée est [init, 1, 0, 1, 1, init, 1, 0, 1, 0, 1].

VI. 2. 2 Heuristique 2

Le but de cette heuristique est de donner priorité à la manifestation sur les sorties d'une faute déjà manifestée sur un état de la machine produit (état contaminé).
A partir de l'état initial l'algorithme s'effectue comme suit.

Si l'état courant n'est pas contaminé par une faute **alors**
(1) calcul d'une transition. C'est la transition courante.
Si l'état destination de la transition n'est pas contaminé par une faute **alors**
(2) l'état courant de la transition courante devient l'état courant de la transition suivante,

aller à (4) pour continuer la recherche par largeur.

fin_si

(3) **Sinon** l'état destination de la transition courante est contaminé par une faute i **alors**

l'état destination de la transition devient l'état courant de la transition suivante,

aller à (7) pour essayer de manifester la faute sur une sortie.

fin_sinon

fin_si

Sinon l'état courant est contaminé par une faute i et l'algorithme cherche à manifester la faute sur une sortie **alors**

(7) **tant que** l'exploration de l'état courant n'est pas saturée **faire**

calculer toutes les transitions possibles à partir de l'état courant (recherche en largeur).

fait

Si existe une transition telle que sa sortie distingue la machine juste de la machine fausse Mi **alors**

le vecteur d'entrée de cette transition est un vecteur de test,

(8) **Si** existe une transition telle que l'état destination est contaminé par une faute j≠i **alors**

l'état destination devient l'état courant,

aller à (7) pour essayer de tester la nouvelle faute j.

fin_si

Sinon aucune des transitions calculées n'a l'état destination contaminé par une faute j≠i **alors**

aller à (5) pour choisir un état destination non saturé.

fin_sinon

fin_si

Sinon aucune des transitions calculées ne permet de distinguer la machine juste de la machine fausse Mi **alors**

Si existent transitions telle que leur état destination est contaminé par la faute i **alors**

(9) choisir un des états destination pour être l'état courant de la transition suivante,

aller à (7) pour réessayer de tester la faute i.

fin_si

Sinon aucune transition n'a son état destination contaminé par la faute i **alors**

retour arrière pour changer, s'il y a lieu, le choix effectué dans 9. Sinon on laisse tomber la faute i pour le moment.
aller à (8) pour s'intéresser au testd'une autre faute j.

fin_sinon

fin_sinon

fin_sinon

(4) **Si** l'exploration de l'état courant n'est pas saturée **alors**

aller à (1) pour continuer la recherche en largeur à partir de l'état courant.

fin_si

Sinon l'exploration de l'état courant est saturée **alors**

(5) **Si** existe un état parmi les états destination des transitions explorées en largeur à partir de l'état courant, pour lequel l'exploration n'est pas saturée **alors**

cet état est choisi pour être l'état courant de la transition suivante
aller à (1).

fin_si

(6) **Sinon** on ne peut plus appliquer la recherche en largeur ni à partir de l'état courant ni à partir d'aucun de ses états suivants **alors**

un retour arrière est nécessaire pour changer un choix de l'état courant effectuer au pas (5) ou pour continuer une recherche par largeur arrétée au pas (3).

fin_sinon

fin_sinon

Exemple

Reprenons l'exemple de la machine M traité par l'heuristique 1. Dans cette heuristique la machine produit M*M1*M2*M3*M4 est construite en huit pas (figure 11). Sur la figure 11 on note les états produit. Les sorties produit ne sont pas notées explicitement. Lorsqu'une étoile apparaît au dessous un état de Mi ceci veut dire que la faute i est manifestée sur la sortie ce qui permet de distinguer Mi de M. Dans le cas où l'étoile apparaît au dessus d'un état de Mi ceci veut dire que la faute i est manifestée sur l'état c.à.d que l'état de Mi est contaminé par la faute i.

L'algorithme commence par définir un fichier de fautes à donner au simulateur. Ce fichier contient la liste des quatre fautes. On lance une première simulation avec le vecteur d'entrée d'initialisation "init". On est dans l'état initial a a a a a.

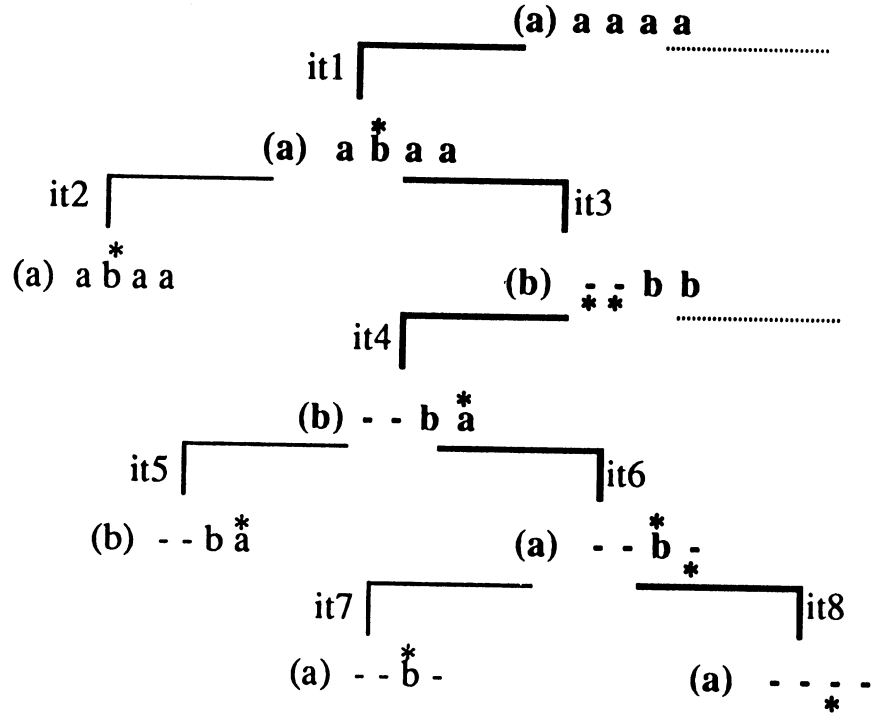


Figure 11. Machine produit M*M1*M2*M3*M4 de l'heuristique 2.

Au premier pas on applique l'entrée x=0 à partir de l'état courant produit a a a a a. L'état suivant produit est a a b a a. L'état de la machine M2 est corrompu par la faute f2. L'algorithme considère cette faute et cherche à la manifester sur la sortie. Les deuxième et troisième pas sont effectués à partir de l'état suivant a a b a a contaminé par f2, en appliquant respectivement les entrées x=0 et x=1. L'état suivant produit du pas 3 permet de manifester les fautes f1 et f2 sur la sortie. Dans la suite de l'algorithme les machines M1 et M2 ne sont plus simulées. Sur la figure 11 ceci est illustré par la présence des tirets (-) dans l'état produit b - - b b.
Le pas 4 cherche parmi les états suivants produit des pas 2 et 3 [a - - a a, b - - bb] l'état qui manifeste une faute différente de f1 et f2. Puisque l'état recherché n'existe pas alors l'algorithme cherche dans cette liste un état dont l'exploration ne soit pas saturé. L'état a - - a a est déjà visité et toutes les valeurs possibles de l'entrée ont été considérées. C'est donc b - - b b qui est l'état choisi pour être l'état courant produit du pas 4 qui s'effectue avec une entrée x=0. L'état suivant produit b - - b a manifeste la faute f4. La recherche en largeur s'arrête.

Les pas 5 et 6 sont effectués avec l'état courant produit b - - b a et les entrées x=0 et x=1 respectivement. L'état suivant produit du pas 6 manifeste la faute 4 sur la sortie.
L'algorithme cherche parmi les états suivants produit des pas 5 et 6 l'état qui manifeste une faute résiduelle. C'est l'état suivant a - - b - du pas 6 qui est l'état recherché puisqu'il manifeste la faute f3 sur l'état de la machine M3. Cet état est alors pris comme état courant produit des pas 7 et 8.
Le pas 8 manifeste la faute f3 sur la sortie.
Comme toutes les machines ont été distinguées l'algorithme s'arrête. La séquence résultante de test est [init, 0, 1, 0, 1, 1].

VI. 2. 3 Implantation et résultats

La première heuristique a été implantée en langage C. Elle a été testée sur la machine "Traffic" dont la couverture est passée à 100 % après ajout de 3 vecteurs supplémentaires à la première séquence fournie par la première phase.
Toutefois des problèmes d'interface existent dans le système HIFAULT. Par suite cette heuristique n'est pas encore automatisée dans sa partie du dialogue avec HIFAULT. Actuellement le simulateur est utilisé en mode interactif ce qui rend impossible le traitement des exemples.
La deuxième heuristique est encore en phase de mise au point. Cependant elle a été appliquée "manuellement" sur les exemples présentés dans le tableau 3. Les résultats sont illustrés sur les tableaux du § (VI. 3) ci dessous.

VI. 2. 4 Heuristique d'accélération de la deuxième phase

Elle a pour but d'exploiter au maximum la première séquence pour générer la seconde séquence de détection des fautes résiduelles.
Elle consiste à dépouiller le résultat de la simulation de la première phase pour identifier les fautes qui contaminent les états de la machine. La séquence qui manifeste chacune de ces fautes i est ainsi connue. Il reste à chercher la séquence qui permet de manifester cette faute i sur la sortie. Ceci est résolu en appliquant l'heuristique 2 de l'approche précédente et effectuant une simulation parallèle sur les deux machines M et Mi.
Toutefois cette heuristique est inefficace pour générer les séquences de test des fautes résiduelles qui ne contaminent pas au moins un état de la machine. Pour détecter ces fautes on a recours à l'une des deux heuristiques présentées ci-dessus. Cette heuristique est considérée comme une préphase qui vise à accélérer la deuxième phase puisqu'elle prend en charge la détection d'un ensemble des fautes résiduelles de la première phase.

L'algorithme correspondant à cette heuristique s'effectue en trois pas et ceci pour chacune des fautes i.
pas1. à partir de la première séquence extraire la séquence minimale qui manifeste la faute sur un état de la machine.
pas2. appliquer l'heuristique 2 avec un état courant produit de M*Mi égal au produit de l'état de la machine juste par l'état contaminé trouvé.
pas3. concaténer les deux séquences trouvées dans les deux pas précédents.

Dans le premier pas on considère la séquence I des vecteurs d'entrée et la séquence S des états correspondants. Soit S = [S0, s_1, ..., s_k, ..., sf]. Soit s_k le premier état de cette séquence qui est contaminé par la faute k. La plus courte séquence S_k allant de S0 à s_k est obtenue en supprimant toutes les boucles qui se trouvent entre S0 et s_k. Soit S_k= [S0, ..., sk]. A cette séquence réduite S_k exprimée en termes d'états correspond la séquence réduite I_k recherchée exprimée en termes de vecteurs d'entrée.
Puisque les premiers k+1 états de S = [S0, s_1, ..., s_k[ne sont pas contaminés par la faute k alors les états de S_k = [S0, ..., sk[ne sont pas non plus contaminés par la faute k.

Exemple
Soit une machine représentée par son graphe d'états de la figure 12. Soit la séquence de test trouvée par la première phase I=[i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11] et la séquence correspondante en termes d'états S=[S0, s1, s2, s3, s4, s5, s1, s2, s3, s6, s7, S0].
Soit k une faute qui n'est pas détectée par la séquence S mais qui se manifeste sur l'état final s6.

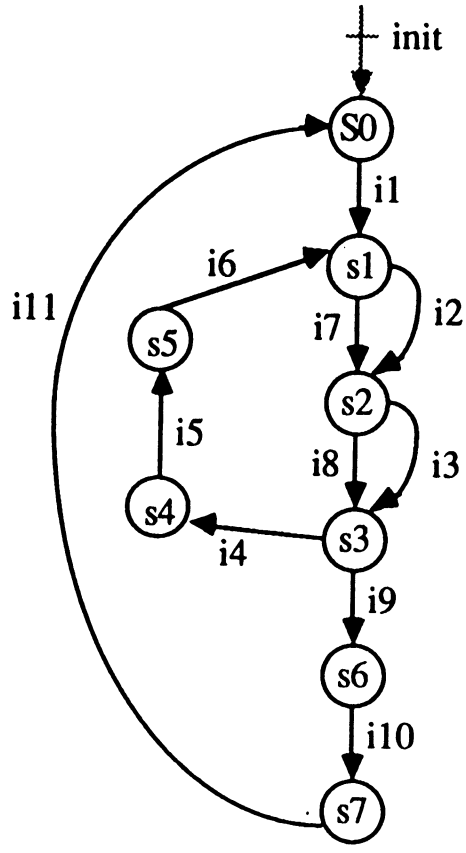


Figure 12. Exemple de machine

Le premier pas de l'algorithme cherche la plus courte séquence des vecteurs d'entrée I_k qui permet de manifester la faute k sur l'état s6. Pour cela on cherche la plus courte séquence S_k de S allant de S0 à s6. On détecte la boucle [s3, s4, s5, s1, s2, s3] entre S0 et s6. La séquence S_k est calculée à partir de la séquence [S0, s1, s2, s3, s4, s5, s1, s2, s3, s6] de laquelle on retranche la boucle trouvée. On obtient la séquence S_k=[S0, s1, s2, s3, s6]. La séquence I_k recherchée est celle qui correspond à S_k exprimée en termes de vecteurs d'entrée. Soit I_k=[i1, i2, i3, i9]. Cette séquence permet de manifester la faute k sur s6. Elle ne contamine pas les états [S0, s1, s2, s3] prédécesseurs de s6 dans S_k.

Implantation et résultats expérimentaux de l'heuristique d'accélération

Cette heuristique est en cours d'inplantation en langage C. Toutefois, elle a été testée "manuellement" sur les machines présentées dans le tableau 3.

Une simulation de fautes a été effectuée avec la première séquence de test fournie par l'algorithme de parcours du graphe de contrôle après avoir injecté les fautes non détectées par la première phase et avoir mis en sortie du circuit, les sorties des bascules d'états. Dans les deuxième et troisième colonnes, le tableau 10 rappelle la couverture donnée par la première séquence générée par la première phase et le

nombre des fautes résiduelles après simulation de cette première séquence. Dans les quatrième et sixième colonnes il montre les résultats de la simulation en indiquant le nombre des fautes qui ont contaminé un état et la longueur de la séquence supplémentaire permettant de détecter ces fautes. Par ailleurs ce tableau donne la nouvelle couverture de faute après application de l'heuristique d'accélération. Au cours de cette heuristique, la simulation a été effectuée en mode interactif pour la recherche de la séquence qui permet de manifester une faute sur les sorties primaires.

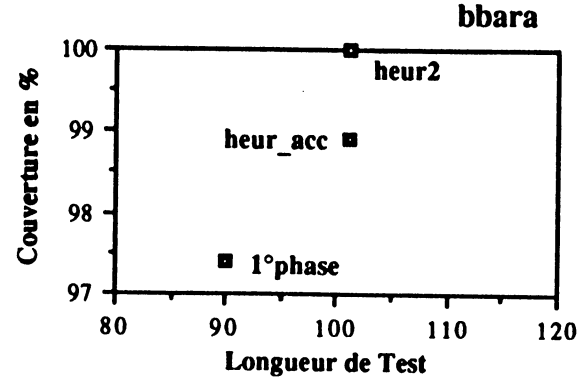
	Résultats de la première phase			Résultats de l'heuristique d'accélération		
	Couverture en %	nombre des fautes résiduelles	nombre des fautes contaminant un état	nombre des fautes detectées	séquence supplément.	Couverture en %
bbara	97.4	3	2	2	11	98.9
cse	99.2	2	0	0	-	
dk14	99.4	1	0	0	-	
dk15	99.3	1	0	0	-	
dk16	99.4	2	0	0	-	
dk17	99.0	1	0	0	-	
sand	97.0	35	0	0	-	
styr	98.9	8	3	3	27	99.24
tav	98.2	1	0	0	-	
s1	98.3	10	0	0	-	
planet	98.8	9	1	1	22	98.9

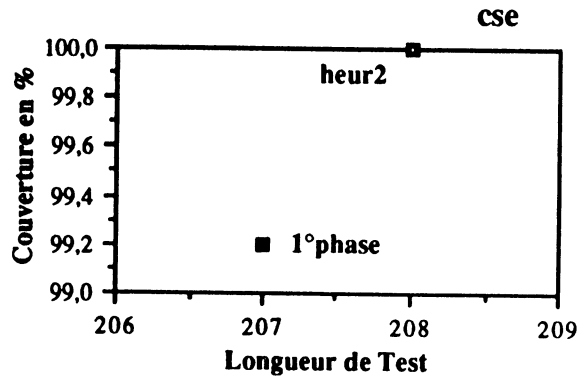
Tableau 10. Résultats de l'heuristique d'accélération de la deuxième phase

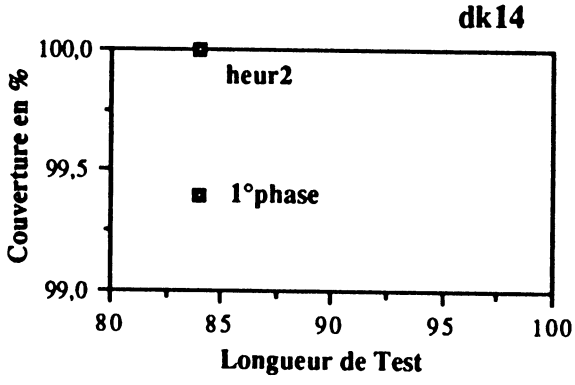
VI. 3 Résultats expérimentaux de la deuxième phase

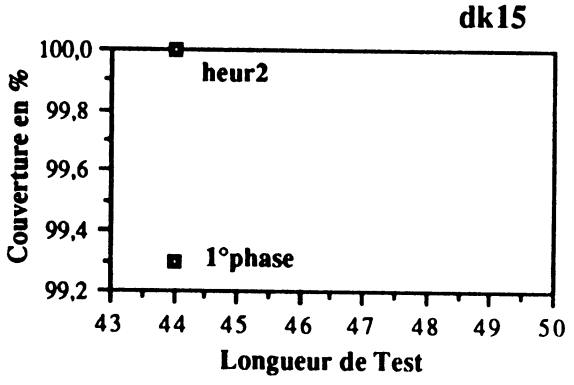
La deuxième phase a été appliquée "manuellement" sur les machines d'états finis présentées dans le tableau 3. Les tableaux ci-dessous illustrent le taux de couverture en fonction de la longueur de la séquence de test. Cette séquence est la concaténation de trois séquences. La première est la séquence déterminée par l'algorithme de parcours de graphe. La seconde est calculée par l'heuristique d'accélération (référenciée par heur_acc) fondée sur les résultats de simulation de la première séquence. La troisième séquence est fournie par l'heuristique 2 (référenciée par heur2) pour les fautes non détectées ni par la première phase ni par l'heuristique

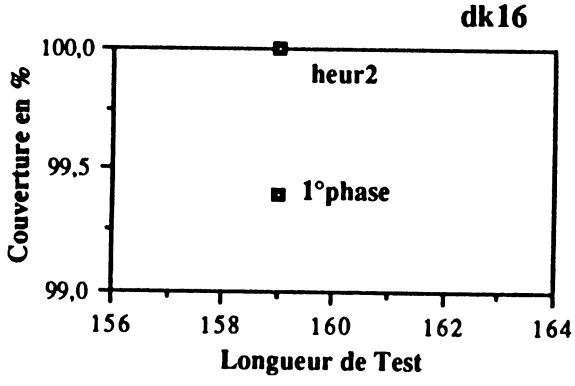
d'accélération. L'heuristique 2 n'a pas pu être appliquée sur les exemples "styr", "s1" et "planet" vu le grand nombre de simulations à lancer depuis le clavier.

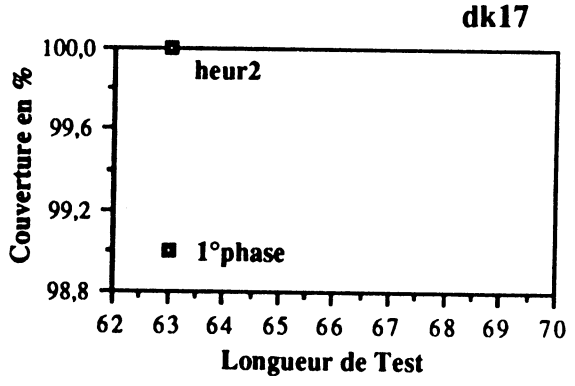


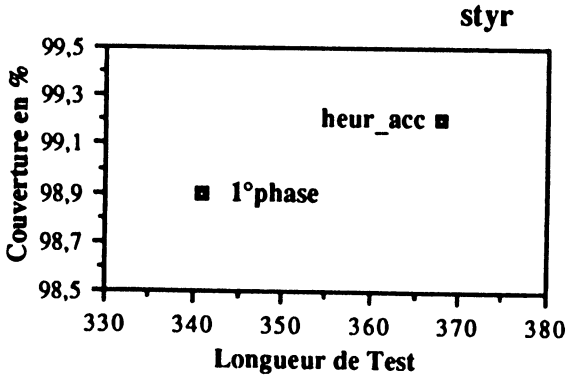


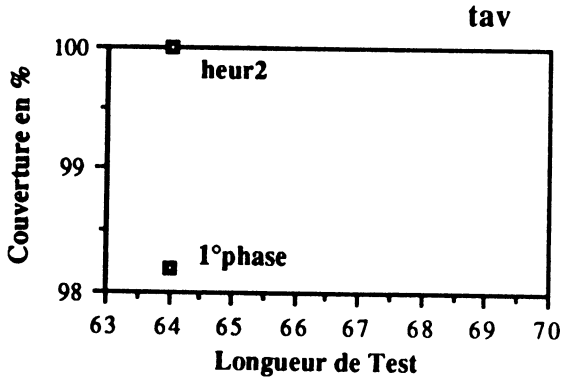


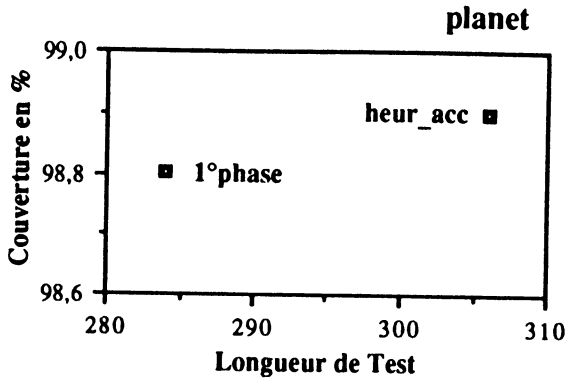












VI. 4 Comparaison entre les résultats d'Asyl et ceux de [Dev87] et [Che90]

Il est important de noter qu'une comparaison entre ces trois méthodes n'est pas facile vu que l'implantation physique en cellules standards des circuits traités dans [Che90] et [Dev87] ne nous est pas connue. Comme la couverture est exprimée en termes de collage sur des connexions, il peut donc y avoir des variations sur cette couverture. Pour pallier ce problème, nous avons réalisé deux circuits à l'aide de deux outils de CAO totalement différents (COMPASS, ASYL). Le premier est réalisé avec des cellules standards de la bibliothèque de VLSI Technology (CMOS 1µ2). Pour le second cette bibliothèque a été réduite aux portes And, Nand, Or Nor, Inverseur et bascules D pour se placer dans les cas les plus défavorables (connexions et collage nombreux).

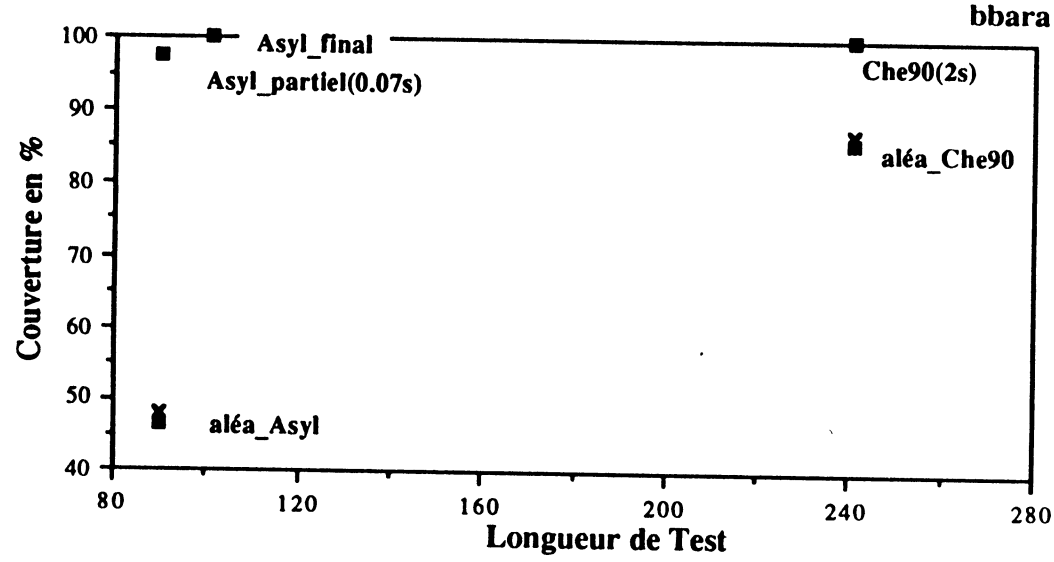
Les résultats de la comparaison sont illustrés sur les tableaux suivants et ce pour les machines présentées dans le tableau 3. Pour chacune de ces machines on note le taux la couverture de faute tel qu'il a été donné dans [Che90] et [Dev87] d'une part et celui donné après application de la séquence générée par l'algorithme de la première phase de notre méthode noté Asyl_partiel et celle générée après l'application de l'algorithme de la deuxième phase noté Asyl_final. De plus ces tableaux donnent le taux de couverture des séquences aléatoires de même longueur que les séquences fournies par les trois méthodes.

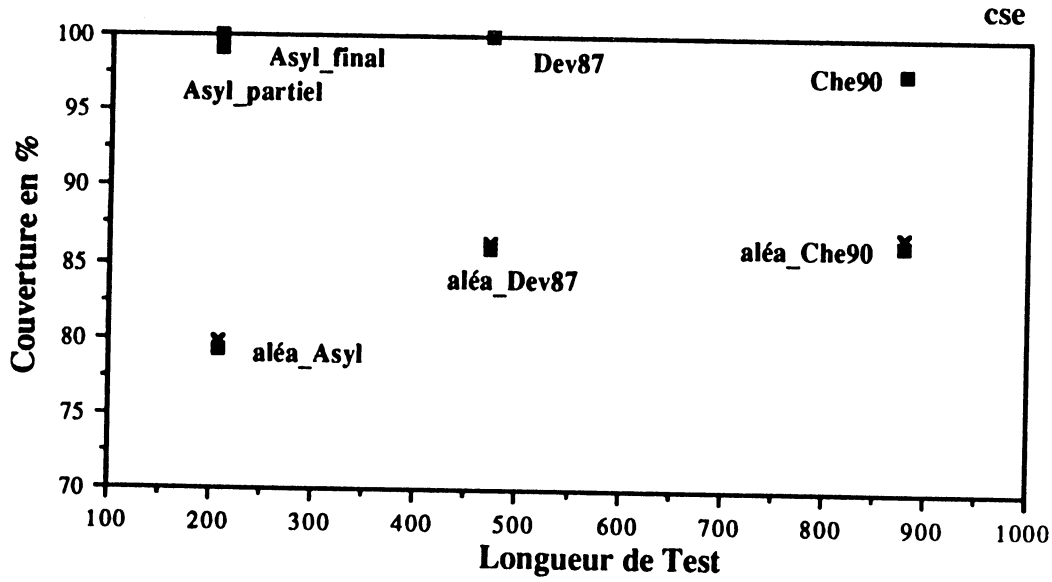
Ces tableaux montrent que la séquence de test générée par Asyl_final fournit un taux de couverture de 100% après soustraction des fautes redondantes. Cette séquence est au moins deux fois moins longue que celles générées par [Che90] et [Dev87].

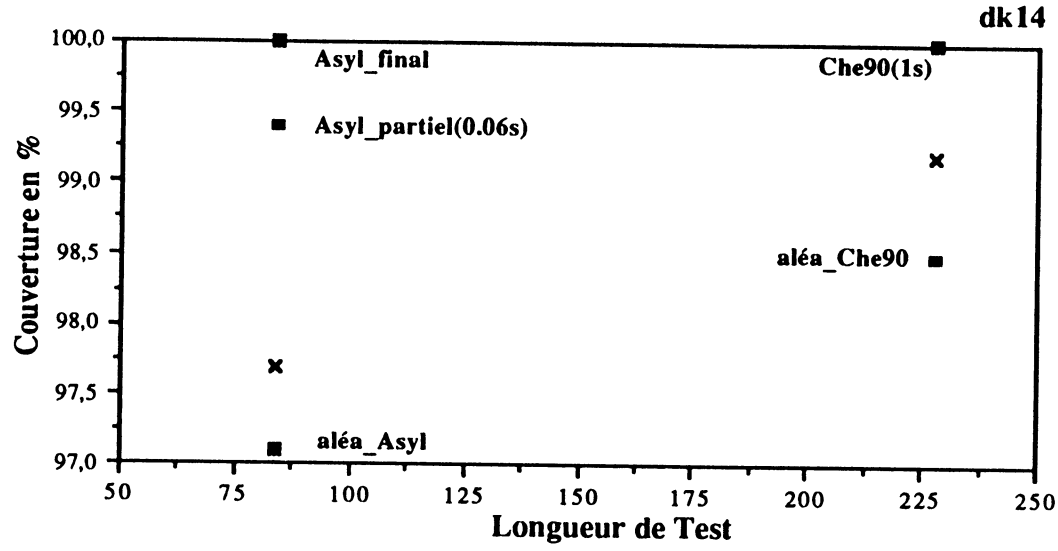
Remarque importante

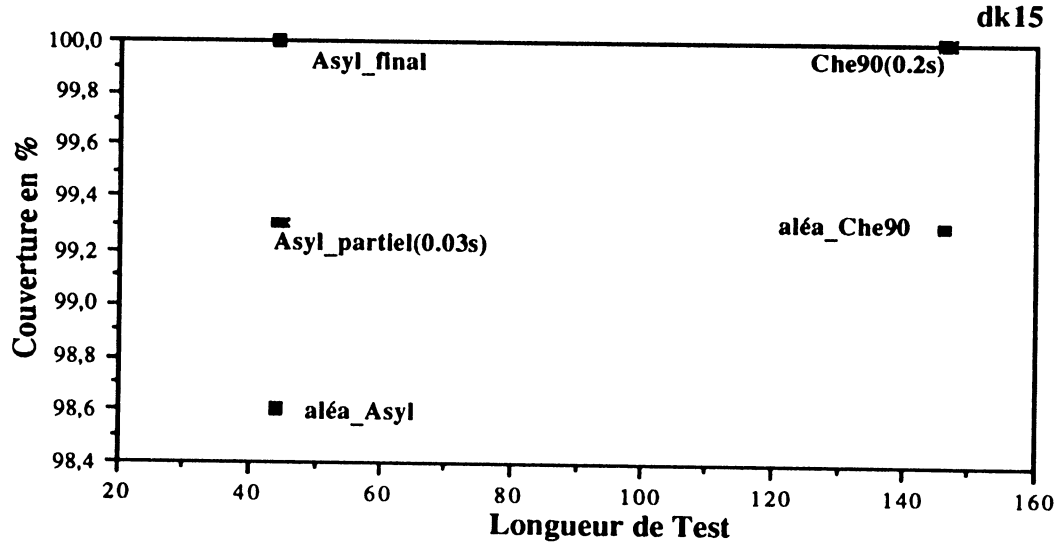
Les couvertures données par les méthodes Asyl_final, [Che90] et [Dev87] sont estimées par rapport aux fautes détectables; les fautes non détectables (redondantes)

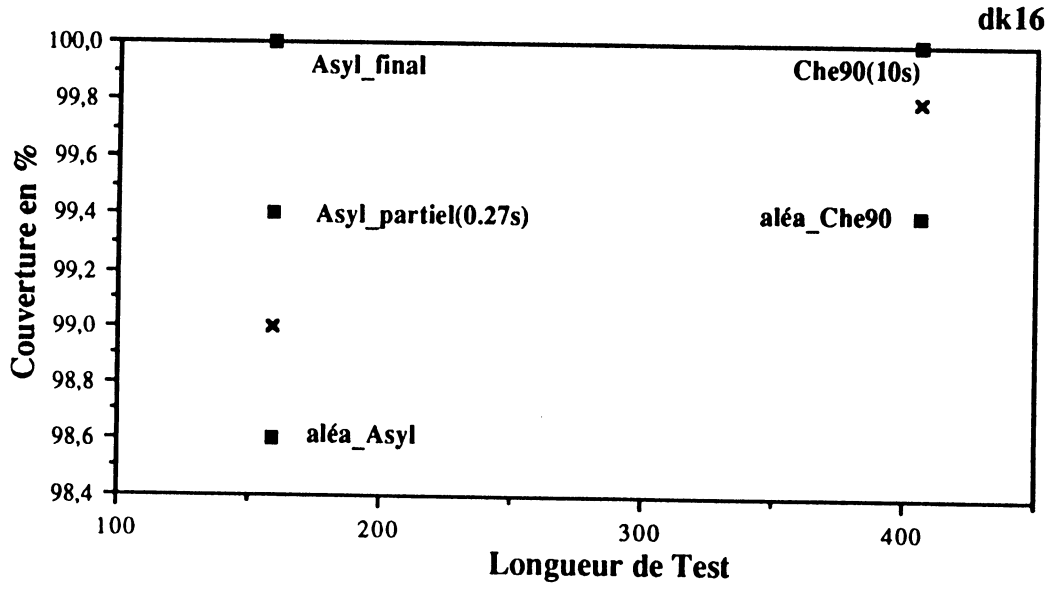
ayant été elliminées. Par contre l'estimation sur les séquences aléatoires ainsi que sur Asyl_partiel a été effectuée sur l'ensemble des fautes, incluant les fautes non détectables. Ceci veut dire en fait que la couverture des séquences aléatoires / fautes détectables est meilleure que ce celle qui est illustrée. Pour les exemples où les fautes non détectables ont pu être identifiées, un point corrigé illustré par une croix montre la couverture du test aléatoire / fautes détectables.
On s'aperçoit donc à nouveau de la concurrence importante entre les méthodes de génération de test fonctionnel et celle du test aléatoire.

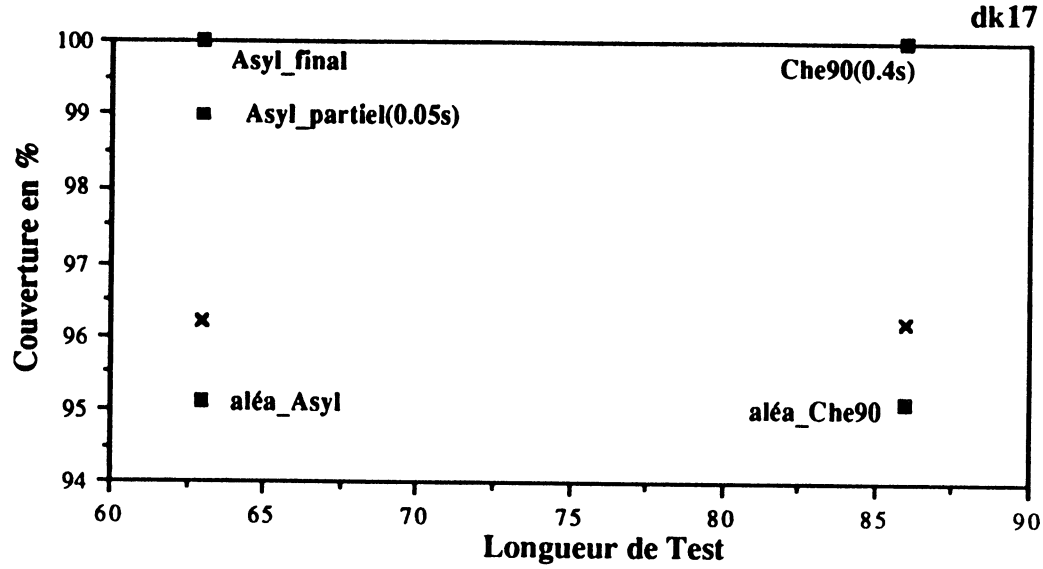


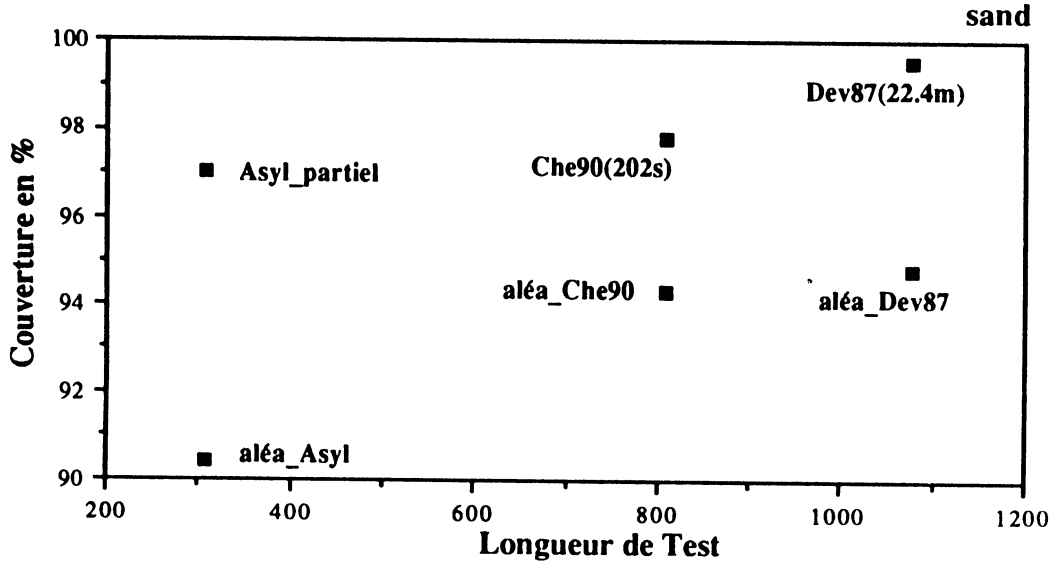


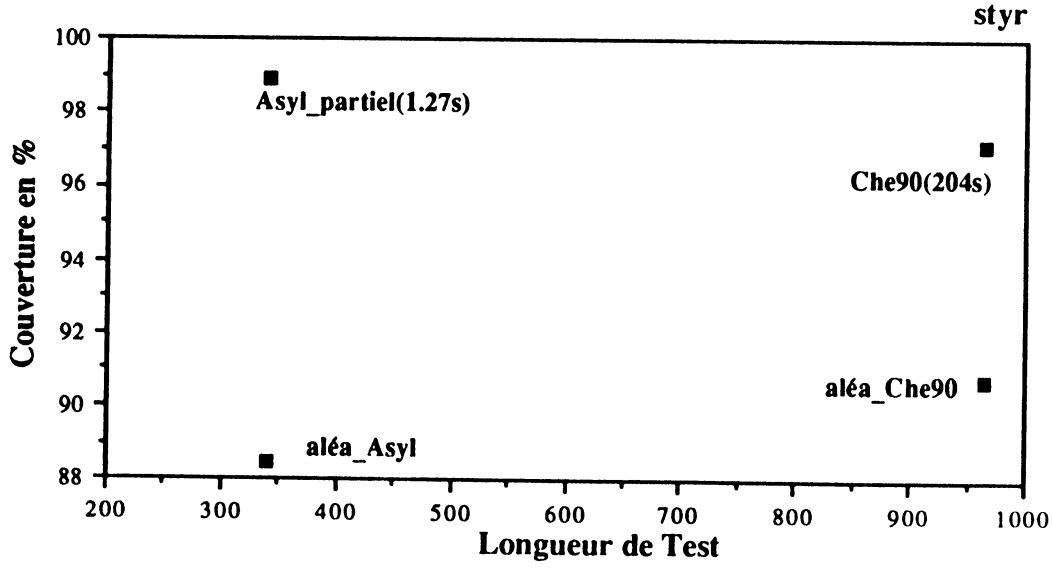

dk14
Asyl_final
Che90(1s)
Asyl_partiel(0.06s)
aléa_Che90
aléa_Asyl
Couverture en %
Longueur de Test
100,0
99,5
99,0
98,5
98,0
97,5
97,0
50
75
100
125
150
175
200
225
250

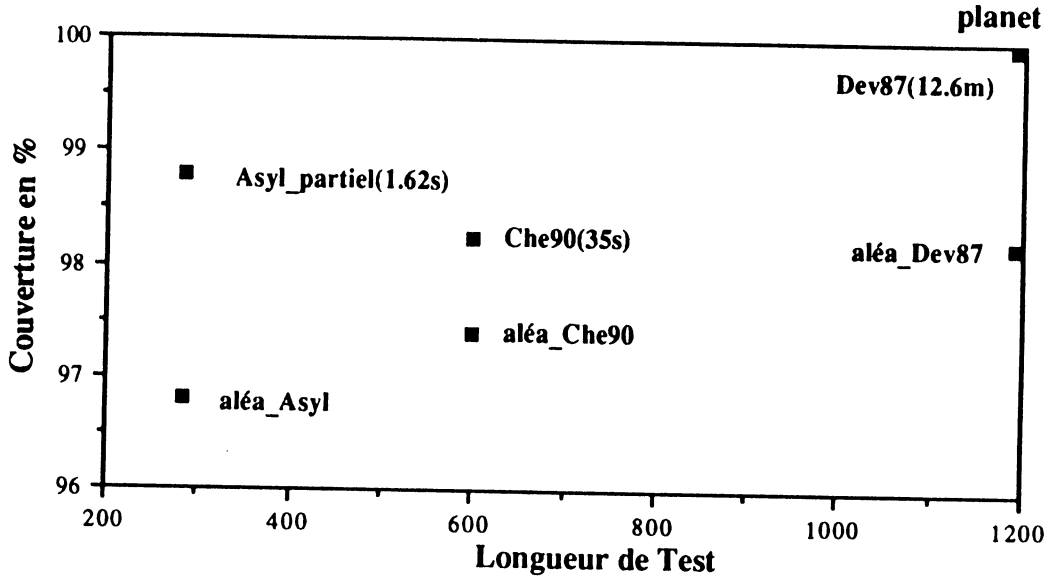

dk15
Asyl_final
Che90(0.2s)
Asyl_partiel(0.03s)
aléa_Che90
aléa_Asyl
Couverture en %
Longueur de Test
100,0
99,8
99,6
99,4
99,2
99,0
98,8
98,6
98,4
20
40
60
80
100
120
140
160


dk16
Asyl_final
Che90(10s)
Asyl_partiel(0.27s)
aléa_Che90
aléa_Asyl
Couverture en %
Longueur de Test
100,0
99,8
99,6
99,4
99,2
99,0
98,8
98,6
98,4
100
200
300
400

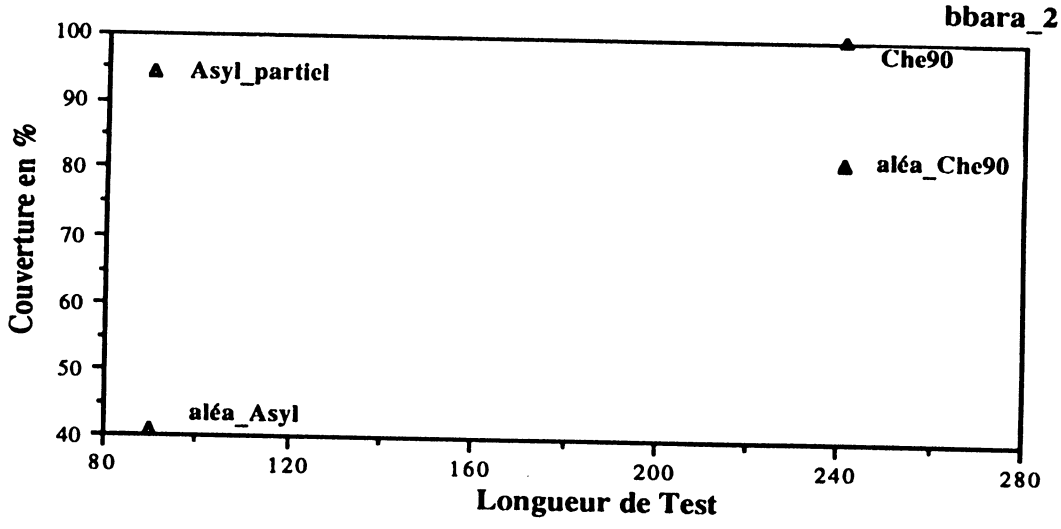

dk17
Asyl_final
Asyl_partiel(0.05s)
aléa_Asyl
Che90(0.4s)
aléa_Che90
Couverture en %
Longueur de Test
100
99
98
97
96
95
94
60
65
70
75
80
85
90

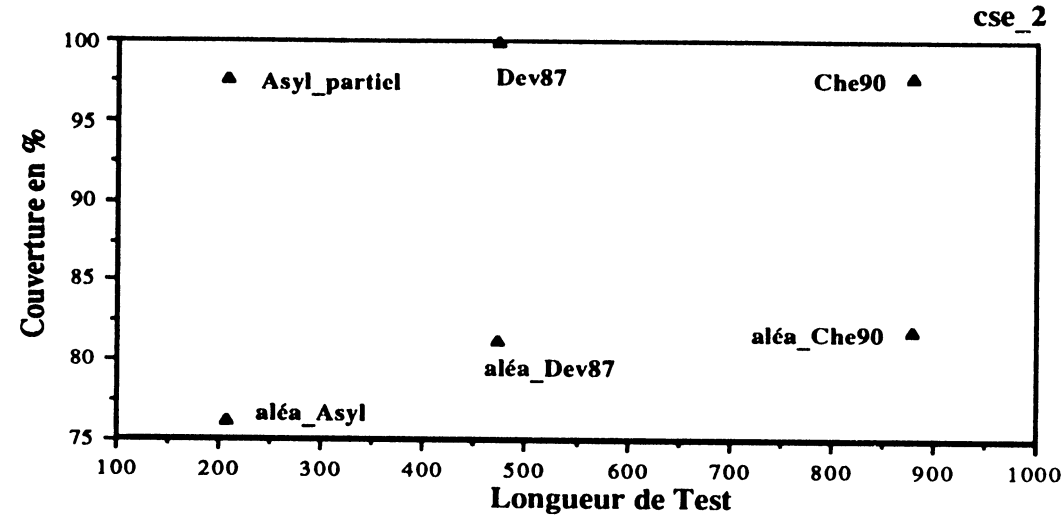

sand
Asyl_partiel
aléa_Asyl
Che90(202s)
aléa_Che90
Dev87(22.4m)
aléa_Dev87
Couverture en %
Longueur de Test
100
98
96
94
92
90
200
400
600
800
1000
1200

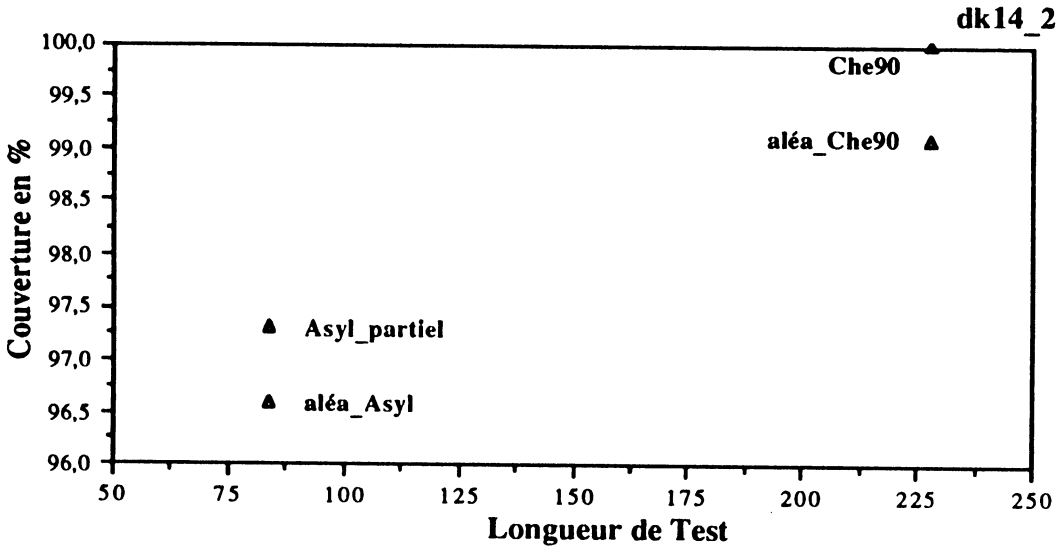

styr
Asyl_partiel(1.27s)
aléa_Asyl
Che90(204s)
aléa_Che90
Couverture en %
Longueur de Test
100
98
96
94
92
90
88
200
300
400
500
600
700
800
900
1000

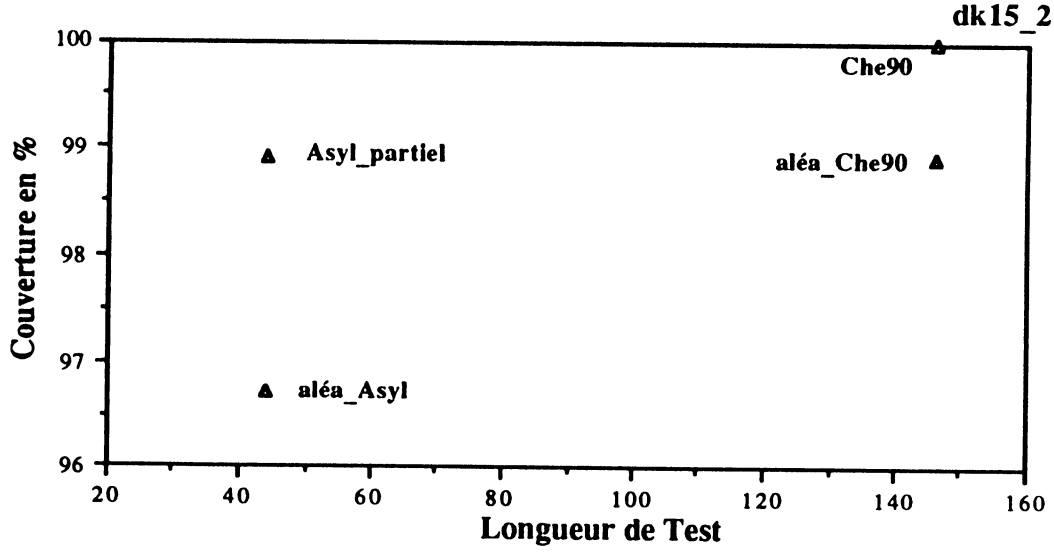


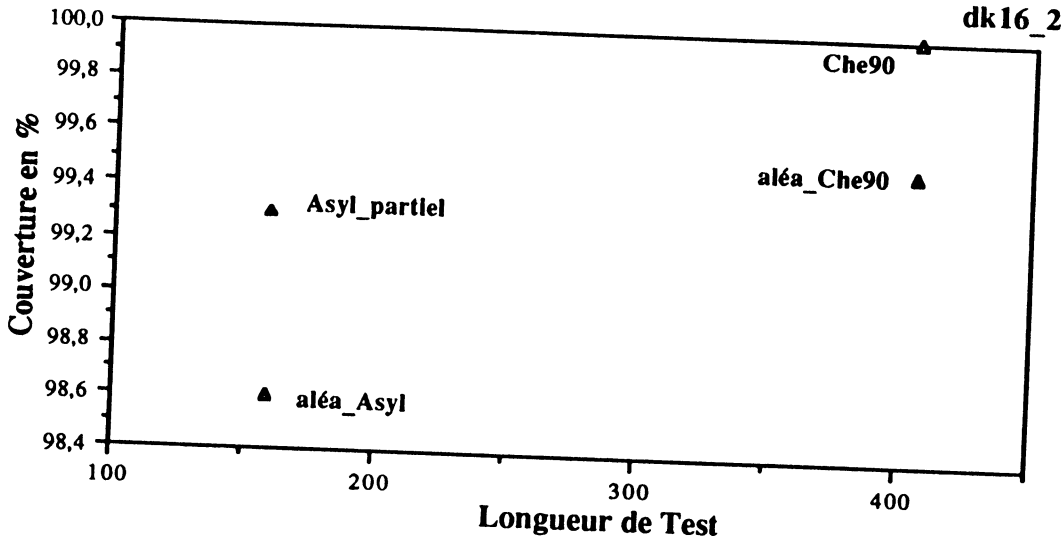
Résultats sur une deuxième implantation

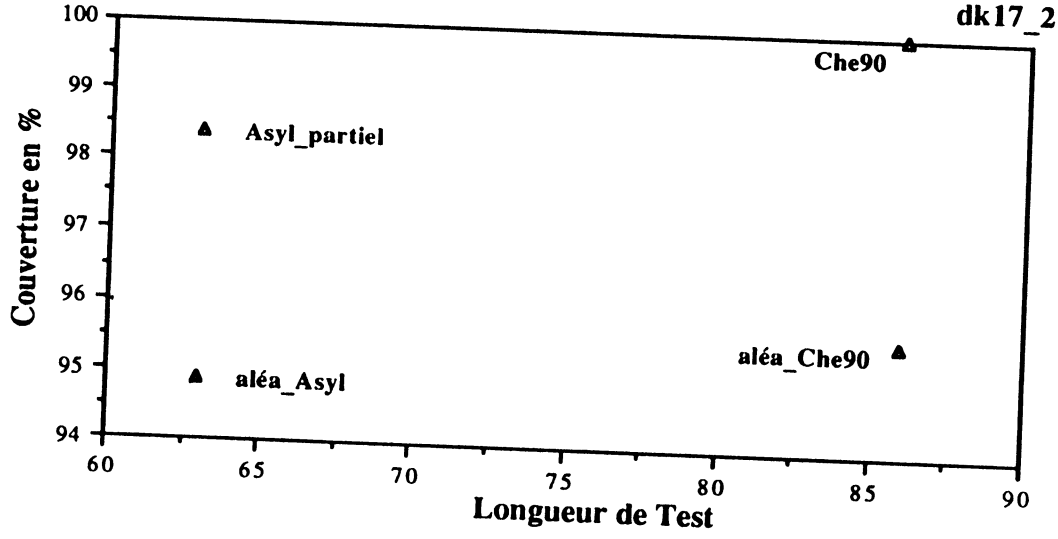


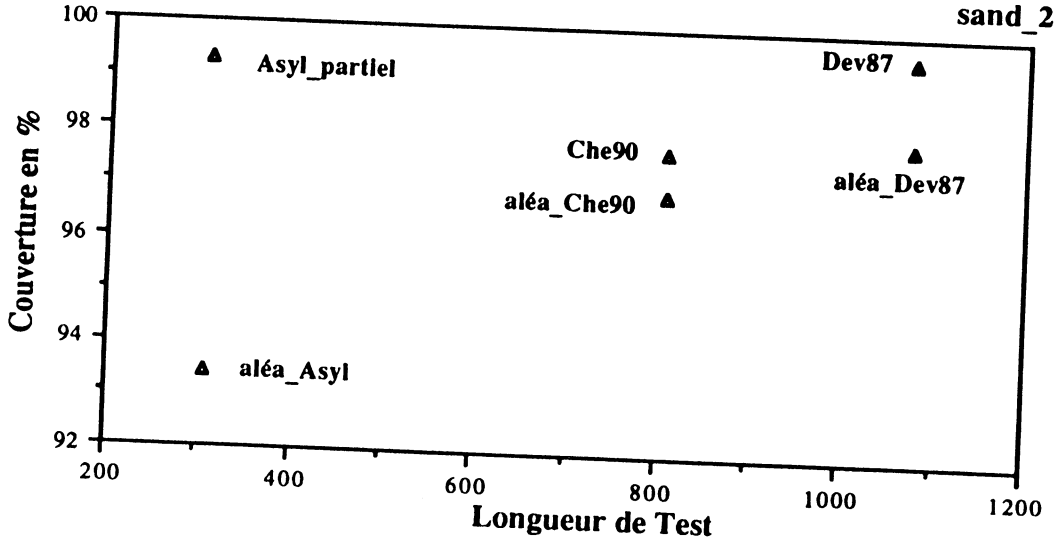

cse_2
Couverture en %
Longueur de Test
Asyl_partiel
Dev87
Che90
aléa_Che90
aléa_Dev87
aléa_Asyl
100
95
90
85
80
75
100
200
300
400
500
600
700
800
900
1000

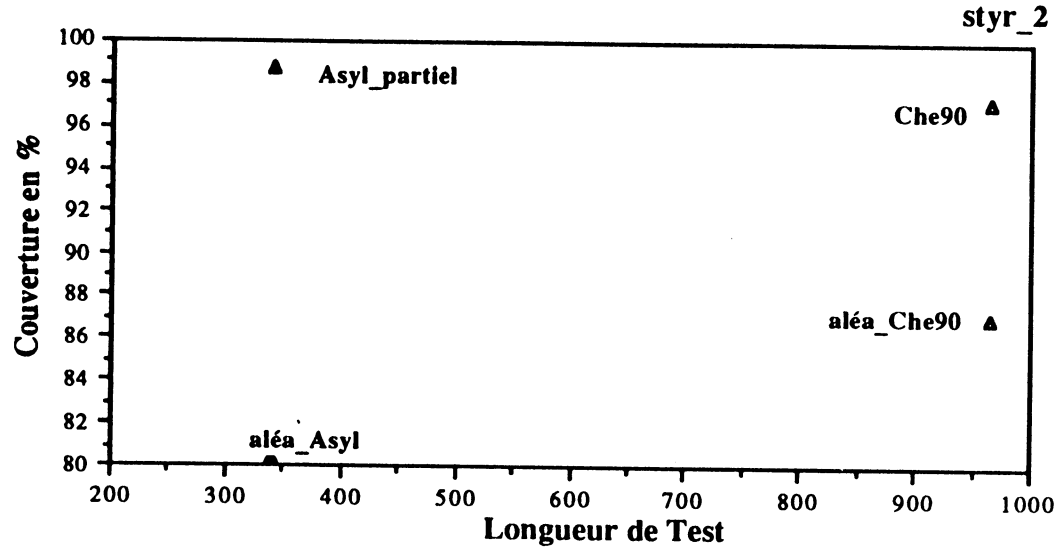

dk14_2
Couverture en %
Longueur de Test
Che90
aléa_Che90
Asyl_partiel
aléa_Asyl
100,0
99,5
99,0
98,5
98,0
97,5
97,0
96,5
96,0
50
75
100
125
150
175
200
225
250

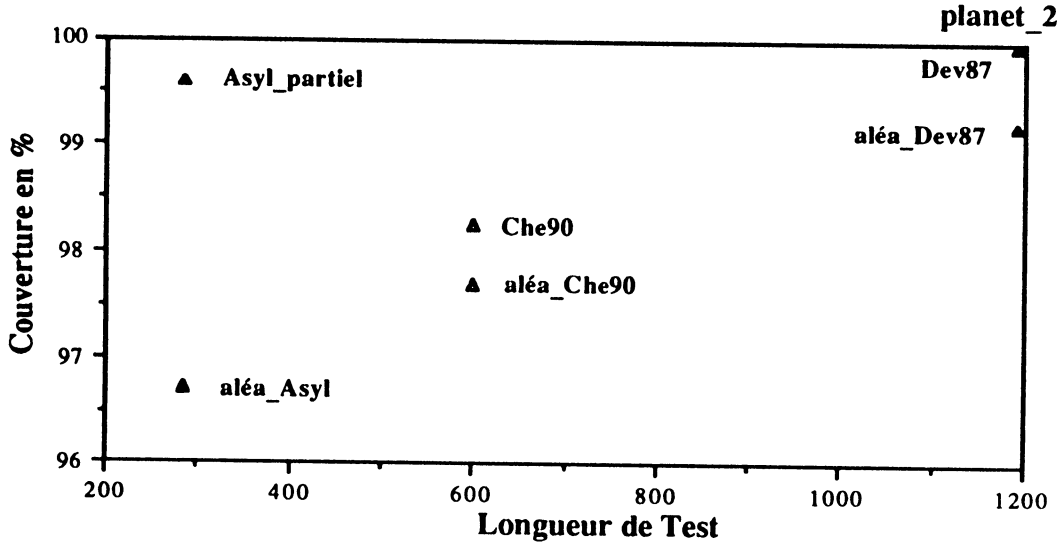

dk15_2
Couverture en %
Longueur de Test
Che90
Asyl_partiel
aléa_Che90
aléa_Asyl
100
99
98
97
96
20
40
60
80
100
120
140
160


dk16_2
Che90
aléa_Che90
Asyl_partiel
aléa_Asyl
Couverture en %
Longueur de Test
100,0
99,8
99,6
99,4
99,2
99,0
98,8
98,6
98,4
100
200
300
400


dk17_2
Che90
Asyl_partiel
aléa_Che90
aléa_Asyl
Couverture en %
Longueur de Test
100
99
98
97
96
95
94
60
65
70
75
80
85
90


sand_2
Asyl_partiel
Dev87
Che90
aléa_Dev87
aléa_Che90
aléa_Asyl
Couverture en %
Longueur de Test
100
98
96
94
92
200
400
600
800
1000
1200


styr_2
Asyl_partiel
Che90
aléa_Che90
aléa_Asyl
Couverture en %
Longueur de Test
100
98
96
94
92
90
88
86
84
82
80
200
300
400
500
600
700
800
900
1000


planet_2
Asyl_partiel
Dev87
aléa_Dev87
Che90
aléa_Che90
aléa_Asyl
Couverture en %
Longueur de Test
100
99
98
97
96
200
400
600
800
1000
1200

Chapitre 3: Génération hiérarchisée de test

I Etat de l'art et approche générale

La génération hiérarchisée de vecteurs de test a comme objectif de pallier l'explosion combinatoire des méthodes de génération de test du type chemin sensible sur les réseaux de portes logiques. Elle consiste à décrire un circuit comme une interconnexion de blocs plus complexes que des portes logiques, à associer à chaque bloc des vecteurs ou des séquences de test puis à les "justifier" pour amener leur valeurs depuis les plots d'entrées ainsi que vers les sorties pour manifester leurs effets sur les sorties primaires (figure 1).

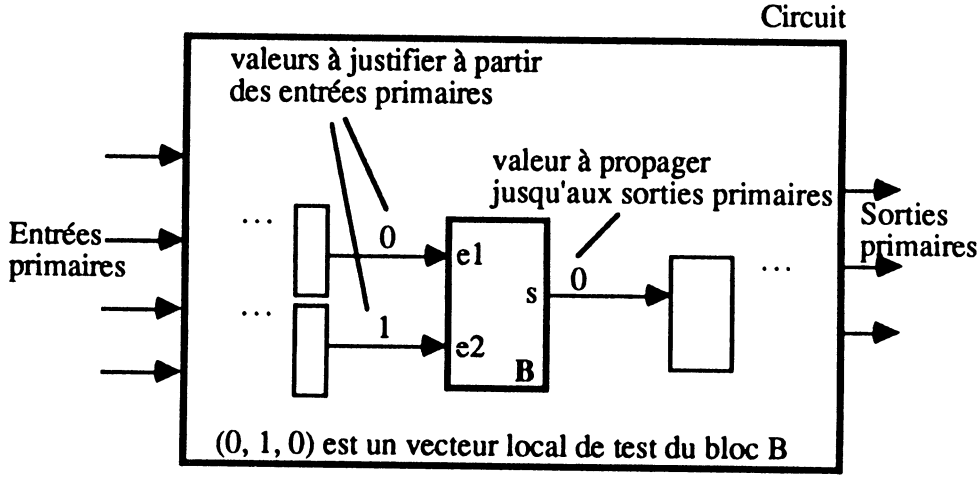


Figure 1. Principe de la génération hiérarchisée de test

Dans une réelle approche hiérarchisée la justification et la propagation avant se font également bloc par bloc. Cette même approche peut à nouveau être utilisée au niveau du bloc en introduisant ainsi plusieurs niveaux de la hiérarchie (figure 2).

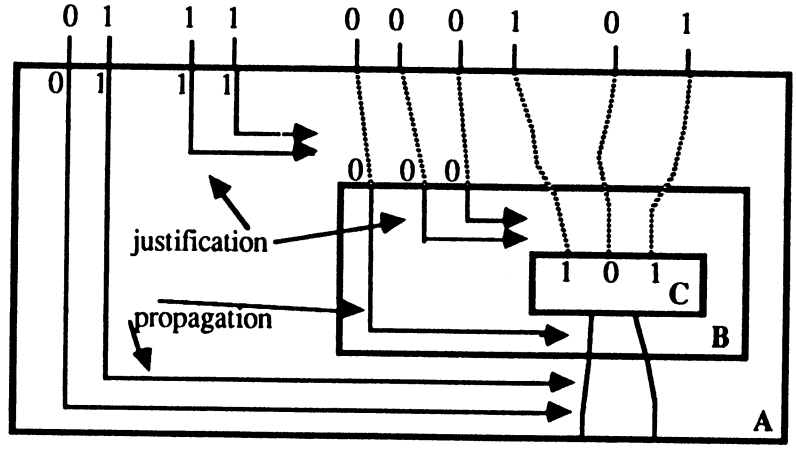


Figure 2. Exploitation de la hiérarchie à plusieurs niveaux

Les blocs définissant le circuit sont généralement typés et on distingue couramment les blocs combinatoires booléens, les blocs de logique séquentielle aléatoire, les automates d'états finis, les opérateurs arithmétiques incluant les UAL caractérisés par leurs structures répétitives (ILA) dûes aux "tranches de bits", les blocs mémoires (registre ou bloc ROM, RAM), les bus, ...
Le circuit global peut être quelconque ou réduit à des cas particuliers. Un des cas simplifiés couramment considérés est le cas des chemins de données caractérisés par le fait que les signaux de contrôle de tous les blocs sont déclarés contrôlables. Les interconnexions sont déclarées par connexions simples ou bus. Le dernier cas convient mieux aux descriptions architecturales.
Les méthodes de génération de test au niveau des blocs sont des méthodes classiques (D-algorithme, Podem ou différence booléenne) pour les circuits combinatoires, test spécifique aux structures répétitives (ILA) pour les blocs de traitement arithmétique [Jay89], un test spécifique aux automates d'états finis tel qu'il est décrit au chapitre précédent, un test fonctionnel classique pour les blocs mémoires, etc... Ces méthodes locales n'influent pas sur la génération hiérarchisée de test si ce n'est par le formatage des données de test, locales à chaque bloc. Ceci peut en effet être donné "à plat" en valeur binaire soit regroupé par valeurs symboliques. Une valeur symbolique représente un ensemble de données de test "équivalentes" traitées identiquement au niveau global du circuit. Ainsi 100 additions à effectuer pour un bloc additionneur à deux entrées seront représentées par un triplet symbolique de données (X, Y, Z) où X et Y représentent les entrées et Z la sortie de l'additionneur. En sortie, les valeurs justes et fausses en symbolique posent des problèmes pour traiter le problème de masquage [Bel80].
Pour la traversée des blocs en justification et propagation avant, les blocs traversés sont représentés au niveau fonctionnel. L'intérêt sera porté sur les traversés dites "transparentes" c. à. d. ne modifiant pas ou "peu" les données. Il s'agira du chemin identité appelé I-path défini dans [Aba85] ou de fonctions transparentes définies dans [Jay89]. Une approche réellement hiérarchisée évite de descendre au niveau des portes logiques et de traiter ces blocs comme de la logique séquentielle aléatoire. Pour ce qui concerne les modèles fonctionnels, la difficulté en propagation arrière pour la justification réside en une existance problématique de fonctions inverses dites aussi S-path dans [Fre88]. Si les fonctions inverses d'un bloc n'existent pas, le bloc sera déclaré opaque en traversée arrière. Pour la propagation avant, les modèles fonctionnels sont attrayants car ils permettent à la propagation de se rapprocher des techniques de simulation. Mais à ce niveau d'abstraction, toute fonction non injective pourra masquer la faute en délivrant la même sortie pour une

entrée juste et fausse. On sera donc amené à se restreindre aux fonctions injectives dites aussi F-path dans [Fre88]. Si ceci n'est pas respecté, le modèle fonctionnel ne peut donc apporter qu'une première définition de chemin de propagation dite symbolique. Celui-ci devra être validé par simulation pour chaque faute.
Par ailleurs, les heuristiques de justification peuvent être effectuées soit en partant des données locales de test jusqu'à arriver aux entrées primaires; on parlera de la technique ascendante, soit en cherchant les entrées primaires capables d'amener les valeurs souhaitables aux données locales ; on parlera de la technique descendante.
Le tableau ci-dessous propose un classement des principales approches en recherche, nombreuses ces deux dernières décades. Les thèmes traités dans chaque approche sont les suivants:

Le type des blocs traités. Indique pour chaque approche les blocs qu'elle peut traiter. Ces blocs peuvent être combinatoires booléens ou arithmétiques, des registres, bus et compteurs. Un bloc appelé "join" représente le regroupement de plusieurs bus, un bloc appelé "split" représente la division d'un bus en plusieurs et un bloc appelé "swizzle" représente un changement dans l'ordre des fils entre la sortie d'un bloc et l'entrée d'un autre.

La méthode de la génération de test au niveau bloc. Lorsqu'elle est spécifiée, la méthode de génération de test au niveau bloc est fondée sur une approche

classique qui se sert d'un algorithme tel le D-algorithme ou Podem,
fonctionnelle indépendante de la structure du bloc ou
de différence booléenne qui considère la fonction non minimisée d'un bloc.

Le type des vecteurs locaux de test. Indique si l'approche utilise des vecteurs symboliques ou réels

Le type d'interconnexion. Indique si les interconnexions sont de simples fils ou organisées en bus.

La restriction sur le circuit global. Indique si la partie traitée est du type chemin de donnée considérée seule ou en présence d'un automate d'états finis.

Les techniques de propagations avant et arrière. Indique le type de fonctions pris en compte au cours des prpagations. Il s'agit des fonctions transparentes et / ou directes injectives pour la propagation avant et inverses calculables pour la propagation arrière.

L'heuristique de justification au niveau circuit. Précise si la justification est effectuée par une technique ascendande ou descendante.

L'heuristique de testabilité. Indique si l'approche a recours à une heuristique de mesure de testabilité pour guider les propagations au niveau circuit.

Niveaux hiérarchiques. Précise si l'approche est appliquée sur deux ou plusieurs niveaux hiérarchiques.

Langages utilisés. Il s'agit de langages classiques du type C ou Pascal ou de langages orientés objet du type Lisp ou smalltalk.

Dans ce tableau les croix indiquent une correspondance entre une référence dans la littérature et un thème de la classification.

		Bha 85	Kri 87	Cha 87	Mur 88	Alf 88	Ani 89	Jay 89	Sar 89	Cal 89	Roy 90	Su 89 90	Kun 90	Lee 98 90	Lee 91	Krü 91
Type de blocs	booléen	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	op. arithmétique	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	registre	x			x	x	x	x							x	x
	bus	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	compteur															
	split, join	x			x			x					x			x
	swizzle							x								
méthode de génération au niveau bloc	non spécifiée	x		x	x	x	x				x		x	x	x	x
	D-algo		x													
	Podem											x				
	approche fonct.							x								
	autre								*	**						
vecteurs locaux de test	symboliques	x	x		x	x	x				x				x	
	réels			x				x	x	x		x	x	x		x
interconnexion	simple		x	x					x	x						
	bus	x			x	x	x	x			x	x	x	x	x	x
Restriction su circuit global	ch. de données	x		x	x	x		x	x	x	x	x	x			x
	ch. de données + auto. d'états finis		x				x							x	x	
techniques de propagations avant et arrière	fonc. transparente (I-path)		x	x	x	x	x	x	x	x	x	x		x		x
	fonc. directe injective (F-path)		x				x			x		x	x	x		
	fonc. inverse calculable (S-path)		x				x			x		x	x			
heuristique de justification niveau circuit	ascendante	x	x		x	x	x	x			x		x			x
	descendante			x					x	x		x		x	x	
heuristique de testabilité				Ben 84		Alf 87			Brg 84	Brg 84						***
plusieurs niveaux hiérarch	2niveaux	x		x					x		x	x	x		x	x
	plusieurs niveaux		x							x						
Langages utilisés	Lisp										x	x				
	smalltalk		x													
	C++												x			
	C ou Pascal	P	C		C	P	C		C					C	C	P

* théorie des dominateurs [Sch87]

** différence booléenne [Lar89]

*** l'heuristique d'analyse de testabilité dépend de la complexité des fonctions des blocs.

Approche générale

L'approche proposée ici consiste à explorer comme précédemment l'usage du modèle fonctionnel [Kar91b], [Kar91c]. Pour en assurer une certaine efficacité, elle sera restreinte aux chemins de données. Les signaux de contrôle sont déclarés contrôlables.

Le circuit est décrit en termes de blocs fonctionnels interconnectés. Les vecteurs de test locaux à chacun des blocs sont associés aux fonctions des blocs. Ce sont des vecteurs symboliques. A partir des vecteurs symboliques locaux et de la description du circuit, les vecteurs symboliques globaux au niveau circuit sont générés par des procédures de propagation arrière (méthode ascendante) pour la justification et de propagation avant [Cra88a], [Cra88b].

En propagation arrière, les blocs à fonctions inverses non définies sont déclarés opaques et conduisent à définir des points de test améliorant la testabilité du circuit. En propagation avant et arrière, on définit des arbres de propagation symbolique. Le programme de test final est à valider si des fonctions non injectives sont utilisées dans la propagation avant. En fait pour les circuits considérés (chemins de données) les fonctions utilisées sont des fonctions "quasi-transparentes" ce qui assure précisément leur injectivité. Signalons que les problèmes temporels sont simplifiés par suite d'une restriction aux circuits synchrones à barrières temporelles bien identifiées.

La génération des vecteurs réels globaux de test nécessite la connaissance des vecteurs réels locaux aux blocs. Les vecteurs réels sont déduits en remplaçant les symboles dans les vecteurs symboliques globaux par leur valeurs réelles.

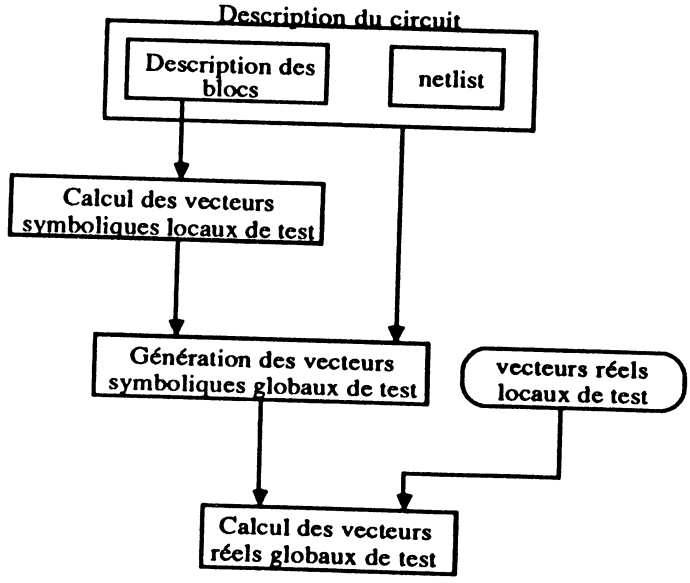


Figure 3. Approche générale proposée

Un des intérêts de la méthode proposée est de pouvoir étudier la stratégie de test de façon indépendante des vecteurs réels locaux de test. Elle est ainsi indépendande de la conception finale de chaque bloc. Cette stratégie garantit, après modification, une couverture de 100% au niveau circuit si la couverture au niveau bloc est de 100%. Plusieurs essais de modification de la conception du bloc peuvent être effectués pour améliorer la couverture du bloc sans que cela modifie les résultats donnés par la stratégie de test au niveau circuit [Cra89b], [Cra89c], [Cra90].
Par ailleurs la méthode proposée a l'avantage d'être plus rapide que les méthodes travaillant directement sur les vecteurs réels locaux de test, d'être capable de traiter les conflits lorsque ceci est possible et de proposer un ensemble minimal de points de test afin d'améliorer la testabilité du circuit au lieu d'une insertion systématique de cellules de "scan path".
Cependant elle a l'inconvénient d'être restreinte aux blocs ayant des fonctions transparentes ou ayant un mode quasi-transparent. Elle ne peut pas traiter tous les types de blocs séquentiels et exige un scan path entre les parties opérative et contrôle dans le cas d'un circuit composé d'une partie opérative et une partie de contrôle.

La suite de ce chapitre traite chacune des étapes énoncées sur la figure 3. Le § II est consacré à la description du circuit. Dans le § III on définit les vecteurs symboliques de test locaux aux blocs du circuit. La génération des vecteurs symboliques globaux de test est présentée dans les paragraphes IV, V, VI, VII et VIII. Le § IX calcule les vecteurs réels globaux de test. Le § X montre l'efficacité de la méthode en s'appuyant sur des résultats expérimentaux.

II Modélisation d'un circuit

Dans cette étude un circuit est vu comme une interconnexion de blocs. Les blocs sont considérés au niveau architectural (RTL). A titre d'exemple si nous considérons un chemin de donnée d'un circuit intégré, les blocs peuvent être des registre, additionneur, soustracteur, unité arithmétique et logique (UAL), porte logique, bus, etc...
La modélisation d'un circuit proposée ici consiste à modéliser les blocs et à déclarer leurs interconnexions (netlist). Pour faciliter la description les blocs sont typés. Un multiplexeur à deux entrées par exemple est un type de blocs qui peut être instancié plusieurs fois dans le circuit. Notre approche utilise une modélisation fonctionnelle des bloc-types. Elle ne prend pas en compte leur structure (niveau portes logiques ou niveau inférieur). Les fonctions des blocs peuvent être fournies par le concepteur, extraites d'une bibliothèque dans le cas où les blocs appartiennent à une bibliothèque donnée ou déduites d'une description de haut niveau de type VHDL par exemple.

II. 1 Déclaration des interconnexions

Une connexion entre deux blocs d'un circuit peut représenter un seul fil ou une nappe de fils. Les connexions entre des blocs logiques ou des connexions relatives aux signaux de contrôle sont souvent constituées par un seul fil. Le second type de connexion est adapté aux chemins de donnée où une connexion représente un ensemble de fils dont le nombre est défini par la largeur des bus. Lorsque l'ordre des fils aux extrémités initiale et finale est conservé la connexion est dite *connexion régulière*. Sur la figure 4a les connexions entre les blocs B1 et B2 sont régulières. Elles peuvent être remplacées par une seule connexion (figure 4b).

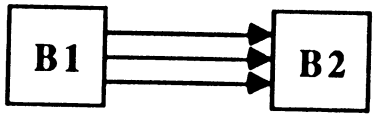


Figure 4a. Connexions régulières sur 3 bits

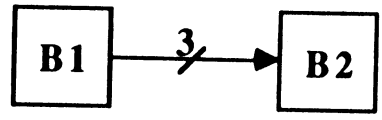


Figure 4b. Une connexion

Toutefois il existe des chemins de donnée où les connexions entre deux blocs sont dites *connexions irrégulières*. Ceci se produit dans l'un ou les deux cas suivants:

1. le bus d'entrée d'un bloc est formé par un sous ensemble de l'ensemble des fils du bus de sortie du bloc père ou par la concaténation de deux ou de plusieurs bus de sortie de plusieurs pères.

2. les fils d'une connexion ne conservent pas à l'extrémité finale l'ordre qu'ils ont à l'extrémité initiale. La figure 5a illustre ce cas.
Les connexions irrégulières entre deux blocs peuvent être représentées par un bloc fictif dont la fonction effectue la division d'un bus, la concaténation de plusieurs bus et/ ou la permutation de l'ordre des fils à la sortie par rapport à leur ordre à l'entrée (figure 5b). Dans la littérature technique de tels blocs sont appelés "split", "join" et "swizzle" respectivement.

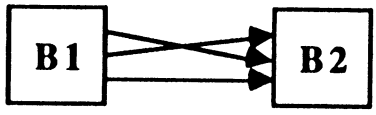


Figure 5a. Exemple de connexions irrégulières

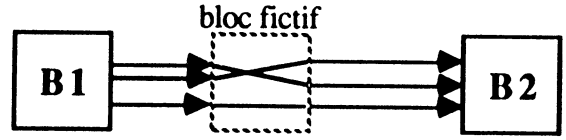


Figure 5b. Modèle de connexions irrégulières

Les connexions entre les blocs d'un circuit sont définies par leurs extrémités initiale et finale. L'extrémité initiale d'une connexion est définie en déclarant le bloc initial (père) et la sortie correspondante. L'extrémité finale est définie en déclarant le bloc final (fils) et son entrée correspondante (figure 6).

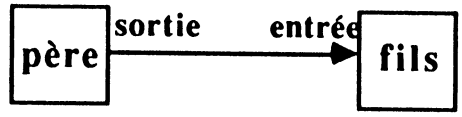


Figure 6. Blocs père et fils d'une connexion

Dans cette étude les connexions sont déclarées par un ensemble de litéraux écrits en Prolog. Les connexions ayant le même bloc père et le même bloc fils sont déclarées par un seul littéral . Le format d'un littéral de connexion est le suivant:
connec (bloc_fils, [entrée], bloc_père, [sortie]).

Exemple

Soit B1, B2 et B3 trois blocs d'un circuit interconnectés comme le montre la figure 7.

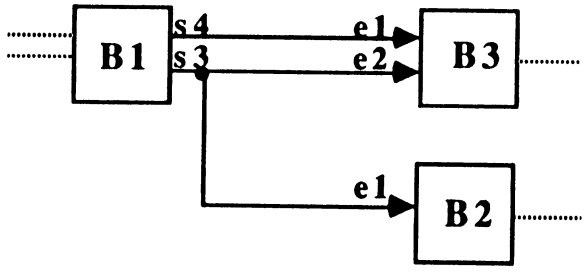


Figure 7. Exemple de blocs interconnectés

Les litéraux correspondant aux trois connexions de cet exemple sont les suivants:
connec (B2, [e1], B1, [s3]).
connec (B3, [e1, e2], B1, [s4, s3]).

II. 2 Description fonctionnelle d'un bloc

La description fonctionnelle d'un bloc consiste à décrire ses *fonctions de base*. Les fonctions de base d'un bloc sont les fonctions élémentaires qu'un bloc peut effectuer. La mémoire de la figure 8a, par exemple, comporte deux fonctions de base qui sont l'écriture et la lecture d'un mot mémoire. L'additionneur de la figure 8b comporte une fonction de base qui est l'addition de ses deux entrées e1 et e2.

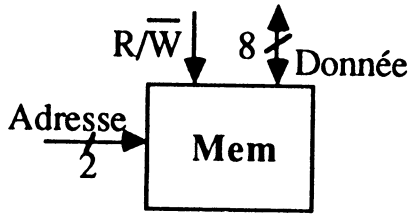


Figure 8a. Exemple d'une mémoire statique

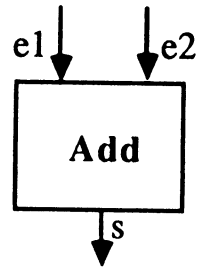


Figure 8b. Exemple d'un additionneur

II. 2. 1 Identification des fonctions de base d'un bloc

Les fonctions de base d'un bloc combinatoire sont définies comme des fonctions booléennes ou arithmétiques des sorties par rapport au entrées.
Les blocs séquentiels traités sont réduits aux éléments de mémorisation tels les registres et les mémoires. Les contrôleurs ou bloc séquentiel aléatoire sont déclarés des blocs "opaques" (§ II.3.2). Lorsqu'il s'agit d'une fonction de base du type mémoire les points internes de mémorisation peuvent être des entrées et/ ou des sorties de la fonction. Dans un bloc séquentiel on identifie des fonctions de base qui permettent de charger et/ ou lire un élément de mémorisation. La fonction d'écriture change la valeur de l'élément de mémorisation (figure 9a) alors que la fonction de lecture manifeste cette valeur sur les sorties du bloc (figure 9b). L'élément de mémorisation se comporte comme une variable de sortie pour la fonction d'écriture et d'entrée pour la fonction de lecture. Dans le cas où une fonction de base peut être appliquée sur plusieurs éléments de mémorisation d'un bloc, l'adresse identifiant un élément parmi les éléments du bloc, est un paramètre de la fonction.

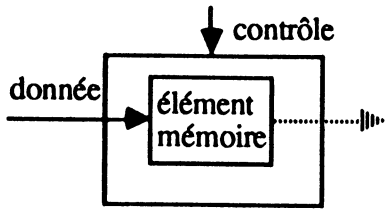


Figure 9a. Elément de mémorisation à écriture

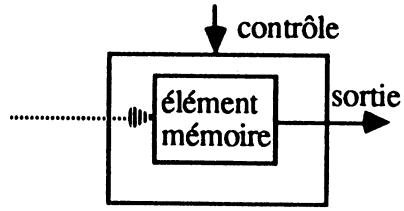


Figure 9b. Elément de mémorisation à lecture

Un bloc à éléments de mémorisation peut être schématisé de différentes manières suivant que les éléments sont à écriture et lecture ou écriture ou lecture. Dans le premier cas les fonctions d'écriture ou de lecture peuvent être appliquées séparément ou confondues.

Cas 1: Eléments de mémorisation à écriture et lecture séparées

Le bloc est remplacé par deux sous-blocs fictifs (fig 10a). Le premier sous-bloc effectue la fonction d'écriture et le deuxième la fonction de lecture d'une variable de mémorisation. L'entrée du premier sous-bloc et la sortie du second sont les entrée et sortie du bloc origine.

Une RAM est un exemple typique où les éléments de mémorisation sont à écriture et lecture séparées.

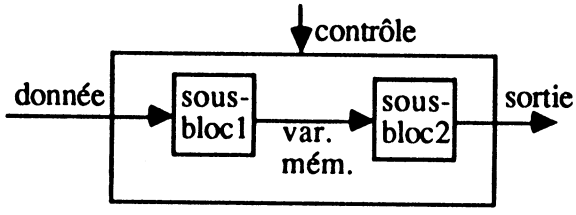


Figure 10a. Bloc à fonctions d'écriture et lecture séparées

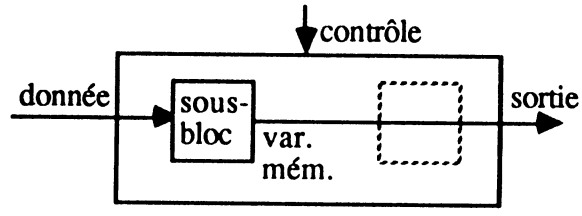


Figure 10b. Bloc à fonction composée d'écriture et lecture

Cas 2: Eléments de mémorisation à fonction composée d'écriture et de lecture

C'est un cas particulier du cas précédent où le deuxième sous-bloc disparait et la variable de mémorisation est confondue avec la variable de sortie du bloc (figure 10b). C'est le cas typique d'un registre pour lequel une valeur chargée dans l'élément de mémorisation se manifeste également sur la sortie du bloc.

Cas 3: Eléments de mémorisation à lecture ou écriture uniquement

Dans le cas d'éléments de mémorisation à lecture (écriture) uniquement, le bloc comprend un sous-bloc pouvant effectuer la fonction de lecture (écriture) et ayant la variable de mémorisation pour entrée (sortie). L'entrée (sortie) du sous-bloc est

déconnectée de l'entrée (sortie) du bloc origine. Elle est non contrôlable (observable). Ceci est illustré sur la figure 11b (figure 11a). Les éléments de mémorisation d'une ROM par exemple sont des éléments à lecture seulement.

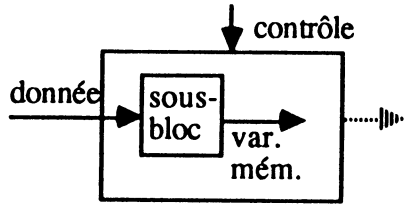


Figure 11a. Modèle d'un bloc à éléments mémorisation en écriture

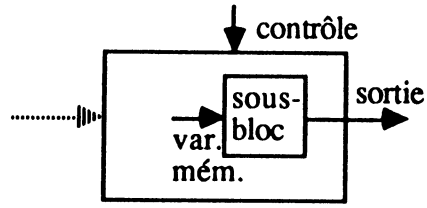


Figure 11b. Modèle d'un bloc à éléments de mémorisation en lecture

II. 2. 2 Traitement du temps

Dans la description des fonctions de base d'un bloc le temps est discrétisé en instants définis par une horloge de base. Dans le cas où les blocs du circuit sont synchronisés sur un front d'une seule horloge cette horloge définit l'horloge de base. Elle est modélisée implicitement. Dans le cas où les blocs du circuit sont synchronisés sur différents fronts de plusieurs horloges, l'horloge de base est alors une horloge fictive produit des différentes horloges du circuit. Les différentes horloges sont considérées comme des entrées primaires aux circuit et apparaissent explicitement dans les modèles des fonctions de base.

Les blocs séquentiels du circuit sont supposés évoluer au rythme de l'horloge de base. Le chargement d'une valeur dans un point de mémorisation est commandé par le signal d'horloge lui-même (figure 12a) ou bien par un signal de chargement (figure 12b) qui reproduit l'horloge de base dans le cas de l'activation du chargement (figure 12c).

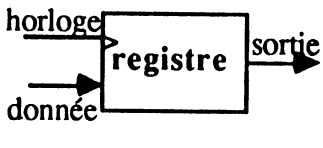


Figure 12a. registre commandé par l'horloge

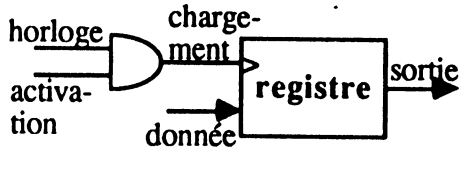


Figure 12b. registre à signal d'activation

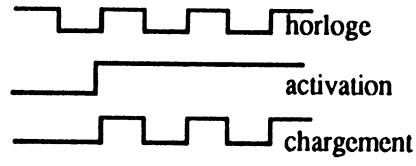


Figure 12c. reproduction de l'horloge sur le signal d'activation

Si le chargement des points mémoire se fait aux fronts montants (descendants) de l'horloge, les instants discrets sont les fronts montants (descendants) de l'horloge. Ceci est illustré sur la figure 13.

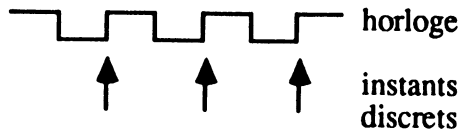


Figure 13. Instants discrets relatifs aux fronts montants de l'horloge

Un instant discret est représenté par une valeur symbolique. La valeur symbolique t d'un instant discret correspond au $t^{ème}$ front montant (descendant) de l'horloge compté à partir d'un front fictif initial. Dans la suite la "valeur symbolique d'un instant" sera remplacée par "instant symbolique" tout court. Si t est un instant symbolique, t+1 et t-1 représentent respectivement les instants suivant et précédent à t. Dans ce cas t est pris comme repère relatif. Ceci est illustré sur la figure 14.

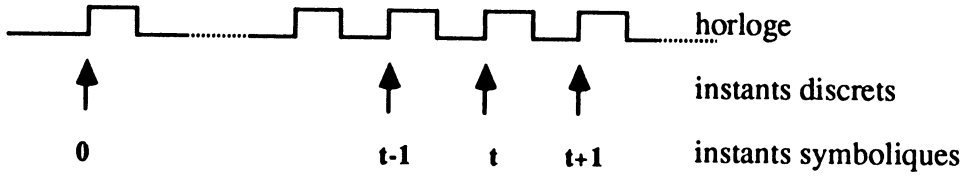


Figure 14. Instants symboliques relatifs aux fronts montants de l'horloge

Dans la description d'une fonction de base, les instants symboliques sont utilisés pour répondre aux questions suivantes:
à quel(s) instant(s) faut-il appliquer les entrées (donnée et contrôle) si on veut avoir une donnée valide à un instant t?
Si l'entrée est valide à l'instant t, quand faut-il s'attendre à une sortie valide?
Une valeur est valide à un instant symbolique quelconque veut dire qu'elle a été appliquée avant cet instant c.à.d avant le front de l'horloge correspondant à cet instant. Dans notre approche on suppose que la période de l'horloge est suffisante pour qu'une fonction combinatoire aient ses entrées et les sorties correspondantes valides au même instant. Ceci revient à ne pas tenir compte du retard dû à la traversée d'une fonction combinatoire. Dans l'exemple de la fonction combinatoire de la figure 15a, les entrées et sortie sont valides à l'instant t (figure 15b).

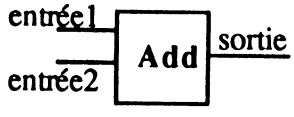


Figure 15a. Exemple d'un bloc à fonction combinatoire

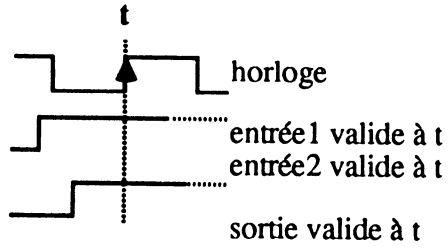


Figure 15b. Entrées et sorties valides au même instant symbolique

Dans le cas d'une fonction séquentielle si l'entrée est valide en un instant, la sortie ne peut pas l'être au même instant car elle ne change qu'après le front de l'horloge correspondant à cet instant. Elle sera valide à l'instant suivant. Dans l'exemple de la figure 16a, la sortie du registre est valide à l'instant symbolique t+1 alors que l'entrée "donnée" est valide à l'instant t (figure 16b).

Figure 16a. Exemple de fonction séquentielle

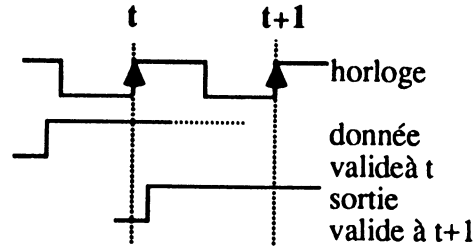


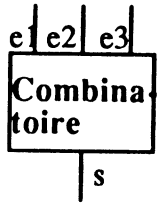
Figure 16b. Donnée et sorties valides à des instants symboliques consécutifs

II. 3 Description d'une fonction de base

Une fonction de base est décrite par ses fonctions directe et inverse. Il existe différentes façons de décrire une fonction directe ou inverse. Elle peut être tabulée (figure 17a), arithmétique (figure 17b), booléenne ou donnée sous une forme algorithmique (figure 17c).

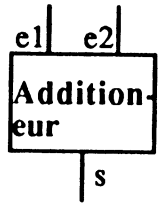
Une fonction est dite *opaque* lorsqu'elle ne peut pas être décrite.

Les signaux d'entrée et de sortie d'une fonction peuvent être des signaux primaires au bloc ou internes lorsqu'il s'agit de variables de mémorisation (§ II.2.1). Dans la suite on parlera de signaux d'entrée et de sortie.



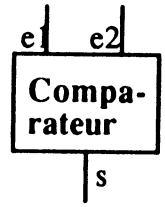
e1 e2 e3 = 110 => s=1
e1 e2 e3 ≠ 110 => s=0

Figure 17a. Fonction directe tabulée



s = e1 + e2

Figure 17b. Fonction directe arithmétique



e1 < e2 => s=1
sinon s=0

Figure 17c. Fonction directe algorithmique

II. 3. 1 Modélisation des fonctions directe et inverse en Prolog

Une fonction directe ou inverse est décrite par des clauses temporisées qui déclarent ses paramètres et définissent les conditions de son application et ses effets.

Le format général d'une clause de description d'une fonction est le suivant:

description_type(type_bloc, nom_fonction, paramètres, instant1):-
mettre_valeur ([(conditions, instant2)]),
égale ([(effets, instant3)]).

Le terme "description_type" est remplacé par le terme "fonc_directe" ou "fonc_inverse" s'il s'agit de la description d'une fonction directe ou inverse.

Le terme "type_bloc" désigne le type du bloc.

Le terme "nom_fonction" donne le nom de la fonction à décrire.

Le terme "paramètres" inclut les entrées et sorties primaires et variables internes de mémorisation qui entrent en jeu pour exercer la fonction et observer ses effets. Ce terme comporte cinq listes qui sont la liste de signaux d'entrée primaire, de contrôle, de variables de mémorisation d'entrée, de signaux de sortie et des variables de mémorisation de sortie. A chacun des signaux de donnée et de contrôle d'entrée, sortie et variables de mémorisation est affectée une valeur symbolique.

"instant1" désigne l'instant symbolique auquel l'entrée (sortie) est considérée valide s'il s'agit de la modélisation d'une fonction directe (inverse).

Le littéral "mettre_valeur ([(conditions, instant2)])" spécifie la valeur des signaux de contrôle qu'il faut appliquer et l'instant symbolique de l'application afin d'exercer la fonction directe ou inverse. C'est le terme "conditions" qui comporte les signaux de contrôle et leur valeur respective. Le terme "instant2" spécifie l'instant symbolique où les signaux de contrôle doivent être valides. Cet instant est exprimé en fonction de l'instant "instant1".

Le littéral "égale ([(effets, instant3)])" spécifie la valeur que prennent les signaux de sortie et d'entrée dans le cas d'une fonction directe, ou simplement les signaux d'entrée s'il s'agit de la fonction inverse et précise l'instant de l'affectation. C'est le terme "effets" qui comporte les signaux à affecter et leur valeur d'affectation. "instant3" spécifie l'instant symbolique de l'affectation. Cet instant est exprimé en fonction de l'instant "instant1".

II. 3. 2 Caractéristiques des fonctions de base

Une fonction de base est caractérisée par quatre paramètres qui sont sa transparence, sa dépendance des entrées et sorties, la transformation des données et l'injectivité de sa fonction directe.

Type de transparence

On définit deux types de transparence d'une fonction de base non opaque.

Une fonction à une entrée de donnée est dite *transparente* si la sortie est égale à l'entrée.

Une fonction de base à N entrées Z=f(X1, ..., Xi-1, Xi, Xi+1, ..., XN) admet un mode *quasi-transparent* par rapport à une entrée Xi si en forçant une valeur aux entrées X1, ..., Xi-1, Xi+1, ..., XN, la fonction est égale à l'entrée Xi. L'entrée Xi ne subit ainsi aucune transformation. La fonction de base d'un additionneur par exemple admet un mode quasi-transparent par rapport à chacune de ses entrées. Il suffit de fixer une entrée à la valeur 0 pour que la sortie soit égale à la deuxième entrée.

De façon générale, toutes les opérations booléenne et arithmétique comprenant des éléments neutres admettent un mode quasi-transparent.

Lorsque le mode quasi-transparent existe, les fonctions directe et inverse sont modélisées dans ce mode.

Type de la transformation

Ce facteur traduit le type de la transformation qui existe entre les entrées et la sortie d'une fonction de base. Cette transformation peut être séquentielle. Dans le cas contraire on précise si elle est booléenne ou arithmétique.

S-activation: la transformation entre entrée et sortie est une fonction séquentielle.

B-activation: la transformation entre entrée et sortie est une fonction booléenne.

A-activation: la transformation entre entrée et sortie est une fonction arithmétique.

Degré de dépendance

Une fonction Z=f(X) où X est une variable booléenne générale codant une entrée est une fonction unaire dite à *dépendance simple* (SD). Une fonction N-aire Z=f(X1, X2, ..., Xn) est dite à *dépendance multiple* (MD).

Le *degré de dépendance* est égal au nombre N des entrées de la fonction. La fonction de base d'un inverseur par exemple est du type SD alors que pour un additionneur à deux entrées, la fonction de base est MD-type avec N égal à 2.

Les entrées (sortie) du bloc dont dépend une fonction directe (inverse) sont déclarées dans un littéral de dépendance qui permet d'assister les propagations avant et arrière. Cette dépendance est modélisée en langage prolog comme suit:

dépendance_directe (type_bloc, nom_fonction, [variables_entrées_primaires], [variable_entrées_mémorisation]).
dépendance_inverse (type_bloc, nom_fonction, [variables_sorties_primaires], [variable_sorties_mémorisation]).
Comme leur nom l'indique, les termes "type_bloc" et "nom_fonction" désignent le type du bloc et le nom de la fonction de base.

Injectivité

L'injectivité est définie pour la fonction directe.

II. 3. 3 Modélisation en Prolog des caractéristiques d'une fonction de base

Les caractéristiques d'une fonction de base sont regroupées dans un seul littéral ayant le format suivant.

caractéristiques(type_bloc, nom_fonction, [type_transparence_fonc_directe, injectivité], [type_transparence_fonc_inverse], [mode quasi_transparent], type de transformation, degré N de dépendance).

Comme leur nom l'indique, les termes "type_bloc" et "nom_fonction" désignent respectivement le type du bloc et le nom de la fonction de base.
Les termes "type_transparence_fonc_directe" et "type_transparence_fonc_inverse" donnent le type de transparence des fonctions directe et inverse respectivement. Ils seront remplacés par transparente, calculable ou opaque.
Le terme "mode quasi_transparent" donne le mode quasi-transparent par rapport aux entrées de la fonctions.

II. 4 Exemple1: Description d'une fonction de base arithmétique

Soit une UAL (figure 18) pouvant effectuer 4 fonctions de base (2 entrées de contrôle C1 et C2).

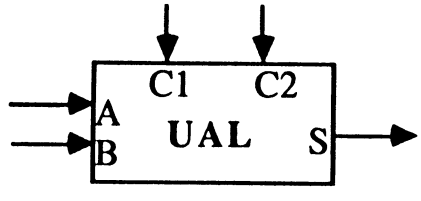


Figure 18. Exemple d'un UAL à 4 fonctions

Considérons la fonction de base "addition des entrées A et B". Elle admet une fonction directe mais pas de fonction inverse. De plus elle admet un mode quasi-transparent par rapport à l'entrée A et par rapport à l'entrée B.
Les paramètres de cette fonction sont les signaux de contrôle C1 et C2, de donnée d'entrée A et B et de sortie S. On remarque que ces fonctions n'ont pas de variables de mémorisation d'entrée ni de sortie. Un "-" indique cette absence. Aux signaux A, B, C1, C2 et S sont affectées respectivement les valeurs symboliques X, Y, V1, V2 et W.

II. 4. 1 Description des fonctions directes et inverses

```
fonc_directe (UAL, Add,([(A, X),(B, Y) ], [(C1, V1),(C2, V2)],-, [(S,W) ],-),t) :-
mettre_valeur ( [ ((V1, 0) , (V2, 0) , t) ] ),
égale ( [ ((W, X+Y), t) ] ).

/** description du mode quasi-transparent **/

fonc_directe(UAL, Add_A,([(A, X),(B, Y)], [(C1, V1),(C2, V2)],-, [(S,W) ],-),t) :-
mettre_valeur ( [ ((V1, 0) , (V2, 0) , t) ] ),
égale ( [ ((Y, 0), (W, X), t) ] ).

fonc_inverse(UAL, Add_A,([(A, X),(B, Y)], [(C1, V1),(C2, V2)],-, [(S,W) ],-),t) :-
mettre_valeur ( [ ((V1, 0) , (V2, 0) , t) ] ),
égale ( [ ((Y, 0), (X, W), t) ] ).

fonc_directe(UAL, Add_B,([(A, X),(B, Y)], [(C1, V1),(C2, V2)],-, [(S,W) ],-),t) :-
mettre_valeur ( [ ((V1, 0) , (V2, 0) , t) ] ),
égale ( [ ((X, 0), (W, Y), t) ] ).

fonc_inverse(UAL, Add_B,([(A, X),(B, Y)], [(C1, V1),(C2, V2)],-, [(S,W) ],-),t) :-
mettre_valeur ( [ ((V1, 0) , (V2, 0) , t) ] ),
égale ( [ ((X, 0), (Y, W), t) ] ).
```

La première clause décrit la fonction de base directe. Elle signifie que si les valeurs X et Y des entrées A et B sont valides à l'instant symbolique t (tête de la clause) et que les signaux de contrôle C1 et C2 ont la valeur 0 valide à l'instant t (premier littéral) alors la sortie S prend la valeur W=X + Y valide à l'instant t (deuxième littéral).

La deuxième (quatrième) clause décrit la fonction directe en mode quasi-transparent par rapport à l'entrée A (B). Elle signifie que pour que la valeur X (Y) de l'entrée A (B) qui est valide à l'instant symbolique t (tête de la clause) subisse la transformation identique il faut que les signaux de contrôle C1 et C2 aient la valeur 0 valide à l'instant t (premier littéral) et que la valeur Y (X) de l'entrée B (A) soit mise à 0. Ceci produira une sortie S de valeur W=X (W=Y) valide à l'instant t (deuxième littéral).
La troisième (cinquième) clause décrit la fonction inverse en mode quasi-transparent par rapport à l'entrée A (B). Elle signifie que si la valeur W de la sortie S est valide à l'instant symbolique t (tête de la clause) il suffit que les signaux de contrôle C1 et C2 aient la valeur 0 valide à l'instant t (premier littéral) et que la valeur Y (X) de l'entrée B (A) soit mise à 0 pour que l'entrée A (B) ait la valeur X=W (Y=W) valide à l'instant t (deuxième littéral).

II. 4. 2 Modélisation de dépendance

La modélisation en Prolog de la dépendance de la fonction de base "addition de A et B" est donnée par les litéraux suivants correspondant respectivement à la fonction directe de la fonction de base et aux fonctions directe et inverse en mode quasi-transparent par rapport à l'entrée A et B.

```
dépendance_directe(UAL, Add, [ A, B ], -).
dépendance_directe(UAL, Add_A, [ A,- ], -).
dépendance_inverse(UAL, Add_A, [ S ] , - ).
dépendance_directe(UAL, Add_B, [ -, B ], - ).
dépendance_inverse(UAL, Add_B, [ S ] , - ).
```

II. 4. 3 Modélisation des caractéristiques de la fonction de base

Les caractéristiques de la fonction de base "addition de A et B" sont regroupées dans le littéral suivant.

```
caractéristiques ( UAL, Add, [calculable, injective], [opaque], [Add_A, Add_B], A-activation, 4).
```

Comme le montre ce littéral , la fonction directe Add n'est pas transparente et la fonction inverse est opaque. Le mode quasi-transparent existe par rapport aux entrées A et à B. Cette fonction est arithmétique, multiple dépendance avec un degré de dépendance égal à 4.

II. 5 Exemple 2: Description d'une fonction de base de type mémoire

Soit la RAM de la figure 19. La description de la fonction de base "écrire un mot mémoire" est donnée ci-dessous. Les fonctions directe et inverse existent. Elles font intervenir une variable de mémorisation de sortie notée Var. Les paramètres de ces

fonctions sont la donnée d'entrée primaire D, l'adresse A, le signal de lecture et d'écriture R/Wb et la variable de mémorisation de sortie Var. Les valeurs symboliques qui sont affectées à ces paramètres sont respectivement X, V2, V1 et M.

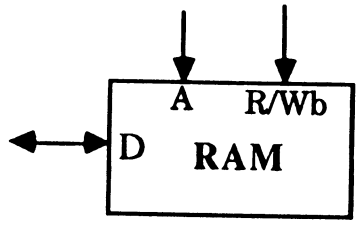


Figure 19. Une mémoire RAM

II. 5. 1 Description des fonctions directes et inverses

fonc_directe (RAM, écriture_D,([(D, X)], [(A, V2), (R/Wb, V1)], -, -,[(Var, M)]), t) :-
mettre_valeur ([((V1, 0) , (V2, V2), t)]),
égale ([((M, X), t+1)]).

fonc_inverse(RAM, écriture_D,([(D, X)], [(A, V2), (R/Wb, V1)], -, -,[(Var, M)]), t) :-
mettre_valeur ([((V1, 0) , (V2, V2), t-1)]),
égale ([((X, M), t-1)]).

La première clause signifie que si la valeur X de l'entrée D est valide à l'instant symbolique t, le signal d'écriture R/Wb a la valeur V1 égale à 0 valide à l'instant t et que la valeur V2 de l'adresse est valide à l'instant t alors la variable de mémorisation de sortie S prend la valeur M=X valide à l'instant t+1 (deuxième littéral).
La deuxième clause relative à la fonction inverse signifie que si la valeur M de la variable de mémorisation de sortie Var est valide à l'instant symbolique t, il suffit que le signal d'écriture R/Wb ait la valeur V1 égale à 0 valide à l'instant t-1 (premier littéral), et l'adresse A la valeur V2 valide à t-1 pour que l'entrée D prenne la valeur X=M valide à l'instant t-1.

II. 5. 2 Modélisation de dépendance

Les littéraux de dépendance de la fonction de base "écrire" sont les suivants.
dépendance_directe(RAM, écrire, ([D], -).
dépendance_inverse(RAM, écrire, (-, [Var]).

II. 5. 3 Modélisation des caractéristiques de la fonction de base

Les caractéristiques de la fonction de base "écrire un mot mémoire" sont données dans le littéral suivant.

caractéristiques (RAM, écrire, [transparent, injective], [transparent],- , S-activation, 2).

La fonction de base est transparente, séquentielle, multiple dépendance avec un degré de dépendance égal à 2.

III Données symboliques locales de test

Définition

Une donnée symbolique représente un ensemble de données réelles ou arguments d'une variable d'une fonction de base d'un bloc. Il s'agit de variable booléenne simple ou générale.

Dans la modélisation symbolique utilisée, des données locales de test distinctes d'une variable ne sont pas distinguées si elles sont traitées de façon identique pendant les propagations avant et arrière. Des traitements distincts de données distinctes seraient donc un gaspillage de temps. Ce gain en temps dû au traitement symbolique sera évalué sur des résultats expérimentaux donnés au § X.

Vecteurs symboliques locaux de test

Dans notre approche les vecteurs locaux de test d'un bloc sont associés aux fonctions de base du bloc. Ce sont des vecteurs symboliques. Un vecteur de test symbolique associé à une fonction de base d'un bloc représente un ensemble de valeurs réelles de test associées à l'ensemble des variables de la fonction. Chacune des variables d'entrée de la fonction est appelée composante du vecteur de test.

Au niveau des blocs, à chaque vecteur réel de test sont associées les sorties juste et fausse. Dans une représentation symbolique on pourrait affecter à une sortie par exemple la valeur symbolique juste Z et fausse Z*. Cependant ces deux valeurs juste et fausse génèrent les mêmes chemins de propagation avant. Ainsi une seule donnée symbolique de test est retenue pour représentée les valeurs réelles juste et fausse d'une sortie.

Dans un additionneur à deux entrées, (X, Y, Z) représente par exemple 50 valeurs réelles de test à appliquer aux entrées et 50 valeurs de sortie à observer.

Séquences symboliques simples de test

Une séquence symbolique de test est une séquence de vecteurs symboliques V1, V2, ... Vn associés à une fonction de base d'un bloc et qui représentent des données distinctes X1, X2, ...Xn à des instants symboliques t1, t2, ...tn distincts.

C'est le cas de la fonction de chargement du registre de la figure 21 modélisée avec une horloge explicite dont la valeur doit passer de 0 à 1 à deux instants symboliques successifs.

Séquences symboliques composées de test

Au niveau d'un bloc on peut être amené à exprimer un test comme une séquence de plusieurs fonctions de base. De telles séquences sont exprimées par des concaténations de vecteurs ou de séquences symboliques simples de test. Ce cas est fréquent pour les blocs comportant des éléments de mémorisation. Dans l'exemple

de la RAM de la figure 22, une séquence de vecteurs de test est la concaténation des fonctions écriture*lecture.

III. 1 Modélisation d'un vecteur symbolique en Prolog

Un vecteur symbolique affecte des valeurs symboliques et des instants symboliques aux entrées (donnée et contrôle) et sortie de la fonction à laquelle il est associé. Les instants symboliques définissent l'instant d'application des stimuli et l'instant de validation de la réponse (§ II sur la modélisation).
Plusieurs fonctions de base d'un bloc peuvent avoir le même vecteur symbolique. Dans ce cas un seul vecteur représentatif est retenu. Le format général d'un vecteur symbolique de test écrit en Prolog est le suivant.

Vs(type_bloc,fonction,[[liste (val_entrée i,instant ti)],[liste(val_sortie j,instant tj)]]).

Le terme "type_bloc" donne le type du bloc.
Le terme fonction désigne la fonction de base à laquelle le vecteur est associé.

Le troisième terme du littéral comporte deux listes. La première est une liste qui affecte à chaque entrée "entrée i", une valeur symbolique "val_entrée i" et un instant symbolique d'application "instant ti". Les entrées sont divisées en donnée primaire, contrôle et variable de mémorisation. La seconde est une liste qui affecte à chaque sortie "sortie j" une valeur symbolique "val_sortie j" et un instant symbolique de validation "instant tj". Les sorties comportent les sorties primaires et variables de mémorisation. Les instants d'application et de validation sont relatifs à un seul instant pris comme référence. Ces instants peuvent être égaux ou différents.

III. 1. 1 Exemple 1

Dans l'exemple de l'UAL de la figure 20a, un vecteur local symbolique de test est associé à la fonction de base "addition de A et B". Ce vecteur est donné comme suit:
Vs(UAL, Add, [[(X1, t), (X2, t)], [(X3, t), (X4, t)], -, [(Y, t)], -]).

Dans ce vecteur les valeurs symboliques X1, X2, X3 et X4 sont affectées respectivement aux entrées de donnée A et B et de contrôle C1 et C2 à l'instant symbolique t. D'autre part la valeur symbolique Y est affectée à la sortie S au même instant symbolique t.

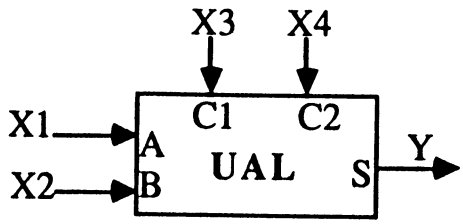


Figure 20a. Exemple d'un UAL à quatre fonctions

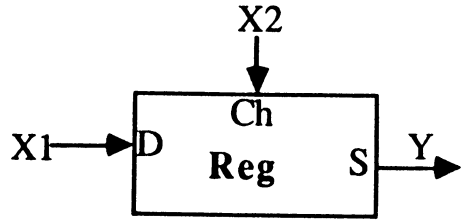


Figure 20b. Registre à horloge implicite

III. 1. 2 Exemple 2

Dans l'exemple du registre de la figure 20b le vecteur local symbolique de test est le suivant :

Vs(Reg, Charg, [[(X1, t)], [(X2, t)], -, [(Y, t+1)], -]).

Dans ce vecteur les valeurs symboliques X1 et X2 sont affectées respectivement aux entrées D et Ch à l'instant symbolique t et la valeur symbolique Y est affectée à la sortie S à l'instant symbolique t+1.

III. 2 Modélisation d'une séquence symbolique de test en Prolog

Le format général d'une séquence de vecteurs symboliques est donnée sous la forme d'un littéral écrit en Prolog.

Ss(type_bloc, fonction, [liste (Vsi(type_bloc, fonction i, -))]).

Le terme "type_bloc" donne le type du bloc.
Le terme fonction désigne la fonction de base ou le produit de fonctions de base d'un bloc auquel la séquence est associée.
Le terme Vsi(type_bloc, fonction i, -) est un vecteur symbolique. Son format est donné au paragraphe précédent § III.1. Lorsqu'il s'agit d'une séquence associée à une fonction de base "fonction i" est égal à "fonction". Dans le cas où la séquence est associée à un produit de fonctions de base "fonction" = $\prod$ "fonction i".

III. 2. 1 Exemple 1 : séquence symbolique simple

Dans l'exemple du registre de la figure 21 la séquence symbolique de test associée à la fonction de base "chargement" est la suivante.

```
Ss(Reg, chargement,     [
                [  -  , [ - , (X3, t-1)], -, -, -] ,
                [ [(X1, t)], [(X2, t), (X4, t)], -, [(Y,t+1)], - ]
                                ]
).
```

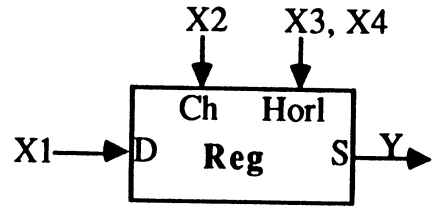


Figure 21. Registre à horloge explicite

Cette séquence comporte deux vecteurs symboliques. Dans le premier vecteur la valeur symbolique X3 est affectée au signal de l'horloge à l'instant symbolique t-1. Dans le second vecteur les valeurs symboliques X1 et X2 sont affectées respectivement aux signaux de donnée et de contrôle à l'instant t. A cet instant l'horloge prend une valeur différente. Elle est désignée par X4 à l'instant t. La valeur symbolique Y est affectée à la sortie qui est valide à l'instant t+1.

III. 2. 2 Exemple 2 : séquence symbolique composée

Soit l'exemple d'une mémoire RAM illustrée sur la figure 22.

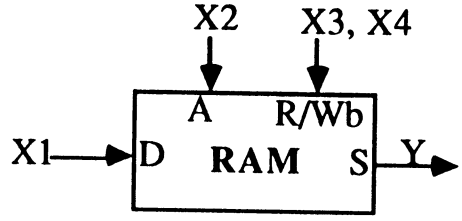


Figure 22. Mémoire RAM à horloge implicite

Les fonctions de base de la RAM sont l' "écriture d'un mot mémoire" et la "lecture d'un mot mémoire". Seule, aucune de ces deux fonctions ne permet d'appliquer les stimuli de test et d'avoir la réponse sur une sortie primaire du bloc. Par contre le produit de ces deux fonctions le permet. Une séquence symbolique est alors associée à la fonction produit "écriture*lecture". Elle est donnée comme suit.

```
Ss(RAM, écriture*lecture,      [
            [ [(X1, t)], [(X2, t), (X3, t)], -, -, -] ,
            [  -  , [(X2, t+1), (X4, t+1)], -, [(Y,t+2)], - ]
                                  ]
  ).
```

Cette séquence comporte deux vecteurs symboliques. Le premier est un vecteur symbolique associé à la fonction d'écriture et le second est un vecteur symbolique

associé à la fonction de lecture. Dans le premier vecteur les valeurs symboliques X1, X2 et X3 sont affectées respectivement aux signaux de donnée, d'adresse et de contrôle à l'instant symbolique t.
Dans le second vecteur l'adresse garde sa valeur à l'instant t+1 alors que les signaux de contrôle prennent une valeur différente. Elle est désignée par X4 à l'instant t+1.
La valeur symbolique Y est affectée à la sortie qui est valide à l'instant t+2.

IV Génération des vecteurs symboliques globaux de test d'un circuit

Cette procédure est résumée sur l'organigramme de la figure 23. Elle comprend deux phases principales.

La première est une phase de prétraitement qui consiste à évaluer la proximité "structurelle" des fonctions de base de chacun des blocs par rapport aux entrées et sorties primaires du circuit. Des paramètres de proximité sont alors définis. Ils constituent un des critères du choix entre les fonctions des blocs à traverser au cours des propagations avant et arrière.

La seconde phase consiste à chercher les chemins symboliques qui permettent de contrôler et d'observer les vecteurs symboliques de test des blocs et fournissent le squelette du programme de test du circuit. Le contrôle d'un vecteur de test d'un bloc est effectué par un processus de propagation arrière à partir du bloc jusqu'à atteindre les entrées primaires du circuit. L'observation du vecteur est effectuée par un processus de propagation avant à partir du bloc jusquà atteindre les sorties primaires du circuit.

Un chemin symbolique est recherché pour chaque vecteur symbolique local de test d'un bloc. La recherche d'un chemin associé à un vecteur local échoue, si une composante du vecteur local de test n'a pu être propagée. On distingue les échecs qui sont dûs à un conflit et pour lesquels un algorithme de "propagation retardée" essaie de résoudre le conflit. Dans le cas où l'algorithme ne réussit pas, une amélioration de la testabilité est proposée en insérant un ensemble de points de test. L'ensemble global des points de test à insérer dans le circuit sera par la suite minimisée.

Le paragraphe suivant § V présente l'algorithme d'évaluation de la proximité des fonctions par rapport aux entrées et sorties primaires du circuit. La méthode de recherche d'un chemin symbolique est présentée dans le § VI. Le § VII est consacré au traitement des problèmes rencontrés au cours des propagations avant et arrière. Le § VIII est dédié au calcul des points de test et de leur minimisation afin d'améliorer la testabilité du circuit.

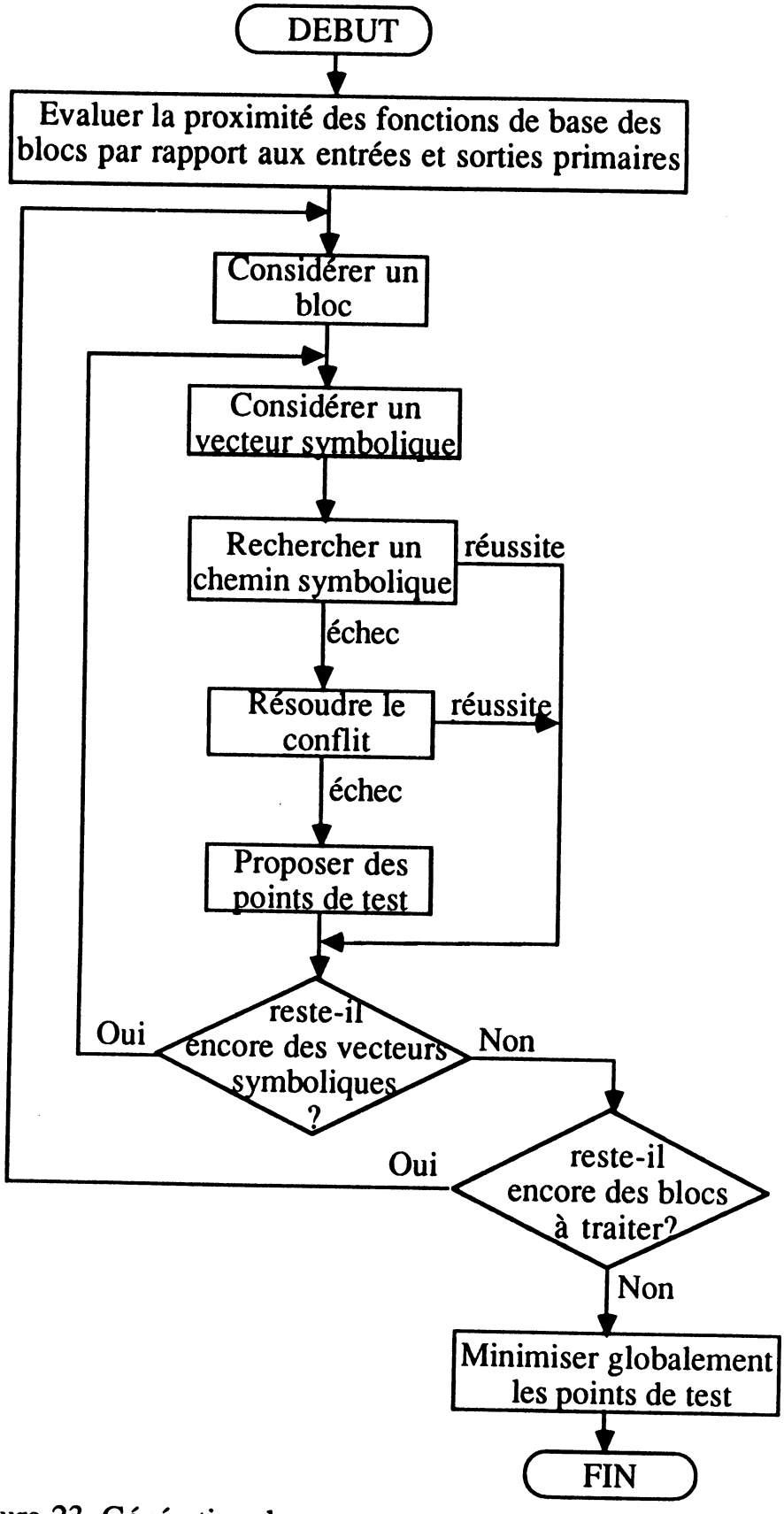


Figure 23. Génération des vecteurs symboliques globaux de test

V Evaluation de la proximité des fonctions de base par rapport aux entrées et sorties primaires du circuit

Soit e_1, ..., e_i, ..., e_n les entrées d'une fonction de base d'un bloc. Soit pe(ei) le nombre minimal de blocs traversés pour accéder à ei depuis les entrées primaires du circuit.
La proximité Pe d'une fonction de base par rapport aux entrées primaires exprime le nombre minimal de blocs à traverser depuis les entrées primaires jusqu'à atteindre toutes les entrées de cette fonction.
Pe (fonction) = max ($pe(e_1)$, $pe(e_2)$, ..., $pe(e_i)$, ...$pe(e_n)$).

Soit s une sortance multiple d'une fonction de base d'un bloc et soit s_1, ..., s_i, ..., s_n les branches de la sortance multiple. Soit $ps(s_i)$ le nombre minimal de blocs traversés pour propager la sortie s_i depuis la sortie du bloc jusqu'aux sorties primaires du circuit.
La proximité Ps d'une fonction de base par rapport aux sorties primaires exprime le nombre minimal de blocs à traverser depuis sa sortie la plus proche des sorties primaires du circuit.
Ps (fonction) = min ($ps(s_1)$, ...,$ps(s_i)$, ..., $ps(s_n)$).
Le calcul des proximités Pe et Ps d'une fonction de base par rapport aux entrées et sorties primaires du circuit nécessite le calcul des termes pe des entrées et et ps des sorties des fonctions de base. Pour calculer pe et ps le circuit est considéré comme un graphe dans lequel les nœuds représentent les fonctions de base des blocs et les arcs représentent les connexions entre les blocs du circuit.

L'algorithme qui calcule les termes pe est un simple parcours du graphe "par largeur d'abord" à partir des entrées primaires du circuit. A l'initialisation on part avec pe=0 qu'on affecte à toutes les entrées primaires du circuit. Un nœud est parcouru si toutes ses entrées sont affectées. Lorsqu'on parcourt un nœud, pe est incrémenté et il est affecté à tous les arcs sortants non encore affectés. L'algorithme s'arrête lorsque les termes pe des entrées de tous les nœuds sont calculés.

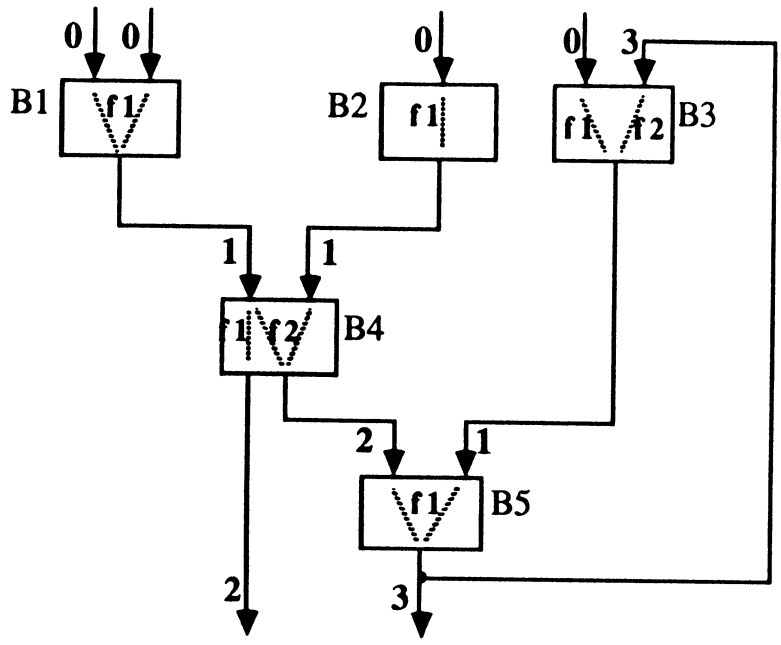


Figure 24. Proximité des entrées des fonctions par rapport aux entrées primaires

L'algorithme qui calcule les termes ps est similaire à l'algorithme de calcul des pe. Cet algorithme part à partir des sorties primaires du circuit. A l'initialisation ps=0 et il est affecté à toutes les sorties primaires du circuit. Un nœud est parcouru si sa sortie est affectée. Lorsqu'un nœud est parcouru ps est incrémenté et il est affecté à tous les arcs rentrants non encore affectés. L'algorithme s'arrête lorsque les termes ps des sorties de tous les nœuds sont calculés.

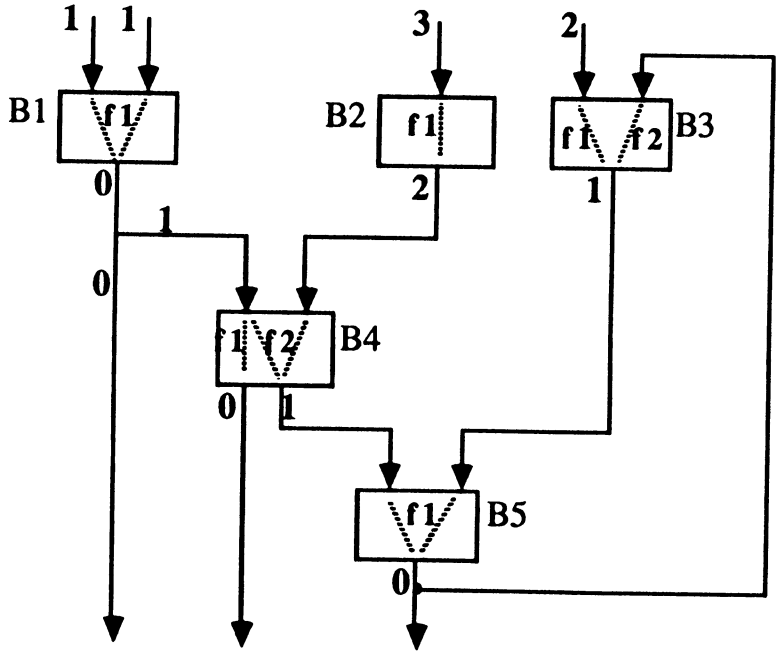


Figure 25. Proximité des sorties des fonctions par rapport aux sorties primaires

VI Recherche d'un chemin symbolique

Etant donné un bloc et un vecteur symbolique local de test, la recherche d'un chemin symbolique associé à ce vecteur consiste à satisfaire le but initial suivant "appliquer et propager le vecteur symbolique local de test". Le vecteur symbolique est appelé vecteur-but initial et le bloc est appelé bloc-but initial.
Le but initial est divisé en deux sous-buts initiaux. Le premier consiste à justifier les valeurs symboliques aux entrées-but du bloc-but initial, depuis les entrées primaires du circuit. Le second consiste à propager les valeurs symboliques aux sorties-but du bloc-but initial, jusqu'aux sorties primaires du circuit. Ces deux sous-buts initiaux deviennent des buts qu'on essaie de satisfaire successivement par des processus de propagation arrière et avant.
L'algorithme qui correspond à la recherche d'un chemin symbolique est donc le suivant.

propager en arrière les valeurs symboliques aux entrées-but du bloc-but initial jusqu'à atteindre les entrées primaires du circuit.

propager en avant les valeurs symboliques aux sorties-but du bloc-but initial jusqu'à atteindre les sorties primaires du circuit.

Déclaration d'un but en Prolog

Au vecteur symbolique Vs(type_bloc,-, [[liste (val_sym_entrée i,instant ti)], [liste (val_sym_sortie j, instant tj)]]) correspond le but initial suivant exprimé sous la forme de la clause suivante en language Prolog.

```
but_initial (bloc):-
propagation_arrière ( liste (val_sym_entrée i,instant ti) ),
propagation_avant   ( liste (val_sym_sortie j, instant tj) ).
```

Les deux sous-buts initiaux associés au but initial sont donnés par les deux clauses suivantes.

```
sous_but_initial (bloc):-
propagation_arrière ( liste (val_sym_entrée i,instant ti) ).

sous_but_initial (bloc):-
propagation_avant ( liste (val_sym_sortie j, instant tj) ).
```

Exemple

Soit l'exemple du bloc de la figure 26 et soit le vecteur symbolique local de test suivant.

Vs (type_bloc, f1, [[(X1, t), (X2, t)], [(X3, t)], -, [(Y, t+1)], -]).

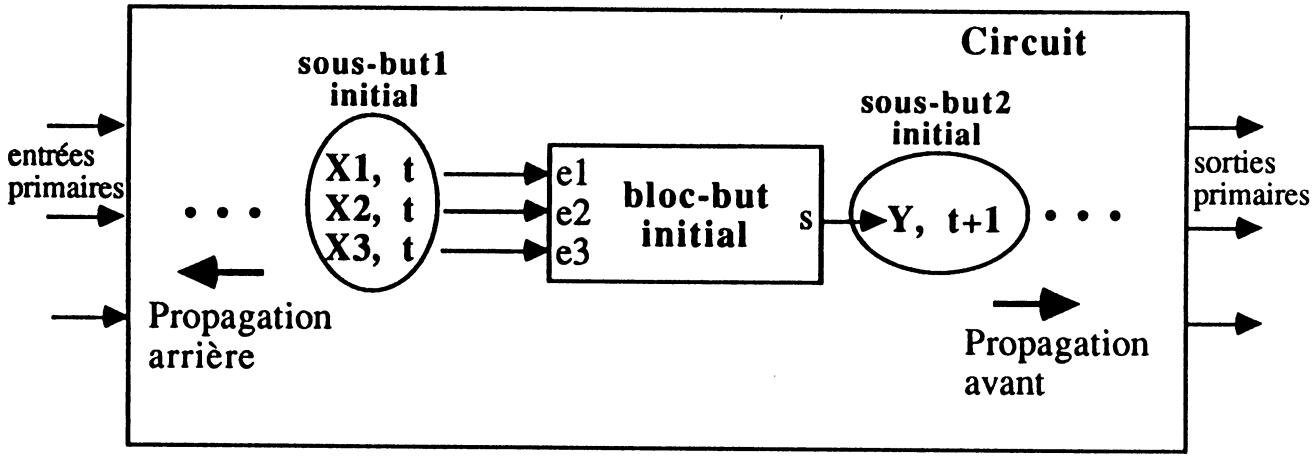


Figure 26. Décomposition d'un but en deux sous-but

Le but initial à satisfaire correspondant au vecteur symbolique est donné par la clause suivante.

but_initial (bloc-but):-
propagation_arrière ([(X1, t), (X2, t)], [(X3, t)], -),
propagation_avant ([(Y, t+1)], -).

Les deux litéraux de cette clause définissent deux sous-buts initiaux.
sous_but_initial (bloc-but):-
propagation_arrière ([(X1, t), (X2, t)], [(X3, t)], -).

sous_but_initial (bloc-but):-
propagation_avant ([(Y, t+1)], -).

Propagations avant et arrière des valeurs symboliques de test

Ces propagations sont effectuées à travers les fonctions directes et inverses associées aux fonctions de base des blocs du circuit en franchissant des barrières temporelles. On parlera de propagations spatio-temporelles. Durant ces propagations les valeurs symboliques subissent des transformations et les instants symboliques sont incrémentés ou décrémentés.

Le processus de propagation est à décision multiple. Dans le cas où la propagation peut se faire à travers plusieurs fonctions, une seule fonction est choisie. En cas de conflit le dernier choix est effacé et un autre choix est effectué parmi les choix qui restent. Ce choix est effectué selon plusieurs critères liés aux caractéristiques des fonctions de base des blocs.

VI. 1 Propagation arrière

La propagation arrière consiste à déterminer un chemin symbolique entre les entrées primaires du circuit et les entrées-but d'un bloc-but initial. Elle est appliquée à partir du bloc-but jusqu'à atteindre les entrées primaires du circuit. Pendant une propagation arrière des entrées-but d'un bloc-but initial, les entrées des blocs traversés deviennent à leur tour des entrées-but à satisfaire.

VI. 1. 1 Algorithme

Cet algorithme est itératif. Chaque itération comporte deux phases. La première détermine les "blocs-père" des entrées-but. Elle fait appel à la description des interconnexions entre les blocs du circuit. La seconde phase traverse chacun des blocs-pères depuis ses sorties jusqu'à ses entrées.

Si on schématise le circuit par un graphe ayant les fonctions des blocs comme nœuds et les connexions comme arcs, la propagation arrière revient à effectuer un parcours "en profondeur d'abord" du graphe. Le résultat est une arborescence dont la racine est formée par le bloc-but initial et les feuilles sont les plots d'entrées du circuit. Des valeurs et des instants symboliques sont affectés aux arcs de l'arborescence.
L'arborescence est stockée au fur et à mesure qu'elle est construite. On stocke le but courant auquel on affecte un indice sous la forme d'une liste qui rappelle son ordre dans l'arborescence. A l'initialisation le bloc courant est le bloc-but initial et le but courant est composé de la liste des valeurs et instants symboliques des entrées-but du but initial.

Cet algorithme est décrit comme suit.

bloc courant <- bloc-but initial
affecter le indice [i,0] à chaque entrée-but i du bloc-but initial et stocker le
(a) **Tant que** entrées-but courant ≠ entrées primaires **faire**
(b) Chercher les blocs-père des entrées-but courant
Pour tout bloc-père **faire**

traverser le bloc-père
bloc-courant<- bloc-père
stocker les nouvelles entrées-but
retour à (a)
fait
fait.

VI. 1. 2 Traversée d'un bloc-père

La traversée d'un bloc-père est effectuée à travers une fonction inverse d'une fonction de base de ce bloc-père. Cette fonction permet de déterminer les valeurs symboliques et les instants symboliques d'application des entrées du bloc-père. La traversée d'un bloc-père est effectuée en trois pas.

Pas1. Etablir la liste des fonctions inverses associées aux fonctions de base du bloc-père qui permettent de propager la sortie vers les entrées.
Pas2. Choisir une fonction parmi les fonctions inverses de la liste.
Pas3. Définir les valeurs symboliques des entrées et les instants de leur application.

Liste des fonctions inverses associées aux fonctions de base du bloc-père

Cette liste contient les fonctions inverses du bloc, capables de modifier la sortie souhaitée. Cette liste est extraite à partir des clauses de dépendance des fonctions de base du bloc-père.

Heuristique de choix d'une fonction inverse

Cette heuristique consiste à effectuer le choix d'une fonction, parmi plusieurs fonctions inverses d'un bloc. Elle est fondée sur deux facteurs qui sont la complexité des fonctions inverses et leur proximité par rapport aux entrées ou sorties primaires du circuit. Ainsi on choisit la fonction inverse

1. la plus proche des entrées primaires
2. la "moins" complexe

La proximité des fonctions inverses par rapport aux entrées est déterminée dans une phase de prétraitement.
Le choix de la fonction la moins complexe fait appel aux clauses de caractéristiques des fonctions de base. Choisir une fonction inverse la "moins" complexe sous entend qu'un ordre de complexité est établi. Les fonctions inverses sont ordonnées comme suit selon leur complexité croissante en traversée arrière.

1. la fonction transparente

2. la fonction en mode quasi-transparent
3. la fonction inverse calculable de type SD
4. la fonction inverse calculable de type MD

Les fonctions de type MD sont aussi classées en fonction de leur degré de dépendance. Ce paramètre est donné dans les clauses de caractéristiques des fonctions de base auxquelles les fonctions inverses sont associées.

Evaluation des entrées

Lors de la traversée d'un bloc les valeurs symboliques aux sorties du bloc subissent une transformation. Cette transformation est définie par la fonction inverse appliquée. Elle détermine les valeurs des entrées du bloc. On distingue deux catégories d'entrées à évaluer.

Dans la première la valeur des entrées est une expression symbolique de la valeur de sortie.

Dans la seconde la valeur des entrées est symbolique mais indépendante de la valeur de la sortie à propager. C'est le cas des entrées de contrôle qui activent la fonction inverse et des entrées de donnée d'une fonction quasi-transparente ou calculable de type MD. Dans le modèle de description de la fonction inverse ces entrées sont évaluées par des valeurs numériques. La traversée arrière affecte des valeurs symboliques à ces entrées mais leur associe également leur valeur numérique sous la forme de contraintes. Ces contraintes sont prises en compte pour détecter la présence des conflits (§ VII).

Instants d'application des entrées

Les instants symboliques d'application des entrées sont établis en fonction des instants symboliques de validation de la sortie. Dans le cas d'une fonction inverse combinatoire les instants symboliques d'application des entrées sont égaux à l'instant symbolique de validation de la sortie. Dans le cas où la fonction inverse est séquentielle les instants symboliques d'application des entrées sont décrémentés par rapport aux instants de validation de la sortie.

VI. 1. 3 Exemple

Considérons le circuit illustré sur la figure 27. Soit B1 le bloc-but initial et soit le sous-but initial qui consiste à propager en arrière les valeurs X1 et X2 des entrées e1 et e2 du bloc B1 à l'instant t.

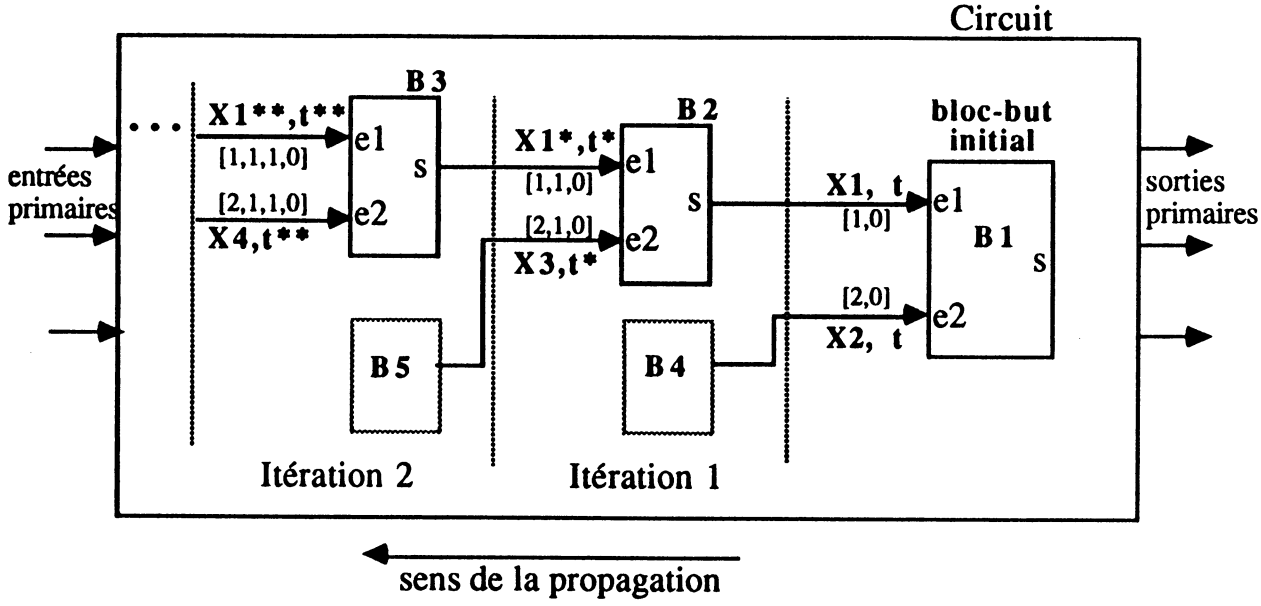


Figure 27. Propagation arrière à travers deux itérations

La propagation affecte les indices [1,0] et [2,0] aux entrées-but courant e1 et e2 respectivement. Ces entrées ne sont pas des entrées primaires du circuit. Une première itération de l'algorithme est déclenché. Elle cherche la liste des blocs-père des entrées e1 et e2 du bloc B1. Cette recherche se sert des clauses de connexion et la liste résultante est [B2, B4]. Chacune de ces entrées-but devient un sous-but. La propagation arrière essaie de propager les deux sous-buts successivement. Elle considère la première entrée-but d'indice [1,0]. En utilisant les clauses de dépendance du bloc père B2, l'algorithme dresse la liste des fonctions inverses ayant s comme sortie et en choisit une fonction inverse selon l'algorithme de choix donné au § VI.1.2. La fonction choisie détermine les valeurs X1* et X3 des entrées e1 et e2 du bloc B2 respectivement et les instants symboliques t* de leur application. Les indices [1,1,0] et [2,1,0] sont affectés aux entrées e1 et e2 du bloc B2. e1 et e2 deviennent les entrées-but courant et B2 le bloc-but courant pour l'itération suivante.

VI. 2 Propagation avant

La propagation avant consiste à déterminer un chemin entre les sorties-but du bloc-but initial et les sorties primaires du circuit. Elle est appliquée à partir du bloc-but initial jusqu'à atteindre les sorties primaires du circuit.

VI. 2. 1 Algorithme

Cet algorithme est itératif. Chaque itération comporte trois phases. La première détermine les "blocs-fils" d'une sortie-but courant en faisant appel à la description

des interconnexions entre les blocs du circuit. La seconde phase choisit un des blocs-fils à traverser. La troisième traverse le bloc-fils choisi depuis ses entrées jusqu'à ses sorties.

Si on schématise le circuit par un graphe ayant les fonctions des blocs comme nœuds et les connexions comme arcs, la propagation avant revient à effectuer un parcours "en profondeur d'abord" du graphe. Le résultat est une arborescence dont la racine est formée le bloc-but initial et les feuilles sont les plots de sorties du circuit. Des valeurs et des instants symboliques sont affectés aux arcs de l'arborescence.

L'arborescence est stockée au fur et à mesure qu'elle est construite. On stocke le but courant auquel on affecte un indice sous la forme d'une liste qui rappelle son ordre dans l'arborescence. A l'initialisation le bloc courant est le bloc-but initial et le but courant comporte la liste des valeurs et des instants symboliques des sorties-but du bloc but initial.

Cet algorithme est décrit comme suit.

bloc courant <- bloc-but initial
affecter le indice [0,i] à chaque sortie-but i du bloc-but initial et stocker le
Pour tout sortie-but courant **faire**
(a) **Tant que** sortie-but courant ≠ sortie primaire **faire**
(b) Chercher les blocs-fils de la sortie-but courant
Choisir un bloc-fils et une fonction de base à traverser
Appeler la propagation arrière si la fonction est de type MD
Traverser le bloc-fils choisi
bloc-courant<- bloc-fils
stocker la nouvelle sortie-but
retour à (a)
fait
fait.

VI. 2. 2 Heuristique de choix d'un bloc-fils

Les critères de choix d'un bloc-fils parmi une liste de blocs-fils sont liés aux fonctions directes associées aux fonctions de base des blocs fils. Le bloc fils choisi est celui qui a la fonction directe "la plus simple" et qui permet de propager l'entrée

vers une sortie du bloc. La liste des fonctions directes d'un bloc-fils est extraite à partir des clauses de dépendance des fonctions de base du bloc.
L'heuristique de choix de la "plus simple fonction directe", parmi toutes les fonctions directes possibles des différents blocs-fils, est fondée sur deux facteurs qui sont la complexité des fonctions et leur proximité par rapport aux sorties primaires du circuit. Ainsi on choisit la fonction de base

1. la plus proche des entrées primaires
2. la "moins" complexe

La proximité des fonctions directes par rapport aux sorties primaires du circuit est déterminée dans une phase de prétraitement.
Le choix de la fonction directe la moins complexe fait appel aux clauses des caractéristiques des fonctions de base auxquelles sont associées les fonctions directes. Choisir une fonction la "moins" complexe sous entend qu'un ordre de complexité est établi. Les fonctions directes sont ordonnées comme suit dans un ordre de complexité croissante.

1. la fonction transparente
2. la fonction en mode quasi-transparent
3. la fonction directe calculable, injective et de type SD
4. la fonction directe calculable, injective et de type MD
5. la fonction directe calculable, non injective et de type SD
6. la fonction directe calculable, non injective et de type MD

Les fonctions de type MD sont aussi classées en fonction de leur degré de dépendance croissante.

VI. 2. 3 Appel de la propagation arrière

La propagation arrière est appelée dans le cas où la fonction directe choisie du bloc-fils est du type MD, afin de justifier la valeur des autres entrées de la fonction.
La valeur à justifier est symbolique mais indépendante de la valeur de l'entrée à propager. C'est le cas des entrées de contrôle qui activent la fonction directe et des entrées de donnée d'une fonction directe de type MD. Dans le modèle de description de la fonction directe ces entrées sont évaluées avec des valeurs numériques. La traversée avant affecte des valeurs symboliques à ces entrées mais leur associe également leur valeur numérique sous la forme de contraintes. Ces contraintes sont prises en compte pour détecter la présence d'un conflit (§ VII).
D'autre part les instants symboliques d'application de ces entrées sont exprimés en fonction de l'instant symbolique d'application de l'entrée à propager. Dans le cas d'une fonction directe combinatoire l'instant symbolique d'application des entrées

est égal à l'instant symbolique d'application de l'entrée à propager. Dans le cas où la fonction directe est séquentielle les instants symboliques d'application des entrées et de validation de la sortie sont décrémentés par rapport à l'instant d'application de l'entrée à propager.

VI. 2. 4 Traversée d'un bloc-fils

La traversée d'un bloc-fils est effectuée à travers la fonction directe choisie. Cette fonction permet de déterminer les valeurs symboliques et les instants symboliques de validation de la sortie du bloc-fils.

Evaluation de la sortie

Lors de la traversée d'un bloc la valeur symbolique des entrées du bloc subit une transformation. Cette transformation est définie par la fonction directe appliquée.

Instants de validation de la sortie

L'instant symbolique de validation de la sortie est exprimé en fonction de l'instant symbolique d'application de l'entrée à propager. Dans le cas d'une fonction directe combinatoire l'instant symbolique de validation de la sortie est égal à l'instant symbolique d'application de l'entrée à propager. Dans le cas où la fonction directe est séquentielle l'instant symbolique de validation de la sortie est incrémenté par rapport à l'instant d'application de l'entrée à propager.

VI. 2. 5 Exemple

Soit le circuit illustré sur la figure 28. Soit B1 le bloc-but initial et soit le sous-but initial qui consiste à propager en avant les valeurs Y1 et Y2 des sorties s1 et s2 du bloc B1 à l'instant t.

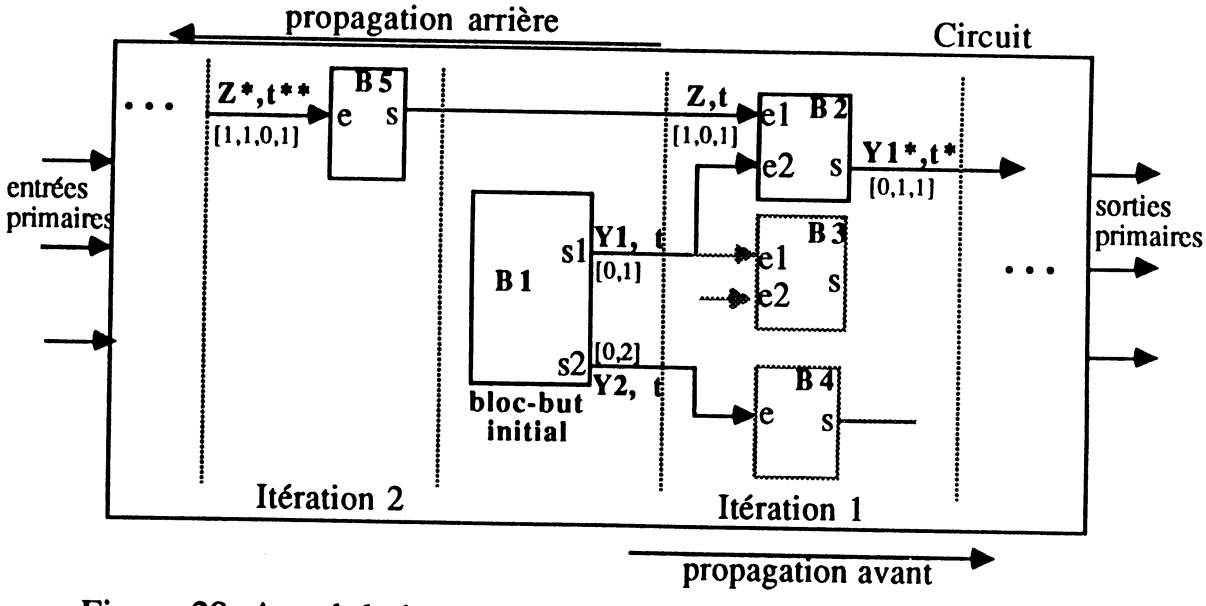


Figure 28. Appel de la propagation arrière pendant la propagation avant

La propagation avant affecte les indices [0,1] et [0,2] aux sorties-but s1 et s2 respectivement. Les sorties s1 et s2 sont considérées l'une après l'autre. La sortie s1 n'étant pas une sortie primaire du circuit, l'algorithme essaie de propager en avant cette sortie. Il cherche la liste des blocs-fils de la sortie s1 du bloc B1. Cette recherche se sert des clauses de connexion et la liste résultante est [B2, B3]. En utilisant les clauses de dépendance du bloc B2 et B3, on dresse la liste des fonctions directes ayant s1 comme entrée et en choisit une fonction directe selon l'algorithme de choix donné au § VI.2.2. La fonction choisie détermine le choix du bloc-fils. L'algorithme choisit une fonction directe de type MD dans le bloc B2. La fonction détermine la valeur Y1* de la sortie s du bloc B2 et les l'instant symbolique t* de sa validation ainsi que la valeur Z de la deuxième entrée e1 de la fonction et l'instant t de son application. Les indices [1,0,1] et [0,1] sont affectés à l'entrée e1 et la sortie s du bloc B2. La propagation arrière de l'entrée e1 constitue le but de l'itération suivante. Le but lié à la propagation avant de la sortie s du bloc B2 est traité en second.

VII Problèmes rencontrés au cours des propagations

Les propagations avant et arrière d'un but initial génèrent une arborescence de propagation où les nœuds sont des fonctions directes ou inverses des fonctions de base des blocs et les arcs représentent les équipotentielles connectant les blocs. Les arcs de l'arborescence sont étiquetés par le nom des équipotentielles et leurs valeurs symboliques affectées à des instants symboliques au cours de la propagation.
Considérons l'exemple du circuit de la figure 29. Soit le but initial qui consiste à contrôler les entrées A et B et observer la sortie C du bloc-but B5. L'arborescence de propagation correspondant au but initial est donnée sur la figure 30. Les nœuds de l'arborescence indiquent les blocs traversés ainsi que les fonctions de base utilisées lors de la traversée.

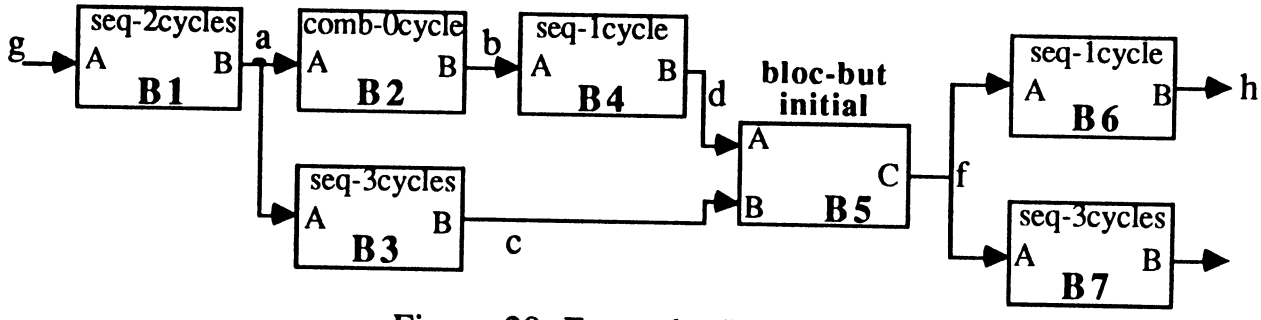


Figure 29. Exemple d'un circuit

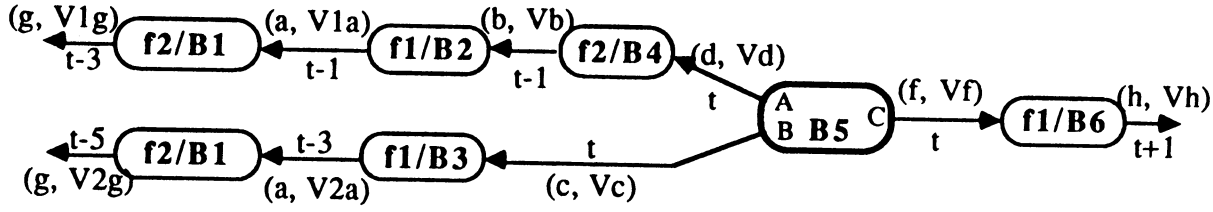


Figure 30. Arborescence de propagation du but initial

Définition 1. Etant donné une arborescence de propagation d'un but, on appelle *barrière temporelle avant d'ordre i par rapport à un bloc-but* B, l'ensemble des couples (équipotentielle, valeur) où l'équipotentielle est évaluée au cours d'une propagation avant, i instants symboliques après l'évaluation des sorties du bloc-but B. Elle sera notée $T_i^+(B)$.

Dans l'exemple du circuit de la figure 29 on définit les barrières temporelles avant suivantes par rapport au bloc B5.
$T_1^+(B5) = \{(h, Vh)\}$.

Définition 2. Etant donné une arborescence de propagation d'un but, on appelle *barrière temporelle arrière d'ordre i par rapport à un bloc-but* B, l'ensemble des couples (équipotentielle, valeur) où équipotentielle est évaluée au cours d'une propagation arrière, i instants symboliques avant l'évaluation des entrées du bloc-but B. Elle sera notée $T_i^-(B)$.

Dans l'exemple du circuit de la figure 29 on définit les barrières temporelles suivantes par rapport au bloc B5.

$T_1^-(B5) = \{(c, Vc), (b, Vb)\}$, $T_3^-(B5) = \{(b, Vb\}$ et $T_5^-(B5) = \{(a, Va)\}$.

Définition 3. On dit qu'un but génère une "oscillation" si un chemin de son arborescence de propagation contient (au moins) deux arcs étiquetés par la même équipotentielle.

Définition 4. Une arborescence de propagation manifeste un *conflit* lorsqu'une barrière temporelle par rapport à un bloc de l'arborescence comporte deux éléments (au moins) pour lesquels une équipotentielle est évaluée deux fois avec deux valeurs incompatibles. Deux valeurs symboliques sont dites incompatibles si les contraintes numériques éventuelles associées à ces deux valeurs sont distinctes.

Considérons l'exemple du circuit de la figure 31a.

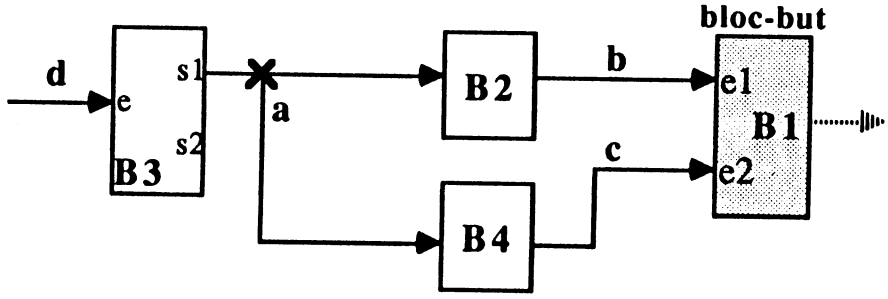


Figure 31a. Exemple d'un circuit

L'arborescence de propagation du but consistant à propager en arrière les valeurs X et Y des entrées e1 et e2 du bloc B1 est donnée sur la figure 31b. La barrière temporelle d'ordre 1 est donnée par $T_1^-(B1) = \{(a, f1^{-1}(X)), (a, f1^{-1}(Y))\}$. Un conflit se manifeste à l'équipotentielle "a" si $f1^{-1}(X) \neq f1^{-1}(Y)$.

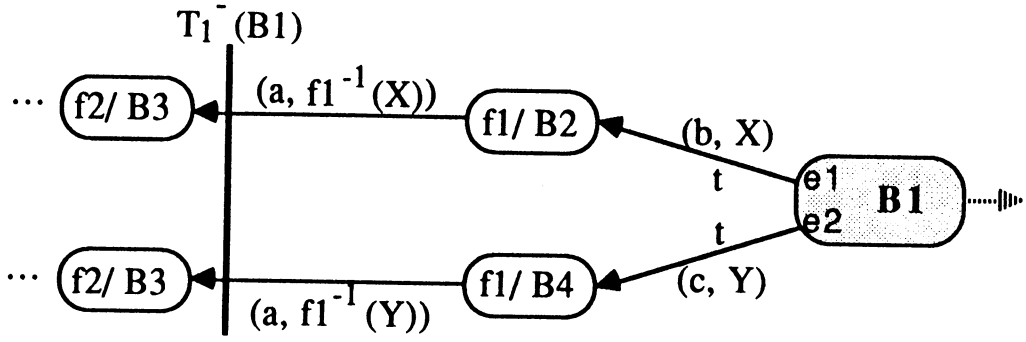


Figure 31b. Arborescence de propagation

VII. 1 Traitement des oscillations

Dans l'exemple de la figure 32, la propagation arrière de l'entrée e1 du bloc-but B1 génère une oscillation (figure 33a).

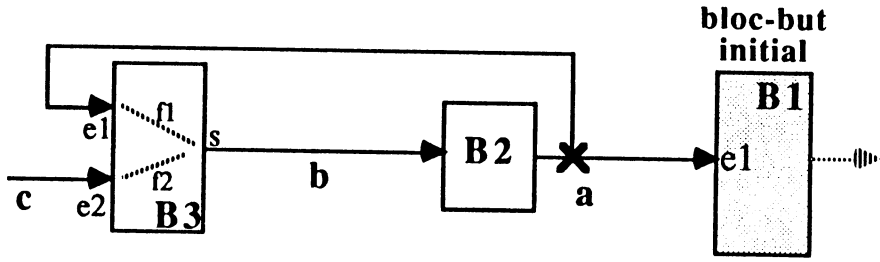


Figure 32. Exemple d'un circuit

La détection d'une oscillation conduit systématiquement à un échec de la propagation pour éviter d'effectuer des cycles infinis. Une procédure de retour arrière est appelée afin de choisir un autre chemin de propagation s'il y a lieu.

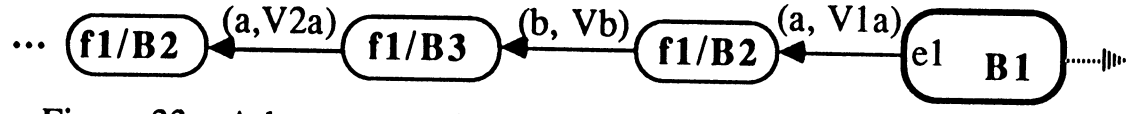


Figure 33a. Arborescence de propagation comportant une oscillation

Dans le cas de l'oscillation de la figure 33a, un retour arrière modifie le choix de la fonction f1 du bloc B3. C'est la fonction f2 qui est choisie pour traverser le bloc B3 (figure 33b). L'ensemble des entrées {e2} de la fonction f2 ne contient pas l'entrée e1 qui a conduit à l'oscillation.

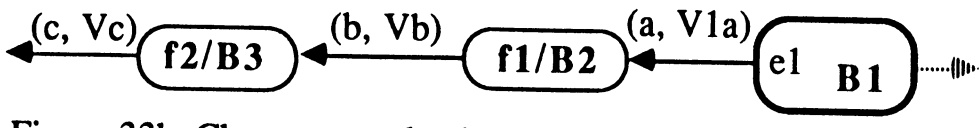


Figure 33b. Changement du choix de la fonction inverse de B3

VII. 2 Traitement des conflits

La détection d'un conflit ne génère pas systématiquement un échec. Un algorithme de résolution de conflit est appelé en premier lieu. Dans le cas où la résolution échoue on a recours à la procédure de retour arrière pour remettre en cause le dernier chemin calculé dans l'arborescence de propagation.

VII. 2. 1 Résolution du conflit par un algorithme de retardement

La résolution d'un conflit qui se manifeste entre deux ou plusieurs chemins d'une arborescence tire profit des éléments de mémorisation traversés par les chemins.
Cette procédure consiste à retarder l'instant d'application de la valeur de l'équipotentielle conflictuelle imposée par l'un des chemins et la sauvegarder dans un élément de mémorisation traversé par ce chemin. On précise que retarder l'instant d'application d'une valeur d'une équipotentielle du circuit revient à l'appliquer à un instant antérieur. Ceci revient à faire passer une équipotentielle du circuit d'une barrière temporelle à une autre barrière d'ordre supérieur.

Considérons l'exemple de la figure 34. Soit le but qui consiste à propager en arrière les valeurs X et Y des entrées e1 et e2 du bloc B1. L'arborescence de propagation associée à ce but est donnée sur la figure 35.

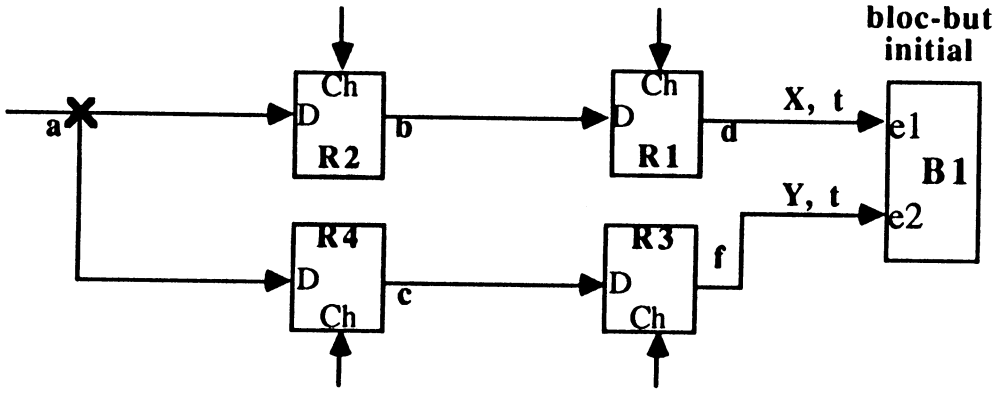


Figure 34. Exemple de circuit

Dans l'arborescence de propagation les chemin1 et chemin2 génèrent un conflit à l'équipotentielle "a" au temps t-2 puisque $T_2^-(B1) = \{ (a, f1^{-1}(f1^{-1}(X))), (a, f1^{-1}(f1^{-1}(Y))) \}$.

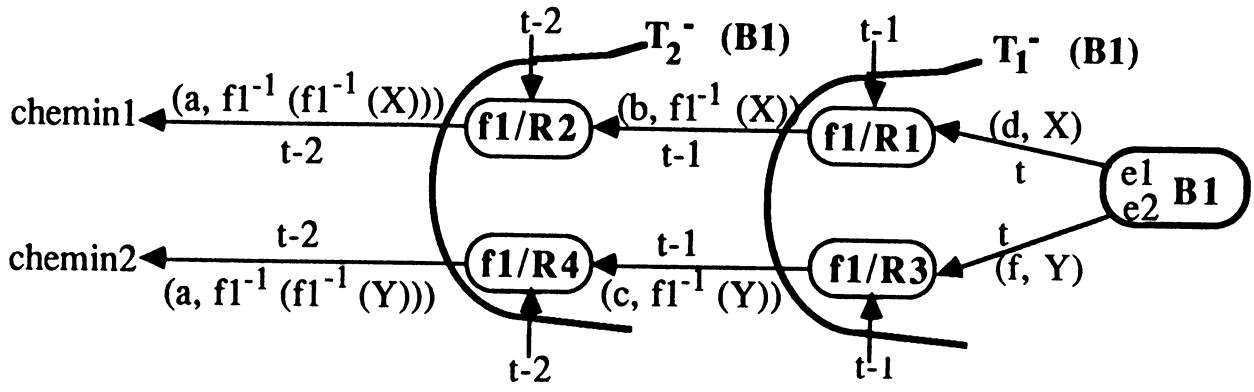


Figure 35. Arborescence de propagation

Les valeurs des équipotentielles du circuit aux différents instants symboliques sont données dans le tableau suivant.

équipotentielle	instant t	instant t-1	instant t-2
a			$f1^{-1}(f1^{-1}(X))$ et $f1^{-1}(f1^{-1}(Y))$
b		$f1^{-1}(X)$	
c		$f1^{-1}(Y)$	
d	X		
f	Y		

Pour résoudre le conflit de l'exemple donné sur la figure 34, l'algorithme de retardement propose d'appliquer à l'équipotentielle "a" la valeur $f1^{-1}(f1^{-1}(X))$ à l'instant t-3 au lieu de t-2 et de sauvegarder cette valeur dans le registre R2 traversé par le chemin1 (figure 36).

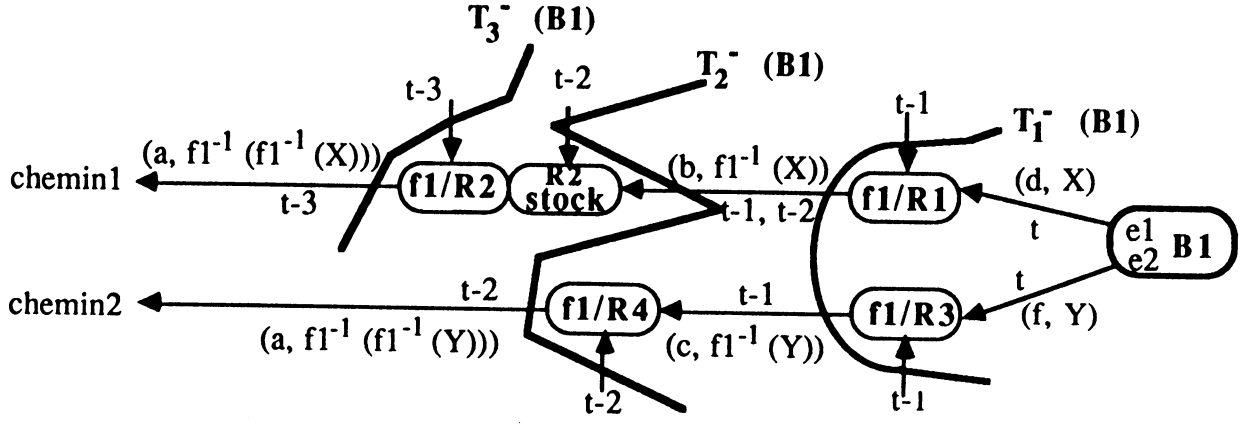


Figure 36. Résolution du conflit à l'équipotentielle "a"

Les nouvelles valeurs des équipotentielles aux différents instants symboliques sont données dans le tableau suivant.

équipotentielle	instant t	instant t-1	instant t-2	instant t-3
a			$f1^{-1}(f1^{-1}(Y))$	$f1^{-1}(f1^{-1}(X))$
b		$f1^{-1}(X)$	$f1^{-1}(X)$	
c		$f1^{-1}(Y)$		
d	X			
f	Y			

La sauvegarde d'une valeur d'une équipotentielle dans un élément de mémorisation est accompagnée d'une interdiction de tout autre chargement dans cet élément tant que la valeur sauvegardée n'a pas été stockée dans un autre élément de mémorisation appartenant au chemin. L'algorithme génère des interdictions sur les valeurs du contrôle du chargement sous la forme d'un littéral dont le format est le suivant.
interdire(bloc, liste_signaux_contrôle, liste_instant, liste_valeur).

Dans l'arborescence de la figure 36, l'algorithme charge la valeur $f1^{-1}(f1^{-1}(X))$ dans le registre R2 à l'instant t-3 et interdit l'utilisation de ce registre à t-2 (dans l'intervalle [t-2, t-1[). En effet si on utilise le registre R2 à l'instant t-2, on perd la valeur $f1^{-1}(X)$ qui n'est chargée qu'à l'instant t-1 dans le registre R1. Le littéral d'interdiction générée est
interdire(R2, [Ch], [t-2], [0]).
Les valeurs des signaux de chargement aux différents instants sont données dans le tableau suivant.

commande chargement	instant t	instant t-1	instant t-2	instant t-3
Ch(R2)			0	1
Ch(R4)			1	
Ch(R1)		1		
Ch(R3)		1		

VII. 2. 2 Ordonnancement dans le traitement des conflits

Le traitement des conflits générés par les chemins d'une arborescence est effectué par une approche dichotomique. Les traitements se font progressivement par sous-arborescences emboitées en partant des feuilles. Dans l'exemple de la figure 37, le conflit qui se manifeste dans la sous-arborescence 1 (ou 2) est traité d'abord, puis celui de la sous-arborescence 2 (ou 1) et finalement celui de la sous-arborescence 3.

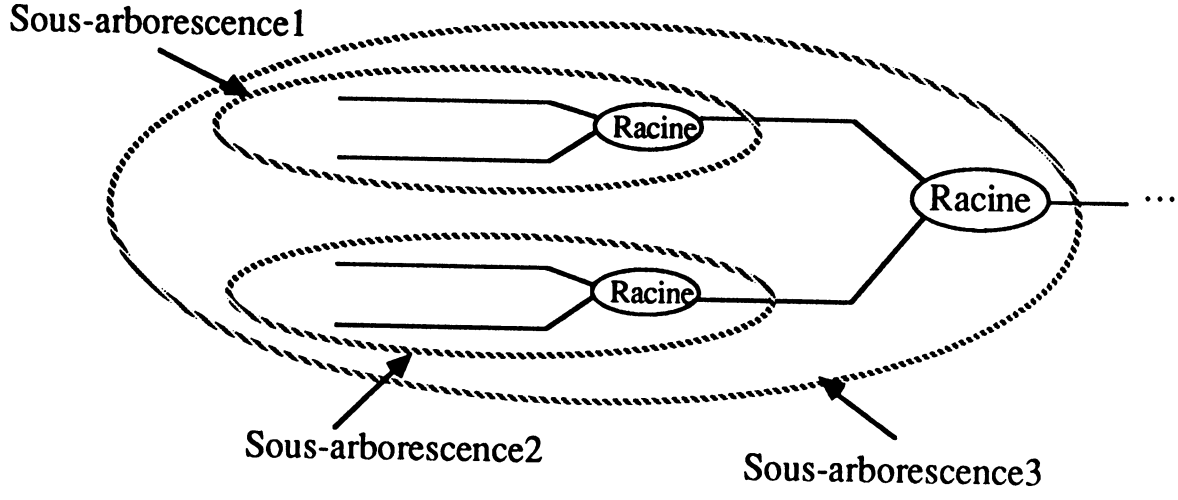


Figure 37. Traitement dichotomique des conflits

VII. 2. 3 Conditions sur les nœuds

La résolution d'un conflit nécessite le choix d'un chemin pour le retardement et le stockage. Certaines équipotentielles passent ainsi dans une barrière temporelle d'ordre supérieur. Ceci est effectué de façon à ne pas remettre en question la cohérence de l'arborescence.

Une équipotentielle évaluée par un chemin de l'arborescence et appartenant à une barrière temporelle d'ordre i peut passer dans une barrière temporelle d'ordre supérieur

si elle n'appartient qu'à cette seule barrière temporelle d'ordre i ou

si elle appartient à d'autres barrières temporelles d'ordre inférieur $j < i$.

Exemple 1

Considérons l'exemple du circuit de la figure 38.

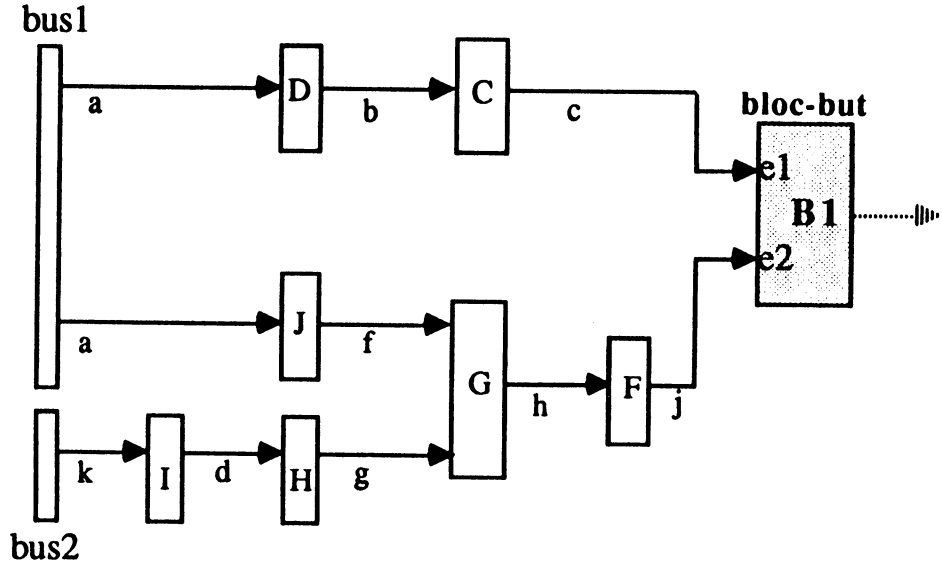


Figure 38. Exemple de circuit

L'arborescence de propagation du but consistant à propager en arrière les valeurs X et Y des entrées e1 et e2 du bloc B1 est illustrée sur la figure 39. L'arborescence manifeste un conflit à l'équipotentielle "a" du circuit à l'instant t-2.

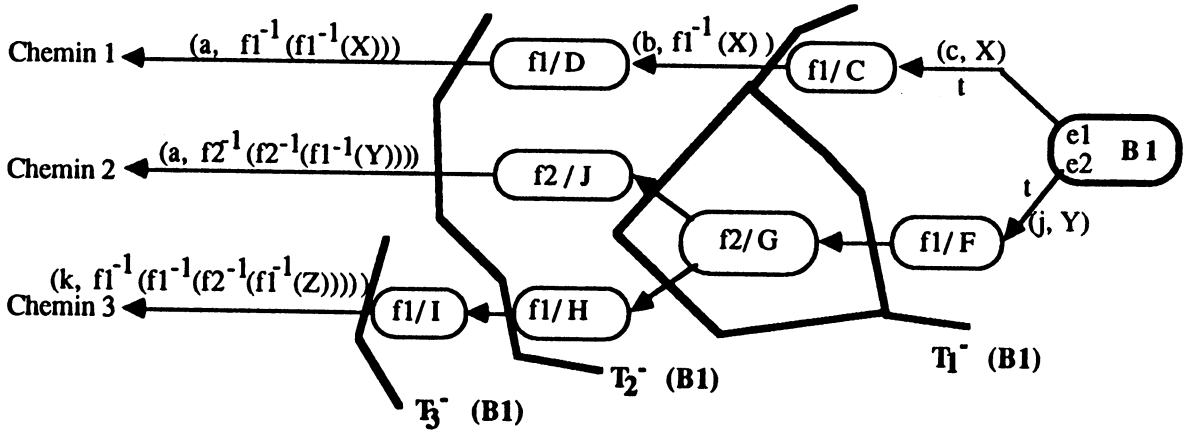


Figure 39. Arborescence de propagation

Plusieurs solutions sont proposées pour résoudre ce conflit. Ces solutions consistent à retarder l'instant d'application de la valeur de l'équipotentielle "a" évaluée par l'un des deux chemins 1 ou 2 et sauvegarder la valeur retardée dans un des éléments de mémorisation. Dans cet exemple le retardement d'une valeur d'une équipotentielle ne cause aucun problème puisqu'aucune équipotentielle n'appartient

à plus d'une barrière temporelle et par suite son passage d'une barrière temporelle à une autre d'ordre supérieur ne risque pas de générer un nouveau conflit.

Solution 1. retarder l'application de la valeur $f1^{-1}(f1^{-1}(X))$ affectée par le chemin 1 à l'équipotentielle "a" et la sauvegarder dans le bloc D. Ainsi l'équipotentielle "a" évaluée sur le chemin 1 passe dans la barrière temporelle T_3^-(B1) et l'équipotentielle "b" dans les barrières temporelles T_2^-(B1) et T_1^-(B1).

Solution 2. retarder l'application de la valeur $f1^{-1}(f1^{-1}(X))$ à l'équipotenielle "a" évaluée par le chemin 1 et la sauvegarder dans le bloc C. Ceci revient à passer l'équipotentielle "a" évaluée par le chemin 1 dans la barrière temporelle T_3^-(B1), l'équipotentielle "b" dans la barrière temporelle T_2^-(B1) et l'équipotentielle "c" dans la barrière temporelle T_1^-(B1).

Solution 3. retarder l'application de la valeur $f2^{-1}(f2^{-1}(f1^{-1}(Y)))$ à l'équipotentielle "a" évaluée par le chemin 2 et la sauvegarder dans le bloc J. Ceci revient à passer l'équipotentielle "a" évaluée par le chemin 2 dans la barrière temporelle T_3^-(B1) et l'équipotentielle "f" dans les barrières temporelles T_2^-(B1) et T_1^-(B1).

Solution 4. retarder l'application de la valeur $f2^{-1}(f2^{-1}(f1^{-1}(Y)))$ à l'équipotentielle "a" évaluée par le chemin 2 et retarder l'application de la valeur $f1^{-1}(f1^{-1}(f2^{-1}(f1^{-1}(Z))))$ à l'équipotentielle "k" évaluée par le chemin 3 et sauvegarder la valeur $(f1^{-1}(Y))$ dans le bloc F. Ceci revient à passer l'équipotentielle "a" évaluée par le chemin 2 dans la barrière temporelle T_3^-(B1), l'équipotentielle "f" dans la barrière temporelle T_2^-(B1), l'équipotentielle "h" dans la barrière temporelle T_2^-(B1), l'équipotentielle "j" dans la barrière temporelle T_1^-(B1), l'équipotentielle "k" dans la barrière temporelle T_4^-(B1), l'équipotentielle "d" dans la barrière temporelle T_3^-(B1), et l'équipotentielle "g" dans la barrière temporelle T_2^-(B1).

Exemple 2. Considérons l'exemple du circuit de la figure 38 où les 2 bus sont les mêmes (bus1=bus2). L'équipotentielle "k" devient "a". L'arborescence de propagation du sous-but consistant à propager en arrière les valeurs X et Y des entrées e1 et e2 du bloc B1 est illustrée sur la figure 40.

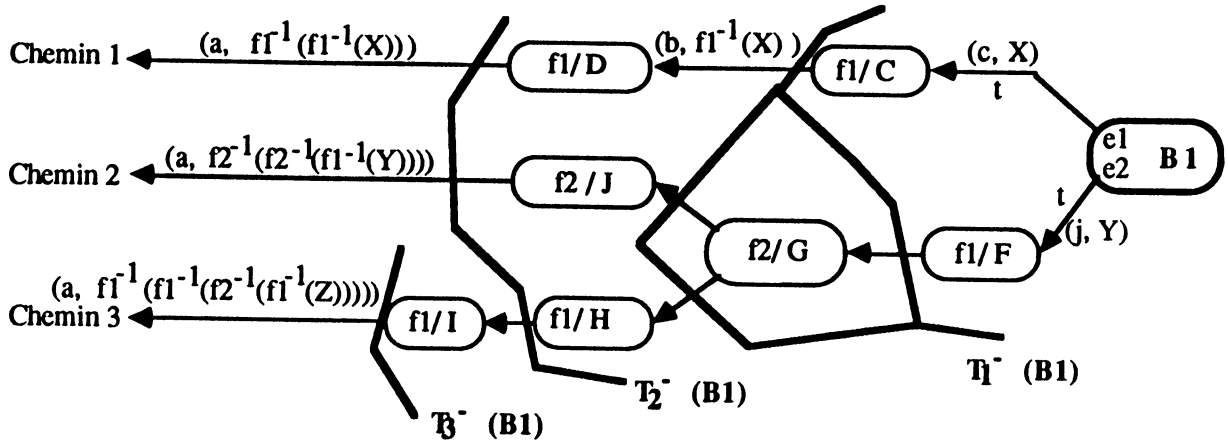


Figure 40. Arborescence où une équipotentielle "a" appartient aux trois chemins

Le chemin 1 et le chemin 2 manifestent un conflit à l'équipotentielle "a" du circuit à l'instant symbolique t-2. Puisque les chemins 2 et 3 évaluent l'équipotentielle "a", ils sont considérés solidaires c.à.d. si le chemin 2 doit être retardé, le chemin 3 doit l'être aussi. Ils forment une sous-arborescence.
Le retardement de la valeur de "a" ne peut pas être effectué sur le chemin 1 car l'équipotentielle "a" évaluée par le chemin 1 appartient à la barrière temporelle T2-(B1) alors que l'équipotentielle "a" évaluée sur le chemin 3 appartient à une barrière temporelle d'ordre supérieure T3-(B1). C'est la sous-arborescence qui est choisie pour le stockage. Les solutions proposées pour résoudre le conflit sont les suivantes:

Solution 1. retarder l'application de la valeur $f2^{-1}(f2^{-1}(f1^{-1}(Y)))$ à l'équipotentielle "a" sur le chemin 2 et la sauvegarder dans le bloc J et retarder l'application de la valeur $f1^{-1}(f1^{-1}(f2^{-1}(f1^{-1}(Z))))$ à l'équipotentielle "a" sur le chemin 3 et la sauvegarder dans le bloc I ou H.

Solution 2. retarder l'application de la valeur $f2^{-1}(f2^{-1}(f1^{-1}(Y)))$ à l'équipotentielle "a" sur le chemin 2 et retarder l'application de la valeur $f1^{-1}(f1^{-1}(f2^{-1}(f1^{-1}(Z))))$ à l'équipotentielle "a" sur le chemin 3 et sauvegarder la valeur $(f1^{-1}(Y))$ de l'équipotentielle "h" dans le bloc F.

VIII Insertion d'un ensemble minimisé de points de test

La recherche d'un chemin symbolique associé à un but peut échouer dans le cas où une entrée d'un bloc du chemin n'est pas contrôlable ou une sortie n'est pas observable. Ceci peut être dû à la présence des fonctions opaques dans un bloc ou à des conflits rencontrés qui ne peuvent pas être résolus. Un ensemble de points de test est alors proposé afin d'assurer la réussite de la recherche de chacun des chemins symboliques. Une seconde étape minimise cet ensemble.

VIII. 1 Points de test associé à l'échec d'un but

Etant donné un vecteur symbolique but Vi à propager, l'échec de la propagation d'une entrée ou d'une sortie de ce vecteur peut nécessiter l'insertion d'un point de test ou d'un ensemble de points de test. A chaque échec on associe une somme de monômes booléens où un monôme est la concaténation des nœuds à rendre contrôlable ou observable et remédier par suite à cet échec.

Dans l'exemple de la figure 41a, la propagation arrière de l'entrée-but e1 rencontre une fonction opaque et par suite le nœud b est non controlable. Un seul point d'insertion peut être mis au nœud b ou a afin de rendre la propagation possible. Ceci s'exprime comme suit: $p(\text{échec}_i) = a + b$.

Dans l'exemple de la figure 41b, la propagation arrière de l'entrée-but e1 rencontre des fonctions opaques et par suite les nœuds b et h sont non contrôlables. Deux points d'insertion sont alors proposés aux nœuds b et h ou un seul point au nœud a suffit. L'ensemble des points de test relatif à l'échec de la propagation de l'entrée-but e1 du bloc-but initial s'exprime comme suit: $p(\text{échec}_i) = b\ h + a$.

Dans l'exemple de la figure 41c, la propagation arrière de l'entrée-but e1 produit un conflit et par suite le nœud j est non contrôlable. Un point d'insertion $p(\text{échec}_i)$ est alors proposé en un nœud d'un des deux chemins. Ceci s'exprime comme suit: $p(\text{échec}_i) = (j + h + a) + (i + b + a)$.

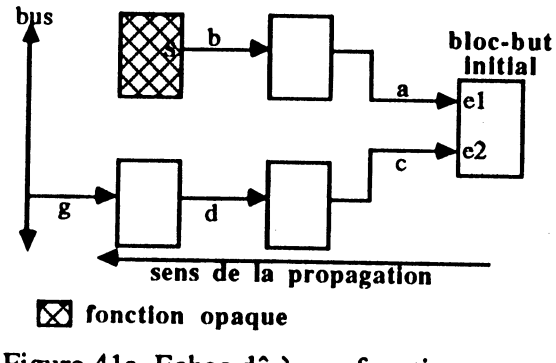


Figure 41a. Echec dû à une fonction opaque

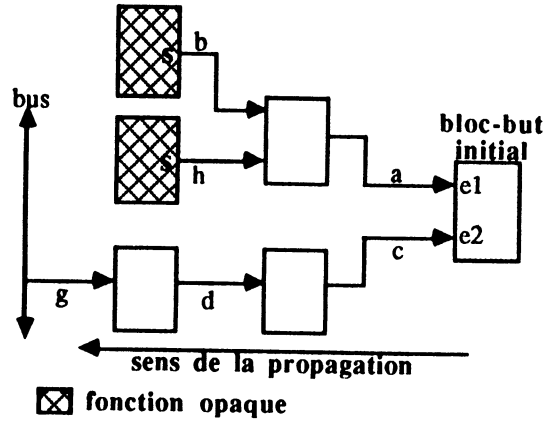


Figure 41b. Echec dû à deux fonctions opaques

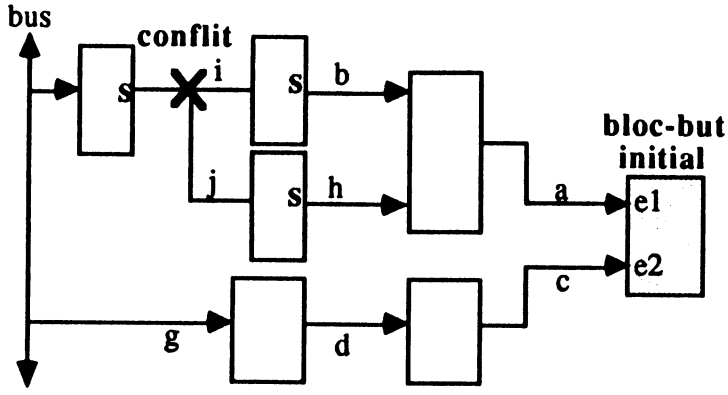


Figure 41c. Echec dû à un conflit de deux chemins

VIII. 2 Points de test associés à l'ensemble des échecs

L'ensemble des points d'insertion PI nécessaires pour propager tous les vecteurs-but de tous les blocs du circuit est le produit des ensembles des points d'insertion associés à l'échec des vecteurs-but. Soit PI = $\prod$ p(échec$_i$). Cette expression est minimisée afin d'avoir un nombre global de points de test qui soit optimal. Les points d'insertion recherchés sont ceux qui constituent le monôme de cardinalité minimale dans l'expression de PI. C'est un problème de minimisation booléenne [Pet56] abondamment traité dans la littérature.

Dans notre approche on adopte un algorithme itératif de recherche du monôme minimal. A chaque itération il compare entre deux monômes et choisit le plus petit en nombre de variables.

Soit un circuit dans lequel deux propagations associées aux vecteurs V1 et V2 (d'un même bloc ou de deux blocs différents) ont échoué. Soit p(échec1) et p(échec2) les ensembles de points de test associés respectivement à l'échec des vecteurs V1 et V2 tel que p(échec1) = a b + c et p(échec2) = a + c

L'ensemble global des points de test du circuit est alors donné par

PI = p(échec1) . p(échec2) = (a b + c) . (a + c)

PI = (a b + a b c + c a + c)

L'ensemble global et optimal des points d'insertion est donné par

min (PI) = c

IX Génération des vecteurs de test réels

Les vecteurs réels de test du circuit sont déduits à partir des vecteurs symboliques locaux et globaux et des valeurs réelles des vecteurs locaux des blocs.
Les vecteurs symboliques globaux sont donnés sous la forme d'une expression fonction des vecteurs symboliques locaux. Pour chacun des vecteurs réels associé à un vecteur symbolique local, les vecteurs réels globaux sont calculés en remplaçant les symboles des expressions symboliques par leur valeur réelle.
Les différentes séquences obtenues peuvent être compactées mais ceci ne sera pas exposé ici.

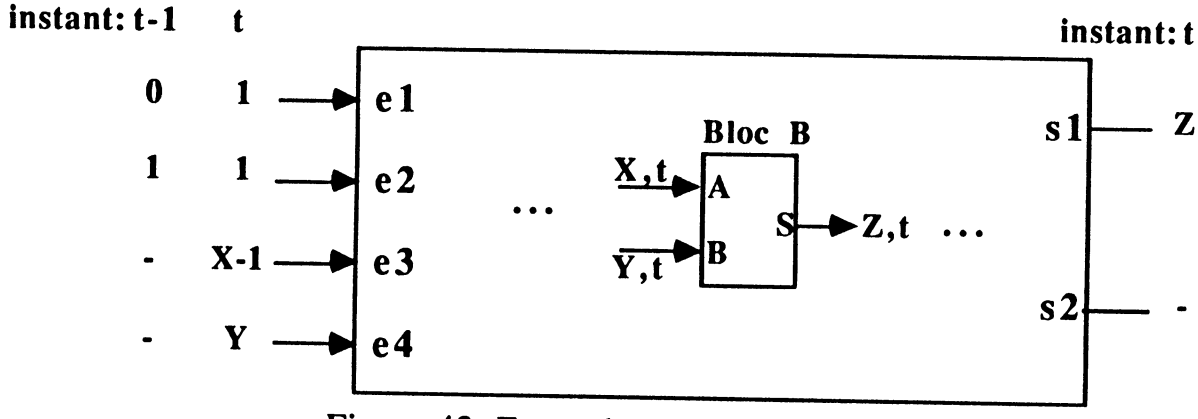


Figure 42. Exemple d'un circuit

Soit l'exemple de la figure 42 et soit un vecteur symbolique local de test du bloc B défini par Vs = [[(X, t), (Y, t)], -, -, [(Z, t)], -]. La séquence des vecteurs symboliques globaux de test associée au vecteur Vs est calculée par les processus de propagation avant et arrière et donnée dans le tableau suivant.

Vecteurs symboliques locaux de test				Vecteurs symboliques globaux de test						
A	B	S	instant	e1	e2	e3	e4	s1	s2	instant
X	Y	Z	t	0	1	-	-	-	-	t-1
				1	1	X-1	Y	Z	-	t

Les vecteurs réels globaux de test associés aux vecteurs réels vecteur1 et vecteur2 sont donnés dans le tableau suivant. Ces vecteurs sont obtenus en remplaçant les valeurs symboliques X, Y et Z dans les expressions symboliques de e3, e4 et s1 par leur valeur 4, 2 et 0 respectivement dans la première séquence et 2, 5 et 1 respectivement dans la seconde.

Vecteurs réels locaux de test					Vecteurs réels globaux de test						
	A	B	S	instant	e1	e2	e3	e4	s1	s2	instant
Vecteur1:	4	2	0	t	0	1	-	-	-	-	t-1
					1	1	3	2	0	-	t
Vecteur2:	2	5	1	t	0	1	-	-	-	-	t-1
					1	1	1	5	1	-	t

...

X Implantation et résultats expérimentaux

Un premier prototype qui calcule les chemins symboliques d'un circuit et propose un ensemble minimal de points d'insertion a été réalisé en langage Prolog II. Il comporte près de 1100 lignes de codes. Ce logiciel prend en entrée une description du circuit en termes de blocs interconnectés. Actuellement, il existe une première version de bibliothèque qui contient la description des blocs qu'on trouve couramment dans un circuit du type chemin de données. Il s'agit des multiplexeurs à deux et quatres entrées, additionneurs à deux entrées sans retenue, soustracteurs, multiplieurs, registres à signal de chargement, portes trois états, bus, ROM. Cette librairie nous a permis de faire tourner le programme sur des exemples de circuits du type chemins de données, afin de valider notre démarche, montrer ses avantages et prévoir des améliorations futures.

Notons qu'aucune comparaison n'a pu être faite avec les résultats des autres méthodes existantes pour la génération hiérarchisée de test. Ceci est dû à l'absence de circuits de référence internationale (benchmarks) pour cette approche.

Notre algorithme a été appliqué sur 4 circuits du type chemins de données. Les résultats sont donnés dans la table 1. Chaque circuit est défini par le nombre de ses blocs, portes équivalentes et vecteurs locaux symboliques et réels. Le nombre des vecteurs de test globaux calculés par le logiciel de recherche de chemins symboliques est fourni dans table 1 ainsi que le nombre de vecteurs réels globaux qui en résulteP. Cette table indique aussi le nombre des points de test proposé dans le cas où l'algorithme de recherche de chemins symboliques échoue. C'est le cas du circuit Filter 1 qui comporte un bloc opaque du type ROM que l'on ne peut pas traverser. Pour ce circuit une proposition d'un point de test a été émise afin de "contourner" la ROM. Après l'insertion du point de test proposé le circuit a une couverture totale.

Les résultats fournis par l'algorithme révèle les avantages dûs à l'utilisation des valeurs symboliques au lieu des valeurs réelles et du processus de retardement.

chemin des données	nb. des blocs	nb. des portes équivalentes	vect. locaux symboliques	vect. globaux symboliques	vect. locaux réels	vect. globaux réels	Points de test ?
Facet1	20	932	30	97	2661	9197	aucun
Facet3	16	988	22	77	1601	5087	aucun
Filter1	52	1752	69	394	470	1394	1
Diff1	28	1056	40	167	1737	7438	aucun

Table 1. Résultats de l'algorithme de calcul des chemins symboliques sur des circuits du type chemins de données de référence internationale

X. 1 Avantages de l'utilisation des valeurs symboliques

L'utilisation des valeurs symboliques dans les propagations réduit considérablement le nombre des propagations, des échecs et par suite le nombre de retours arrières nécessaires. Ceci a pour effet de diminuer le temps de la génération de test. Dans la table 1 le nombre des propagations effectuées par l'algorithme est égal au nombre des vecteurs symboliques locaux aux blocs du circuit. Pour l'exemple du circuit Facet 1, l'algorithme effectue 30 propagations symboliques au lieu de 2661 propagations de vecteurs réels.

La table 2 montre l'avantage de l'utilisation des valeurs symboliques dans la réduction du nombre des échecs des propagations et de retours arrières alors effectués. Dans le circuit Facet 1, la table 2 montre que le processus de la propagation symbolique manifeste 24 échecs au lieu de 2350 pour la propagation de vecteurs réels et 456 retours arrières au lieu de 42511 respectivement.

	propagation réelle		propagation symbolique		Gain (%)	
chemin de données	# retour ar.	# échecs	# retour ar.	# échecs	# retour ar.	# échecs
Facet1	42511	2350	456	24	99%	99%
Facet3	2152	1427	32	20	99%	99%
Filter1	12909	140	3095	34	76%	76%
Diff1	32090	441	361	8	99%	98%

Table 2. Gain dû à l'utilisation des valeurs symboliques

X. 2 Avantages de l'utilisation du processus de retardement

La table 3 montre que l'utilisation du processus de retardement diminue le nombre des échecs, de retours arrières et par suite le temps de la génération de test ainsi que le nombre des points de test à insérer. Dans l'exemple du circuit Facet 1, les 24 échecs sont résolus par l'algorithme de retardement et le nombre de retours arrières passe de 456 à 24.

On remarque que pour le circuit Filter 1, l'application de l'algorithme de retardement ne réussit pas à éliminer les 6 échecs restants. Ceci n'est pas dû à une mauvaise performance de l'algorithme mais, comme on l'a déjà dit, à la présence d'un bloc opaque de ROM non traversable dans le circuit.

	sans retardement		avec retardement		Gain (%)	
chemins de données	# retour ar.	# échecs	# retour ar.	# échecs	# retour ar.	# échecs
Facet1	456	24	24	0	95%	100%
Facet3	32	20	25	0	22%	100%
Filter1	3095	34	569	6	82%	82%
Diff1	361	8	8	0	98%	100%

Table 3. Gain dû au processus de retardement

On fait remarquer que dans table 2 et table 3 le nombre des échecs correspond au nombre des vecteurs symboliques dont la propagation a échoué.

Conclusion

Cette thèse a comme objectif d'apporter une contribution à la génération de test pour circuits logiques de très haute complexité. L'idée de base que seule une meilleure maîtrise des modèles fonctionnels permet d'avancer l'état de l'art dans ce domaine.

Dans la première partie de cette thèse, on considère l'une des applications les plus évidentes, à savoir la génération de test pour machines d'états finis. Dans ce cas, l'utilisation du modèle utilisé pour la synthèse s'impose et il s'agit là d'une approche qui a fait l'objet de nombreuses études et transferts industriels. Sur le plan théorique le point abordé dans cette thèse est le point délicat de la qualité d'une séquence de test en terme de couverture de fautes. Il est clair que toute séquence "même aléatoire" détecte un certain nombre de fautes. Il s'agit donc de démontrer que le pouvoir de détection ou qualité d'une séquence fonctionnelle est supérieure à celle d'une séquence aléatoire et de façon plus pointue supérieure à celle des autres approches du même type proposées dans la littérature. Il semble que nous ayons pu démonter que la séquence fonctionnelle bien ciblée consistant à parcourir tous les arcs d'un graphe d'états, telle qu'elle a été proposée ici, ait pu atteindre un bon taux de couverture et que les raisonnements par identification ou ceux fondés sur des hypothèses d'erreur ne sont pas vraiment justifiés. Les résultats obtenus très encourageants ont été améliorées en utilisant les résultats de simulation de la première séquence (parcours de graphe). En effet les fautes déjà détectées sur les variables d'états sont propagées vers les sorties. D'autre part il a été montré que les fautes résiduelles non détectées peuvent faire l'objet à bon escient d'une approche déterministe. L'originalité de celle-ci est d'utiliser à nouveau le modèle fonctionnel en travaillant par distinction sur les machines justes et fausses. Ceci donne aussi une approche unifiée pour l'ensemble de l'étude et évite complètement l'utilisation du D-algorithme.

La dernière partie de cette thèse a abordé le problème sans doute le plus complexe dans le domaine de la génération de test à savoir la génération hiérarchisée de test. Il semble que ce soit la seule solution dans le futur pour aborder la grande complexité. C'est également une approche qui fait une bonne distinction entre la génération de vecteurs de test au niveau de chaque bloc et la stratégie globale de test. En effet cette dernière doit être définie dès les premières phases de la conception du circuit et elle ne dépend pas des vecteurs précis générés ou utilisés au niveau des blocs. Cette séparation ou "hiérarchisation" va dans le sens d'une approche bien intégrée dans le processus de conception. L'approche présentée dans cette thèse est réaliste car elle fait un grand usage des modes transparents ou quasi-transparents de chaque bloc et se restreint dans un premier temps, au moins pour les

exemples pratiques traités, aux réseaux de blocs à signaux de contrôle accessibles donc de type "chemin de données". Parmi les points originaux de cette thèse, il semble que l'utilisation de valeurs symboliques soit une très bonne proposition même si tous les problèmes théoriques qui en résultent ne sont pas résolus. En effet reconnaitre que deux expressions symboliques sont identiques n'est toujours pas trivial et bien que dans les cas pratiques traités ici, le problème ne s'est pas posé, il peut cependant apparaître pour des circuits plus compliqués. Il faudrait donc s'y pencher. D'autre part, un traitement adéquat des barrières temporisées et l'utilisation de propagations retardées a permis de résoudre un plus grand nombre de conflits. Remarquons, enfin, que l'approche est réellement considérée au niveau RTL et que les propagations avant et arrière des valeurs de test se font réellement sur des modèles purement fonctionnels. La modélisation proposée est prototypée en Prolog. Elle est souple et concise.

Il reste bien entendu de très nombreux points à étudier. S'il a été montré que la modélisation fonctionnelle peut être extraite de la description en VHDL [Cra89a], [Jay90a], mais aucune implantation réelle n'a pu être sérieusement tentée dans un environnement industriel de CAO. D'autre part, comme il a été dit précédemment, le traitement de valeurs symboliques n'a pas été complètement mené à terme. Cette approche doit également être expérimentée sur des circuits où les signaux de contrôle ne sont pas accessibles. De même l'insertion de points de test doit être plus largement traitée et connectée automatiquement à des systèmes d'insertion de multiplexeurs ou de points mémoire montés en scan. Nous pensons donc avoir laissé la porte ouverte à de nombreux travaux ultérieurs de recherche dans ce domaine.

Bibliographie

[Aba85] M. Abadir, M. Breuer
"A knowledge-based system for designing testable VLSI chips", IEEE Design & Test, pp. 56-68, August 1985.

[Abr85] M. Abramovici, J. Kulikowski, P. Menon, D. Miller
"Test generation in LAMP2: concepts ans algorithms", International Test Conference, pp. 49-56, 1985.

[Agr76] P. Agrawal, V. Agrawal
"On Monte Carlo testing of logic tree networks", IEEE Transactions on Computers, Vol. C-25, pp. 664-667, June 1976.

[Agr81] V. Agrawal
"An information theoretic approach to digital fault testing", IEEE Transactions on Computers, Vol. C-30, N° 8, pp. 582-587, August 1981.

[Agr84] V. Agrawal, S. Jain, D. Singer
"Automation in design for testability", Custom Integrated Circuits Conference, May 1984.

[Agr85] V. Agrawal, S. Seth, C. Chuang
"Probabilistically guided test generation", International Symposium on Circuits and Systems", Japan, pp. 687-690, 1985.

[Agr87] V. D. Agrawal, K.-T. Cheng
"Threshold-Value Simulation and Test Generation", Testing and Diagnosis of VLSI and ULSI, Proceedings NATO Adv. Study Institute, Italy, 1987.

[Agr88a] V. Agrawal, S. Seth
"Test generation for VLSI Chips", IEEE Catalog Number EH0278-2, pp. 67- 77, 1988.

[Agr88b] V. D. Agrawal, K.-T. Cheng, P. Agrawal
"Contest: a concurrent test generator for sequential circuits", Design Automation Conference, pp. 84-89, 1988.

[Alf87] G. Alfs, Gerold
"Design and implementation of a heuristic search algorithm for the KARATE system", Diploma thesis, Kaiserslautern University, December 1987.

[Alf88] G. Alfs, R. Hartenstein, A. Wodtko
"The KARL/ KARATE system - Automatic test pattern generation based on RT level descriptions", International Test Conference, pp. 230-235, 1988.

[Ani89] P. Anirudhan, P. Menon

"Symbolic test generation for hierarchically modeled digital systems", International Test Conference, pp. 461-469, 1989.

[Ash90] P. Ashar, A. Ghosh, S. Devadas, A. R. Newton
"Implicit State Transitions Graphs. Applications to Sequential Logic Synthesis and Testing", International Conference on Computer-Aided Design, pp. 84-87, 1990.

[Bel80] C. Bellon, G. Saucier
"Etude de la contamination par des erreurs externes dans un système distribué", R.A.I.R.O. Automatique/ Systems Analysis and Control, vol.14, N° 4, pp. 375-392, 1980.

[Ben84] R. Bennetts
"Design of testable logic circuits", Reading, MA: Addison-Wesley, 1984.

[Bha85] D. Bhattacharya, J. Hayes
"High-level test generation using bus faults", 15th Symposium on Fault-Tolerant Computing, pp. 65-70, 1985.

[Bou71] W. Bouricius and al.
"Algorithms for detection of faults in logic circuits", IEEE transactions on Computers, Vol. C-20, N° 11, pp. 1258-1264, November 1971.

[Bre86] M. Breuer, A. Friedman
"Diagnosis and reliable design of digital systems", Computer Science Press, 1986.

[Brg84] F. Brglez, P. Pownall, P. Hum
"Application of testability analysis: from ATPG to critical path tracing", IEEE International Test Conference, pp. 705-712, 1984.

[Cal89] J. Calhoun, F. Brglez
"A framework and method for hierarchical test generation", International Test Conference, pp. 480-490, 1989.

[Cha70] H. Chang, E. Manning, G. Metze
"Fault diagnosis of digital systems", John Wiley & sons, 1970.

[Cha87] S. Chandra, J. Patel
"An hierarchical approach to test vector generation", DAC, pp. 495-501, 1987.

[Cha89] S. Chandra, J. Patel
"Experimental evaluation of testability measures for test generation", IEEE Transactions on Computer-Aided Design, Vol. 8, N° 1, pp. 93-97, January 1989.

[Che87] K. Cheng, V. Agrawal

"A simulation-based directed search method for test generation", ICCD, pp. 48-51, October 1987.

[Che88a] W. Cheng
"SPLIT circuit model for test generation", Design Automation Conference, pp. 96-101, 1988.

[Che88b] W. Cheng
"The BACK algorithm for sequential test generation", International Conference on Computer Design: VLSI in Computers & Processors, pp. 66-69, 1988.

[Che90] K.-T. Cheng, J. Y. Jou
"Functional Test Generation for Finite State Machines", International Test Conference, pp. 162-168, 1990.

[Cho91] H. Cho, G. D. Hachtel, F. Somenzi
"Redundancy Identification and Removal Based on BDD's and Implicit State Enumeration", MCNC, 1991.

[Cra87] M. Crastes de Paulet, M. Karam, G. Saucier
"Système à base de règles pour la génération assistée de programmes de test de cartes", L'Onde Electrique, Vol. 68, N° 6, pp. 67-74, décembre 1987.

[Cra88a] M. Crastes de Paulet, M. Karam, G. Saucier
"Rule-based test planning", IFIP workshop on knowledge-based systems for test and diagnosis, pp. 141-157, September 1988.

[Cra88b] M. Crastes de Paulet, M. Karam, G. Saucier
"Génération de programme de validation d'ASIC complexes", 6ème colloque international de la fiabilité et la maintenabilité, pp. 177-182, Octobre 1988.

[Cra89a] M. Crastes de Paulet, M. Karam, G. Saucier
"Test expertise from high-level specifications", IFIP Working Group 10.2 Conference on CAD systems using AI techniques, pp. 77-84, Japan, June 1989.

[Cra89b] M. Crastes de Paulet, M. Karam, G. Saucier
"Test expertise for ASICs", IFIP International Conference on Very Large Scale Integration (VLSI'89), pp. 153-162, RFA, August 1989.

[Cra89c] M. Crastes de Paulet, M. Karam, G. Saucier
"Testability expertise and test planning from high-level specifications", International Test Conference (ITC'89), pp. 692-699,USA, August 1989.

[Cra90] M. Crastes de Paulet, M. Karam, G. Saucier

"Intelligent design modifications for testability improvment of ASICs", 7th international conference on reliability and maintainability, pp. 489-495, Brest, France, June 1990.

[Dav76] R. David, G. Blanchet
"About random fault detection of combinational networks", IEEE Transactions on Computers, pp. 659-664, June 1976.

[Dev87] S. Devadas, H. Ma, A. Sangiovanni-Vincentelli
"Logic verification, testing, and their relationship to logic synthesis", Proceedings of the Nato Advanced Study Institute on Testing and Diagnosis of VLSI and ULSI, Como, Italy, 1987.

[Eic77] E. Eichelberger, T. Williams
"A logic design structure for LSI testing", 14th Design Automation Conference, pp. 462-468, June 1977.

[Eic78] E. Eichelberger, T. Williams
"A logic design structure for LSI testability", Journal Design Automation Fault-Tolerant Computing, Vol. 2, N° 2, pp. 165-178, May 1978.

[Fre88] S. Freeman
"Test generation for data-path logic: the F-path method", IEEE Journal of Solid-State Circuits, Vol. 32, N° 2, pp. 421-427, april 1988.

[Fri73] A. Friedman, P. Menon
"Restricted checking sequences for sequential machines", IEEE Transactions on Computers, Vol. C-22, N° 4, April 1973.

[Fuj83] H. Fujiwara, T. Shimono
"On the acceleration of test generation algorithms", IEEE Transactions on Computers, Vol. C-32, pp. 1137-1144, December 1983.

[Gho89] A. Ghosh, S. Devadas, et A. R. Newton
"Test generation for highly sequential circuits", International Conference on Computer-Aided Design, pp. 362-365, Novembre 1989.

[Gib85] A. Gibbons
"Algorithmic graph theory", Cambridge university press 1985.

[Gil61] A. Gill
"State identification experiments in finite automata", Information and Control, Vol.4, pp. 132-154, 1961.

[Gin58] S. Guinsburg
"On the length of the smallest uniform experiment which distinguishes the terminal states of a machine", Journal Association Computing Machinery, Vol. 5, pp. 266-280, July 1958.

[Goe80] P. Goel
"Test generation costs analysis and projections", 17th Design Automation Conference, pp. 77-84, June 1980.

[Goe81] P. Goel
"An implicit enumeration algorithm to generate tests for combinational logic circuits", IEEE Transactions on Computers, Vol. C-30, N° 3, pp. 215-222, March 1981.

[Gol79] L. Goldstein
"Controllability/ observability analysis for digital circuits", IEEE Transactions on Circuits and systems, Vol. CAS-26, pp. 685-693, September 1979.

[Gol80] L. Goldstein, E. Thigpen
"SCOAP: Sandia controllability/ observability analysis program", 17th Design Automation Conference , pp. 190-196, June 1980.

[Gon85] M. Gondran, M. Minoux
"Graphes et algorithmes", Eyrolles, 1985.

[Gup90] Rajesh Gupta, Rajiv Gupta, Melvin Breuer
"The ballast methodology for structured partial scan design", IEEE Transactions on Computers, Vol. C-39, N° 4, pp. 538-544, April 1990.

[Hay73] J. Hayes, A. Friedman
"Test point placement to simplify test detection", 3rd Symposium on Fault-Tolerant Computing, pp. 73-78, June 1973.

[Hay74] J. Hayes
"On modifying logic networks to improve their diagnosability",IEEE Transactions on Computers, Vol. C-23, pp. 56-62, January 1974.

[Hen64] F. Hennie
"Fault detecting experiments for sequential circuits", 5th Annual Symposium on Switching Circuit Theory and Logical Design, Princeton, pp. 95-110, November 1964.

[Hsi71] E. Hsieh
"Checking experiments for sequential machines", IEEE Transactions on Computers, Vol. C-20, pp. 1152-1166, October 1971.

[Iba75] O. Ibarra, S. Sahni
"Polynomially complete fault detection problems", IEEE Transactions on Computers, Vol. C-24, pp. 242-249, March 1975.

[Jay89] C. Jay, "Experience in functional-level test generation and fault coverage in a silicon compiler", Defect and Fault Tolerance in VLSI, USA, Florida, 1989.

[Jay90a] C. Jay, M. Crastes de Paulet, M. Karam, G. Saucier
"Hierarchical test generation from VHDL", Euro ASIC'90, France, May 1990.

[Jay90b] C. Jay, M. Karam, D. Salber, G. Saucier
"Test of finite state machines with enhanced coverage", IEEE workshop on defect and fault tolerance in VLSI systems, pp. 305a-305m, France, November 1990.

[Kar91a] M. Karam, G. Saucier, C. Jay
"Test generation of controllers using the synthesis specifications", Euro ASIC'91, Paris, France, May 1991.

[Kar91b] M. Karam, R. Leveugle, G. Saucier
"Hierarchical test generation based on delay propagation", International Test Conference (ITC'91), USA, October 1991.

[Kar91c] M. Karam, R. Leveugle, G. Saucier
"Hierarchical testability analysis and test generation using functional modelling: a solution to test design bottleneck", First International workshop on the Economics of Design and Test, USA, September 1991.

[Kar91d] M. Karam, G. Saucier
"Test generation of finite state machines from the synthesis specifications", IFIP International Workshop on the relationship between Synthesis, Test, and Verification, Berkeley, USA, November 1991.

[Koh78] Z. Kohavi
"Switching and finite automata theory", Mc Graw-Hill, 1978.

[Kri87] B. Krishnamurthy
"Hierarchical test generation: can AI help?", International Test Conference, pp. 694-700, 1987.

[Krü91] G. Krüger
"A tool for hierarchical test generation", IEEE Transactions on Computer - Aided Design, Vol. 10, N° 4, pp. 519-524, april 1991.

[Kun90] R. Kunda, P. Narain, J. Abraham, B. Rathi
"Speed up of test generation using high-level primitives", DAC, pp. 594-599, 1990.

[Lar89] T. Larrabee

"A framework for evaluating test pattern generation strategies", International Conference on Computer Design, pp. 44-47, 1989.

[Lee91] J. Lee, J. H. Patel
"An architectural level test generator for a hierarchical design environment", International Symposium Fault-Tolerant Computing, pp. 44-51, 1991.

[Lee89] J. Leenstra, L. Spaanenburg
"Using hierarchy in macro cell test assembly", Europeen Test Conference, pp. 63-70, 1989.

[Lee90] J. Leenstra, L. Spaanenburg
"Hierarchical test assembly for macro based VLSI design", International Test Conference, pp. 520-529, 1990.

[Lis87] R. Lisanke, F. Brglez, A. Geus, D. Gregory
"Testability-driven random pattern generation", IEEE Transactions on Computer Aided design, Vol. CAD-6, pp. 1082-1087, November 1987.

[Ma87] H. K. T. Ma, S. Devadas, A. R. Newton, et A. Sangiovanni-Vincentelli
"Test Generation for Sequential Finite State Machines", International Conference on Computer Aided Design, pp. 288-291, 1987.

[Ma88] H. K. T. Ma, S. Devadas, A. R. Newton, et A. Sangiovanni-Vincentelli
"Test generation for sequential circuits", IEEE Transactions on Computer-Aided Design, pp. 1081-1093, October 1988.

[Mal85] S. Mallela, S. Wu
"A sequential circuit test generation system", International Test Conference, pp. 57-61, 1985.

[Mar78] R. Marlett
"EBT, a comprehensive test generation technique for highly sequential circuits", 15th Design Automation Conference, pp. 332-339, June 1978.

[Mar86] R. Marlett
"An effective test generation system for sequential circuits", 23rd Design Automation Conference, pp. 250-256, June 1986.

[Moo56] E. Moore
"Gedanken-experiments on sequential machines", Princeton University, Annals of Mathematics Studies, N° 34, pp. 129-153, 1956.

[Mor89] S. Morley, R. Marlett, J. Deer
"Automating test generation of complex circuits without the burden of full scan", Euro-Asic, pp. 503-508, 1989

[Mic83] A. Miczo

"The sequential ATPG: a theoretical limit", International Test Conference, pp. 143-147, October 1983.

[Mur88] B. Murray, J. Hayes
"Hierarchical test generation using precomputed test for modules", International Test Conference, pp. 221-229, 1988.

[Mut76] P. Muth
"A nine-valued circuit model for test generation", IEEE Transactions on Computers, Vol. C-25, pp. 630-636, June 1976.

[Nit85] S. Nitta, M. Hirabayashi
"Test generation by activation and defect-drive (TEGAD)", Integration, the VLSI Journal, Vol. 3 , N° 1, March 1985.

[Pat86] S. Patil, J. Patel
"Effectiveness of heuristic mesures for automatic test pattern generation", 23rd Design Automation Conference, pp. 547-552, June 1986.

[Pat90] S. Patil, P. Banerjee
"A parallel branch and bound algorithm for test generation", IEEE Transactions on Computer Aided Design, Vol. 9, N° 3, pp. 313-322, March 1990.

[Pat91] J. H. Patel, T. Niermann
"HITEC: A Test Generation Package for Sequential Circuits", European Conference on Design Automation, 1991.

[Par75] K. Parker, E. McCluskey
"Analysis of logic circuits with faults using input signal probabilities", IEEE Transactions on Computers, pp. 573-578, May 1975.

[Par76] K. Parker
"Adaptive random test generation", Journal Design Automation and Fault-Tolerant Computing, Vol. 1, pp. 62-83, October 1976.

[Pet56] S. Petrick
"A direct determination of the irredundant forms of a boolean function from a set of prime implicants", A. F. Cambridge Research Center Report AFCRC-TR-56-110, Bedford, Mass. 1956.

[Poa63] J. Poage
"The derivation of optimum tests for logic circuits", Ph. D thesis, Princeton University, 1963.

[Pom91] I. Pomeranz, S. Reddy

"On achieving a complete fault coverage for sequential machines using the transition fault model", International Automation Conference, pp. 341-346, June 1991.

[Put71] G. Putzolu, J. Roth
"A heuristic algorithm for testing of asynchronous circuits", IEEE Transactions on Computers, Vol. C-20, pp. 639-647, June 1971.

[Rot66] J. P. Roth
"Diagnosis of automata failures: a calculus and a method", IBM Journal of Research and Development, pp. 278-291, July 1966.

[Roy90] K. Roy, J. Abraham
"High level test generation using data flow descriptions", EDAC, pp. 480-484, 1990.

[Sam89] A. Samad, M. Bell
"Automating ASIC design for testability - the VLSI test assistant", International Test Conference, pp. 819-828, 1989.

[Set86] S. Seth, B. Bhattacharya, V. Agrawal
"An exact analysis for efficient computation of random-pattern testability in combinational circuits", 16th Symposium on Fault-Tolerant Computing, pp. 318-323, 1986.

[Sar89] T. Sarfert, R. Markgraf, E. Trischler, M. Schulz
"Hierarchical test pattern generation based on high-level primives", International Test Conference, pp. 470-479, 1989.

[Sau72] G. Saucier
"Recherche d'une sequence de test d'une machine séquentielle", R.A.I.R.O., N° J-1, pp. 99-105, 1972.

[Sav84] J. Savir, G. Ditlow, P. Bardell
"Random pattern testability", IEEE Transactions on Computers, Vol. C-33, N° 3, pp. 79-90, January 1984.

[Sch75a] H. Schnurmann, E. Lindbloom, R. Carpenter
"The weighted random test-pattern generator", IEEE Transactions on Computers, Vol C-24, pp. 695-700, July 1975.

[Sch75b] D. Schuler, E. Ulrich, T. Barker, S. Bryant
"Random test generation using concurrent logic simulation", 12nd Design Automation Conference, pp. 261-267, 1975.

[Sch87] M. Schulz, E. Trischler, T. Sarfert
"SOCRATES: a highly efficient automatic test pattern generation system", International Test Conference, pp. 1016-1026, September 1987.

[Sch88] M. Schulz, E. Auth
"Advanced automatic pattern generation ans redundancy identification techniques", 18th Symposium on Fault-Tolerant Computing, pp. 30-35, June 1988.

[She77] J. Shedletsky
"Random testing : practicality vs. effectiveness", Symposium on Fault-Tolerant Computing, pp. 175-179, June 1977.

[Su89] C. Su
"Computer-Aided design of pseudoexhaustive test for data paths", Ph.D. Thesis, Dept. of Electronical and Computer Engineering, Univ. of Winsconsin - Madison, december 1989.

[Su90] C. Su, C. Kime
"Multiple path sensitization for hierarchical circuit testing", International Test Conference, pp. 152-161, 1990.

[Tom83] S. Tomas, J. Shen
"A survey of: Functional level testing and testability measures", Res. Rep. CMUCAD-83-18, November 1983.

[Wil77] T. Williams, E. Eichelberger
"Random patterns within a structured sequential logic design", Semi-Conductor Test Symposium, pp. 19-27, October 1977.

[Wil82] T. Williams, K. Parker
"Design for testability - a survey", IEEE Transactions on Computers, Vol. C-31, N° 1, pp. 1-15, January 1982.

Annexe 1: Recherche d'un chemin Eulérien dans un graphe Eulérien

a - Recherche d'une arborescence T du graphe G (N, A) ayant le nœud N0 comme racine. Aux arcs (u, v) du graphe G est affecté un nombre booléen T(u, v) tel que: si (u, v) est un arc de T alors T(u, v) = vrai sinon T(u, v) = faux.

b - **Pour** chaque nœud v de l'ensemble N du graphe **faire**

A_V <-- ø

Bv <-- ø

I(v) <-- 0

fait

c - **Pour** chaque arc (u, v) de l'ensemble A **faire**

Si T(u, v) = vrai **alors**

ajouter u à la queue de la liste A_V

ajouter (u, v) à la queue de la liste Bv

Sinon

ajouter u à la tête de la liste A_V

ajouter (u, v) à la tête de la liste Bv

fin si

fait

d - **Si** existe vj / D(vj)>0 **alors**

CE <-- [ ] , NC <-- vj

Sinon CE <-- [ ] , NC <-- N0

fin si

e - **Tant que** I(NC) < d-(NC) **faire**

f - I(NC) <-- I(NC) + 1

g - NC <-- $A_{NC(I(NC))}$

AT <-- $B_{NC(I(NC))}$

h - ajouter AT à la tête de CE

fin Tant que

i - Sortie : CE

Pas a

Dans ce premier pas on cherche une arborescence arborescence T ayant N0 comme racine et passant par tous les nœuds du graphe. Cette recherche se fait par un algorithme de recherche en profondeur d'abord. On affecte aux arcs (u, v) du graphe G un nombre booléen T(u, v) = vrai si (u, v) appartient à l'arborescence T sinon T(u, v) = faux.

Pas b

A chacun des nœuds v du graphe on affecte un nombre I(v) qui définit le nombre d'arcs entrants au nœud v qui ont été traversés et les listes A_V et B_V respectivement des nœuds u et des arcs (u, v) précecesseurs de v dans G.
Dans la procédure actuelle I(v) est initialisé à 0, A_V et B_V sont des listes vides.

Pas c

Le pas c est destiné à la construction des listes A_V et B_V relatives aux nœuds v du graphe. A_V et B_V contiennent respectivement les nœuds et les arcs prédecesseurs de v. Dans ces listes l'arc (ui, v) qui appartient à l'arborescence T et le nœud ui sont classés les derniers dans les listes B_V et A_V.

Pas d

Ce pas est le point de départ de l'algorithme de recherche du chemin Eulérien inverse. Ce chemin sera donné sous forme de liste d'arcs qu'on construit pas à pas dans les procédures qui suivent, en commençant par le nœud extrémité finale jusqu'à atteindre le nœud extrémité initiale. Dans le cas où la recherche vise un chemin Eulérien, CE est initialisée à [ ] et on définit le nœud vj comme étant le nœud courant NC. Le nœud vj est le nœud de degré positif D(vj)>0 dans G. Dans le cas de la recherche d'un cycle Eulérien, CE est initialisée à [ ] et on définit le nœud N0 comme étant le nœud courant NC.

Pas e

I(NC) inférieur au nombre des arcs entrants de NC traduit le fait qu'il existe des arcs prédecesseurs à NC qui n'ont pas encore été traversés.

Pas f, g, h

Un arc AT prédecesseur au NC est alors choisi pour être traversé. Le nœud prédecesseur de NC est un nœud de la liste A_{NC}. C'est l'élément de A_{NC} d'ordre I(NC) noté $A_{NC}(I(NC))$. L'arc prédecesseur de NC est l'élément de B_{NC} d'ordre I(NC) noté $B_{NC}(I(NC))$. I(NC) est incrémenté pour traduire le fait qu'un arc prédecesseur va être traversé (pas f). Le pas h ajoute l'arc au chemin. Ceci se fait en déclarant le nœud u comme nouveau nœud courant (pas g) et en ajoutant l'arc prédecesseur AT à la tête de la liste CE (pas h).

Annexe 2 : Recherche d'une arborescence complète d'un graphe - parcours par profondeur

Notre algorithme de recherche d'une arborescence dans un graphe procède par une recherche en profondeur. C'est un algorithme récursif. A chaque pas on part d'un nœud courant NC déjà visité et on cherche à traverser un arc successeur (NC, v) qui n'a pas encore été visité. Si un tel arc existe alors le nœud v sera le nœud suivant à visiter sinon l'algorithme est appliqué au nœud visité juste avant NC. L'arborescence recherché a une racine N0 et il est donné sous la forme d'un ensemble d'arcs.

a - Pour tout nœud u du graphe chercher A(u)
b - Pour tout nœud u du graphe faire O(u) <-- 0
c - i <-- 1
d - T <-- ø
e - Nœud courant NC <-- ni
f - Appel Procédure RA(NC)
g - Sortie: T

h - **Procédure** RA(NC)
Début
i - O(NC) <-- i
j - i <-- i+1
k - **Pour** tout v appartenant à A(NC) **faire**
l - **Si** O(v) = 0 **alors**
m - T <-- T U {(NC, v)}
n - NC <-- v
o - RA(NC)
fin si
fait
fin RA(NC)

Pas a
Pour chacun des nœuds u du graphe on calcule A(u) ensemble des nœuds adjacents à v tels que (u, v) est un arc direct du graphe. En d'autres termes A(u) est l'ensemble des nœuds extrémité des arcs sortants de u.

Pas b

A chaque nœud u du graphe est affecté un nombre O(u) qui donne l'ordre dans lequel le nœud u a été visité dans l'arborescence. O(u) = 0 exprime le fait que u n'a pas encore été visité. A ce niveau de l'algorithme O(u) est initialisé à 0 et ceci pour tout nœud u du graphe.

Pas c, d, e

L'algorithme de recherche d'une arborescence sera appelé après avoir précisé le numéro de l'itération i initialisé à 1 (pas c), mis à jour l'arbre T initialisé à l'ensemble ø (pas d) et défini le nœud courant initialisé à ni racine de l'arbre (pas e).

Pas f, g

La procédure de la recherche de l'arbre (RA) est appelée avec le nœud courant ni racine de l'arbre. L'arbre complet T sera le résultat de cette procédure (pas g).

Pas h

La procédure de la recherche d'arbre est exécutée à partir d'un nœud courant NC.

Pas i, j

NC est un nœud qui vient d'être visité. Son ordre O(NC) est égale au numéro de l'itération précédente (pas i). Ainsi la racine ni a un ordre égale à 1 puisqu'elle est le premier nœud visité. La nouvelle itération commence en incrémentant le numéro d'itération (pas j).

Pas k

Les nœuds extrémité des arcs sortant du nœud courant NC sont donnés par A(NC). Ces nœuds sont considérés un par un. Pour chacun d'eux les procédures des pas suivants l, m, n, o sont appliquées.

Pas l

Si un nœud v de A(NC) a un ordre O(v) nul cela veut dire qu'il n'a pas encore été visité. v sera alors le nouveau nœud à visiter. Sinon l'algorithme reviendra au pas k pour considéré un autre nœud de A(NC).

Pas m, n, o

On est dans le cas où on a trouvé un nœud non visité v extrémité d'un arc sortant de NC soit (NC, v). L'arc (NC, v) est alors ajouté à l'arbre (pas m). Le nœud v est

défini comme étant le nouveau nœud courant (pas n) pour lequel la procédure de recherche d'arbre est de nouveau appelée (pas o).

Annexe 3 : Transformation d'un graphe en un graphe Eulérien

L'algorithme qui rend Eulérien un graphe G (non Eulérien) est donné comme suit. Il consiste à chercher un flot maximal Fmax dans G.

a - Ajouter au graphe G, une source S et d'un puits P. Soit G' le graphe obtenu. Dans G', S est un nœud prédecesseur aux nœuds de degré > 0 et P est un nœud successeur aux nœuds de degré < 0.
b - Calculer les listes A(v) successeurs des nœuds v du graphe G'. Calculer les capacités et les flots sur les arcs du graphe. La capacité d'un arc (S, ni) successeur à S est égale au degré de son nœud successeur D(ni). La capacité d'un arc (nj, P) prédecesseur à P est égale au degré de son nœud prédecesseur D(nj). Les autres arcs d graphe G' ont une capacité infinie. La valeur du flot des arcs de G' est initialisé à 0.
c - Path <-- true, F <-- 0, Fmax <-- somme des degrés des arcs successeurs à S
d - **Tant que** Path = true et F < Fmax **faire**
    début
e - Construire les listes B(v) des arcs successeurs à v et augmentant le flot en spécifiant s'ils sont des arcs directs ou inverses dans G'. Calculer leur capacité résiduelle.
f - Calculer la plus petite chaîne non saturée (augmentant le flot) de S à P.
g - **si** Path = true **alors**
        début
            calculer la capacité résiduelle de la chaîne trouvée:
            Δ = min Δ(ni, nj) où (ni, nj) sont les arcs de la chaîne.
h - **Pour tout** (ni, nj) de la chaîne **faire**
i - **si** (ni, nj) est direct dans la chaîne **alors**
j - f(ni, nj) <-- f(ni, nj) + Δ
k - **sinon** f(nj, ni) <-- f(nj, ni) - Δ
l - F <-- F + Δ
        fin
    fin
m - **si** path = true **alors**
    début
n - effacer les arcs sucesseurs à S et prédecesseurs à P
o - **Pour tout** ai tel que f(ai) ≠ 0 **faire**
p - ajouter un nombre d'arcs égal à f(ai)
    fin

Annexe 4 : Recherche de la plus petite chaîne augmentante de S à P

Il est donné par l'algorithme suivant. La plus chaîne augmentante de S à P est notée Chaîne.

```
a - Calculer la longueur L de la plus petite chaîne: recherche par niveau (annexe 5)
b -   si L ≤ 2 alors Path <-- false   /** message d'erreur sur le path **/
      sinon
          début
c -           pour tout nœud n du graphe faire construire B'(n).
              /** B'(n) est la liste des arcs successeur ou prédecesseurs à n et
                    augmentant le flot */
d -           Chaîne <-- [ ]
e -           NC <-- P
f -           tant que NC ≠ S faire
              début
g -               rechercher l'arc (ni, NC) appartenant à B'(NC) et tel que
                  L(ni) = L(NC) - 1
h -               rajouter (ni, NC) à la tête de Chaîne
i -               NC <-- ni
              fin
          fin
```

Annexe 5 : Recherche de la longueur de la plus petite chaîne de S à P

La longueur recherchée est notée L. La longueur de la plus petite chaîne allant de S à un nœud n est notée L(n). A(n) est la liste des nœuds entrant ou sortant de n et augmentant le flot.

```
a - pour tout nœud n faire L(n) <-- 0
b - i <-- 1, L(S) <-- 1
c - Liste <-- [ S ]
d - tant que Liste ≠ [ ] faire
      début
e -       Soit w <-- queue de Liste
f -       si L(w) ≠ i alors i <-- i + 1
g -       continue <-- true
h -       pour tout n appartenant à A(w) faire
i -       si continue = true alors
                début
j -                 si n = P alors
                          début
k -                           L(P) <-- i + 1
l -                           Liste = [ ]
m -                           continue <-- false
                          fin
n -                 si L(n) = 0 alors
                          début
o -                           L(n) <-- i + 1
p -                           ajouter n à la tête de Liste
                          fin
                fin
          Liste <-- Liste - w
      fin
q - L <-- L(P) - 1
```

Annexe 6 : Ajout d'arcs d'initialisation èn vue de rendre fortement connexe un graphe

a- recherche des composantes fortement connexes
b- construction du graphe réduit où les nœuds sont les composantes fortement connexes
c- les arcs d'initialisation sont rajoutés entre un état u d'une composante fortement connexe Ci n'ayant pas d'arcs sortants $d^+(Ci) = 0$ et l'état d'initialisation du graphe. L'état u est choisi tel que son degré est positif $D(u) > 0$.

L'algorithme de recherche des composantes fortement connexes est un simple parcours par profondeur d'abord du graphe il est donné comme suit. Dans cet algorithme DFI(n) est un indice qui rappelle l'ordre de la visite de n dans l'arbre. Q(n) est un indice qui sert à définir une composante fortement connexe ; Tous les nœuds n de la composante ont le même indice Q(n). L'indice j sert à numéroter les composantes fortement connexes.

a - **pour tout** nœud n **faire** DFI(n) <-- 0, visité (n) <-- false **fait**
b - i <-- 1, Liste = []<-- true, j=0
c - appel RCFC (N0)

Procédure RCFC (w)
d - DFI(w) <-- i
e - Q(w) <-- DFI(w)
f - ajouter w à la tête de Liste, visité (w) <-- true
g - i <-- i + 1
h - **pour tout** n appartenant à A(w) **faire**
i - **si** DFI(n) = 0 **alors**
début
j - RCFC (n)
l - Q(w) <-- min (Q(w), Q(n))
fin
sinon
o - **si** DFI(n) < DFI(w) **et** visité (n) **alors**
p - Q(w) <-- min (Q(w), DFI(n))
fin
fin
q - **si** Q(w) = DFI (w) **alors**
début
r - j = j+1

```
                Composante(j) = liste des nœuds de Liste depuis la tête jusqu'au
                nœud w,
s -             Pour tout nœud m de Composante (j) faire
                    début
                        retrancher m de Liste,
                        visité (m) <-- false
                    fin
            fin
        fin
```

Annexe 7 : Modélisation d'une fonction de base sur des exemples

Modélisation de la fonction de base "chargement registre" avec horloge implicite

```
prop_direct (Reg, Charg,([(D, X)], [(Ch, V1)], -, [(S, W) ], - ),t) :-
mettre_valeur ( [ ((V1, 1) , t) ] ),
égale ( [ ((W, X), t+1) ] ).

prop_inverse (Reg, Charg,([(D, X)], [(Ch, V1)], -, [(S, W)], - ),t) :-
mettre_valeur ( [ ((V1, 1) , t-1) ] ),
égale ( [ ((X, W), t-1) ] ).
```

Modélisation de la fonction de base "chargement registre" avec horloge explicite

```
prop_direct (Reg, Charg,([(D, X)], [(Ch, V1), (Horl, V2)], -, [(S, W) ], - ),t) :-
mettre_valeur ( [ ((V2, 0) , t-1), ((V1, 1), (V2, 1) , t) ] ),
égale ( [ ((W, X), t+1) ] ).

prop_inverse (Reg, Charg,([(D, X)], [(Ch, V1), (Horl, V2)], -, [(S, W)], - ),t) :-
mettre_valeur ( [ ((V2, 0) , t-2), ((V1, 1), (V2, 1) , t-1) ] ),
égale ( [ ((X, W), t-1) ] ).
```

Modélisation de la fonction de base "lire un mot memoire" avec horloge implicite

```
prop_direct (RAM, lecture,( -, [(A, V2), (R/Wb, V1)], [(Var, M)], [(S, W)], -), t) :-
mettre_valeur ( [ ((V1, 1) , (V2, V2), t) ] ),
égale ( [ ((W, M), t+1) ] ).

prop_inverse (RAM, lecture,(-,[(A, V2), (R/Wb, V1)], [(Var, M)], [(S, W)], -), t) :-
mettre_valeur ( [ ((V1, 1), (V2, V2), t-1) ] ),
égale ( [ ((M, W), t-1) ] ).
```

Annexe 8 : Extraction des vecteurs symboliques locaux à partir des fonctions de base modélisées en Prolog

Le vecteur ou la séquence symboliques associés à une fonction de base sont extraits à partir de la description de sa fonction directe. Les informations requises sont la liste des entrées et les sorties à affecter et les instants d'application et de validation. On rappelle le format de la clause de description d'une fonction directe en langage Prolog.

```
fonc_directe(type_bloc, nom_fonction, paramètres, instant1):-
        mettre_valeur ( [ (conditions, instant2) ] ),
        égale ( [ (effets, instant3) ] ).
```

I Extraction d'un vecteur symbolique associé à une fonction de base

L'extraction dun vecteur symbolique à partir de la fonction directe écrite en langage Prolog est très simple.
La liste des entrées et sortie que le vecteur symbolique affecte est déduite du terme "paramètres" de la tête de la clause de la fonction de base. L'instant d'application des entrées de donnée est "instant1" et l'instant d'application des entrées de contrôle est "instant2". Ces deux instants sont égaux puisqu'on est dans le cas où la fonction de base a besoin d'un seul vecteur d'entrée pour délivrer la sortie. L'instant de validation de la sortie est donné par "instant3".

Dans l'exemple du registre de la figure 20b du § III.1.1 dans le chapitre 3, le vecteur symbolique associé à la fonction "chargement" est déduit de la fonction directe modélisée par la clause suivante (annexe 6).

```
fonc_directe(Reg, Charg,([(D, X)], [(Ch, V1)], -, [(S, W) ], - ),t) :-
mettre_valeur ( [ ((V1, 1) , t) ] ),
égale ( [ ((W, X), t+1) ] ).
```

Un seul vecteur symbolique est associé à la fonction "chargement" puisque "instant1" = "instant2". Le vecteur symbolique recherché affecte les entrées et sorties de la liste suivante [[D], [Ch], -, [S], -]. Cette liste est déduite à partir du troisième terme de la tête de la clause. Les valeurs symboliques X1, X2 et Y sont affectées respectivement à la donnée D, au signal de chargement Ch et à la sortie S. L'instant d'application de X1 et X2 est donné par "instant1" = "instant2" = t. L'instant de validation de la sortie S est recherché dans le dernier littéral de la clause. Il est égal à t+1. On obtient le vecteur symbolique suivant.
Vs(Reg, Charg) = [[(X1, t)], [(X2, t)], -, [(Y, t+1)], -].

II Extraction d'une séquence symbolique associée à une fonction de base

Cette séquence est une suite de vecteurs symboliques où des entrées et/ ou sorties prennent des valeurs symboliques différentes à des instants symboliques différents.
Les informations à extraire à partir de la fonction directe relative à la fonction de base sont la liste et la valeur des entrées et sorties aux différents instants. L'extraction de ces informations à partir de la fonction directe écrite en langage Prolog est effectuée de la façon suivante.
Un vecteur symbolique est associé à chacun des instants "instant1" et "instant2" ≠ "instant1". La liste des entrées et des sorties qu'un vecteur symbolique affecte est déduite du terme "paramètres". A chacune des entrées et sorties de la fonction est affectée une valeur symbolique. Deux valeurs symboliques différentes sont affectées à une entrée ou une sortie si elle est évaluée deux fois dans la fonction et ces deux valeurs sont différentes. Dans un vecteur symbolique un " - " est mis à la place d'une valeur symbolique d'une entrée ou d'une sortie lorsque sa valeur est indifférente dans la fonction de base. Les instants symboliques d'application des entrées sont donnés par "instant1" et "instant2". Les instants symboliques de validation des sorties sont donnés par "instant3".

Dans l'exemple du registre de la figure 21 dans § III.2.1 du chapitre 3, la séquence symbolique associée à la fonction de base "chargement" est déduite à partir de la fonction directe modélisée en Prolog comme suit (annexe 6).

```
fonc_directe (Reg, Charg,([(D, X)], [(Ch, V1), (Horl, V2)], -, [(S, W) ], - ),t) :-
mettre_valeur ( [ ((V2, 0) , t-1), ((V1, 1), (V2, 1) , t) ] ),
égale ( [ ((W, X), t+1) ] ).
```

L'instant "instant1" est t et la liste des instants "instant2" est [t-1, t]. La séquence recherchée comprend donc deux vecteurs symboliques qui correspondent à l'instant t et t-1. La liste des entrées et sortie qu'un vecteur symbolique affecte est la suivante [[D], [Ch, Horl] , -, [S], -]. A l'instant t-1 seul le signal de l'horloge est appliqué. La valeur symbolique X3 lui est affectée. Le vecteur symbolique relatif à l'instant t-1 est le suivant.
[- , [- , (X3, t-1)], -, -, -]
A l'instant t la donnée D est appliquée ainsi que le signal de contrôle de chargement Ch. Les valeurs symboliques X1 et X2 lui sont affectées respectivement. A cet instant l'horloge est appliquée pour la deuxième fois. Une valeur X4 lui est affectée.

X4 ≠ X3 car X4 représente la valeur "1" alors que X3 représente la valeur "0" de l'horloge dans la fonction de base. Une valeur symbolique Y est affectée à la sortie qui est alors valide à t+1. Le deuxième vecteur symbolique de la séquence est exprimé comme suit.

[[(X1, t)], [(X2, t), (X4, t)], -, [(Y,t+1)], -]

III Extraction d'une séquence symbolique associée à un produit de fonctions de base

La séquence symbolique associée à un produit de fonctions de base est extraite à partir des fonctions directes du produit. C'est une séquence de vecteurs ou de séquences symboliques associés aux fonctions de base présentes dans le terme produit. L'ordonnencement de ces vecteurs ou séquences est effectué de manière à obtenir une séquence symbolique minimale. Ceci est illustré sur un exemple.

Dans l'exemple de la RAM de la figure 22 du § III.2.2 du chapitre 3, la séquence symbolique associée à la fonction produit "écriture*lecture" comporte deux vecteurs symboliques. Ils sont extraits à partir des fonctions directes suivantes.

fonc_directe(RAM, écriture,([(D, X)], [(A, V2), (R/Wb, V1)], -, -, [(Var, M)]), t) :-
mettre_valeur ([((V1, 0) , (V2, V2), t)]),
égale ([((M, X), t+1)]).

fonc_directe (RAM, lecture,(-, [(A, V2), (R/Wb, V1)], [(Var, M)], [(S, W)], -), t) :-
mettre_valeur ([((V1, 1) , (V2, V2), t)]),
égale ([((W, M), t+1)]).

Les vecteurs symboliques associés sont les suivants.

Vs(RAM, écriture) = [[(X1, ti)] , [(X2, ti), (X3, ti)], -, -, [Y1, ti+1]].
Vs(RAM, lecture) = [-, [(X4, tj), (X5, tj)], [(X6, tj)], [Y2, tj+1], -].

La séquence symbolique recherchée est formée de ces deux vecteurs dans lesquels les valeurs des variables de mémorisation sont supprimées et des relations sont établies entre les différentes valeurs symboliques et entre les différents instants symboliques.

A partir des fonctions de base, l'égalité X4=X2 et l'inégalité X3≠X6 sont déduites pour les valeurs symboliques de l'adresse et du signal de contrôle lecture/ écriture respectivement.

Le second vecteur est appliqué lorsque la sortie du premier vecteur est valide. Ainsi il peut être appliqué au même instant symbolique tj = ti+1 ou à un instant postérieur tj > ti+1. L'instant d'application du deuxième vecteur est choisi "au plus tôt". Soit tj=ti+1 ce qui guarantit une séquence symbolique minimale en termes de nombre d'instants symboliques qui lui sont nécessaires.
Après avoir effacé les valeurs et les instants symboliques associés aux variables de mémorisation interne, on obtient la séquence symbolique suivante.

Ss(RAM, écriture*lecture) = [
[[(X1, ti)] , [(X2, ti), (X3, ti)], - , -] ,
[[- , [(X2, ti+1), (X6, ti+1)], -, [(Y2,ti+2)], -]
]

Annexe 9 : Génération hiérarchisée de test du chemin de données "Filter1"

Dans cet annexe nous traitons un exemple de chemin de données suivant l'approche hiérarchisée exposée dans le chapitre 3 de cette thèse.
Soit le chemin de données "Filter1" illustré ci-dessous sur la figure 1. Ce circuit comporte 14 registres, 2 additionneurs, un multiplieur, une ROM, 26 portes trois états et 6 bus internes. Une seule horloge rythme le chargement des registres et le transfert à travers les portes trois états. Les commandes de chargement et de transfert sont effectuées aux fronts montants de l'horloge. Les instants symboliques sont définis aux fronts montants de l'horloge de base.

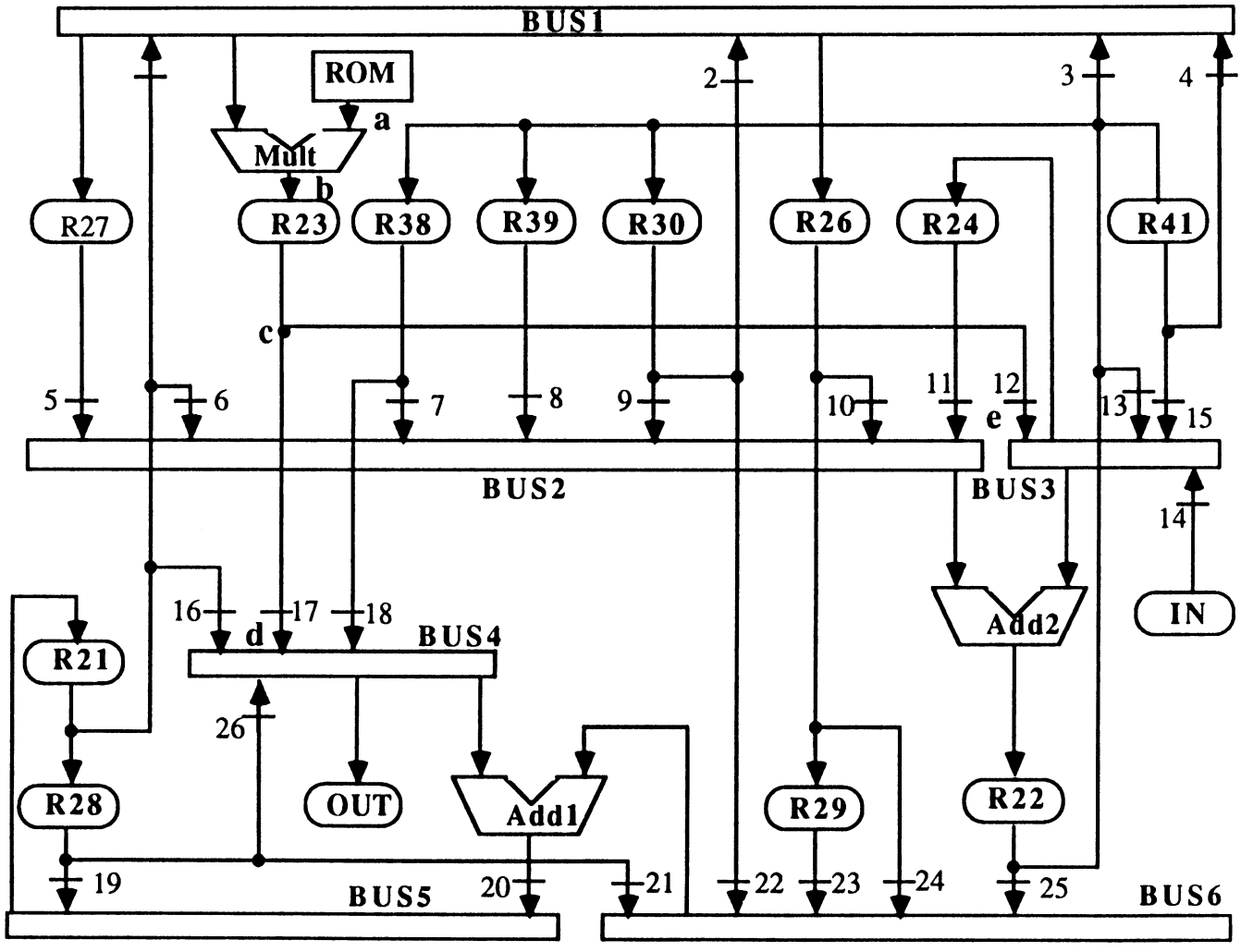


Figure 1. Le circuit "Filter1"

I Modélisation

La modélisation est effectuée avec une horloge implicite.
Le modèle de la fonction de chargement d'un registre est donné dans l'annexe 7.
Le modèle d'une fonction d'addition est donné dans le § II.4 du chapitre 3. Le modèle d'une fonction de multiplication est similaire à celui de la fonction d'addition.
La ROM peut effectuer une seule fonction qui est la lecture. La modélisation de la fonction de lecture d'un mot mémoire est donnée dans l'annexe 7.
Les fonctions de base d'un bus ou d'une porte trois états sont des fonctions de transfert où la sortie copie la valeur de l'entrée.

II Formatage des vecteurs symboliques locaux de test

Les vecteurs de test locaux à un registre, additionneur et multiplieur sont donnés dans le § III. 1 du chapitre 3. Les vecteurs de test associés aux fonctions d'un bus à deux entrées par exemple sont donnés par Vs(Bus, fonction, [[(X1, t), -], -, -, [(Y,t)], -]) et Vs(Bus, fonction, [[-, (X2, t)], -, -, [(Y,t)], -]) où X1, X2 et Y sont des valeurs symboliques associées à la première et deuxième entrée et à la sortie respectivement.
Le vecteur de test associé à une porte trois états est donné comme suit : Vs(Porte_trois_états, transfert, [[(X1, t)], [(X2, t)], -, [(Y, t+1)], -]) où X1, X2 et Y sont des valeurs symboliques associées à l'entrée, la commande de transfert et la sortie respectivement.

III Génération des vecteurs symboliques globaux de test

Dans une phase de prétraitement, la proximité des fonctions de base par rapport aux entrées primaires et aux sorties primaires du circuit est calculée. La figure 2 illustre la proximité par rapport aux entrées primaires et la figure 3 la proximité par rapport aux sorties primaires. Le contrôle de la fonction de base de R41, par exemple, peut être effectué à travers 8 fonctions de base des blocs du circuit (figure 2); l'observation de cette fonction est effectuée à travers 7 fonctions de base (figure 3).
A chacun des vecteurs symboliques locaux de test est associé un chemin symbolique. Le tableau suivant illustre le résultat de l'algorithme de recherche des chemins symboliques. Il montre les chemins qui ont réussi et ceux qui ont conduit à un échec avant et après la procédure de retardement. Le nombre de fois que la procédure de retardement a été appliquée est aussi mentioné.

Les résultats montrent que la recherche des chemins symboliques a échoué pour 6 vecteurs symboliques et que l'ensemble minimal des points de test se réduit au nœud "a" seconde entrée du multiplieur "Mult".

fonction de base	réussite/ échec avant retardemt	appel du processus de retardement	réussite / échec après retardemt	ensemble de points de test
addition / Add1	échec	1	réussite	
addition / Add2	réussite			
chargement/ In	réussite			
chargement/ R21	échec	1	réussite	
chargement/ R22	réussite			
chargement/ R23	échec	0	échec	a + b
chargement/ R24	réussite			
chargement/ R26	réussite			
chargement/ R27	réussite			
chargement/ R28	échec	1	réussite	
chargement/ R29	échec	2	réussite	
chargement/ R30	réussite			
chargement/ R38	réussite			
chargement/ R39	réussite			
chargement/ R41	réussite			
multipli./Mult	échec	0	échec	a
lecture / ROM	réussite			
transfert/ porte1	échec	1	réussite	
transfert/ porte2	réussite			
transfert/ porte3	réussite			
transfert/ porte4	réussite			
transfert/ porte5	réussite			
transfert/ porte6	échec	1	réussite	
transfert/ porte7	réussite			
transfert/ porte8	réussite			
transfert/ porte9	réussite			
transfert/ prt10	réussite			
transfert/ prt11	réussite			

fonction de base	réussite/ échec avant retardemt	appel du processus de retardement	réussite / échec après retardemt	ensemble de points de test
transfert/ prt12	échec	0	échec	a + b + c
transfert/ prt13	échec	2	réussite	
transfert/ prt14	réussite			
transfert/ prt15	réussite			
transfert/ prt16	échec	1	réussite	
transfert/ prt17	échec	0	échec	a + b + c
transfert/ prt18	réussite			
transfert/ prt19	échec	1	réussite	
transfert/ prt20	échec	1	réussite	
transfert/ prt21	échec	2	réussite	
transfert/ prt22	échec	2	réussite	
transfert/ prt23	échec	2	réussite	
transfert/ prt24	échec	2	réussite	
transfert/ prt25	échec	1	réussite	
transfert/ prt26	échec	1	réussite	
f1 / bus1	réussite			
f2 / bus1	réussite			
f3 / bus1	échec	1	réussite	
f4 / bus1	réussite			
f1 / bus2	réussite			
f2 / bus2	réussite			
f3 / bus2	réussite			
f4 / bus2	réussite			
f5 / bus2	réussite			
f6 / bus2	échec	1	réussite	
f7 / bus2	réussite			
f1 / bus3	réussite			
f2 / bus3	réussite			
f3 / bus3	échec	2	réussite	
f4 / bus3	échec	0	échec	a+b+c+e
f1 / bus4	échec	1	réussite	
f2 / bus4	échec	1	réussite	

fonction de base	réussite/ échec avant retardemt	appel du processus de retardement	réussite / échec après retardemt	ensemble de points de test
f3 / bus4	échec	0	échec	a+b+c+d
f4 / bus4	réussite			
f1 / bus5	échec	1	réussite	
f2 / bus5	échec	1	réussite	
f1 / bus6	échec	1	réussite	
f2 / bus6	échec	2	réussite	
f3 / bus6	échec	2	réussite	
f4/ bus6	échec	2	réussite	
f5/ bus6	échec	2	réussite	
Bilan	34 échecs / 35 réussites	39	6 échecs / 63 réussites	ensemble min.= {a}

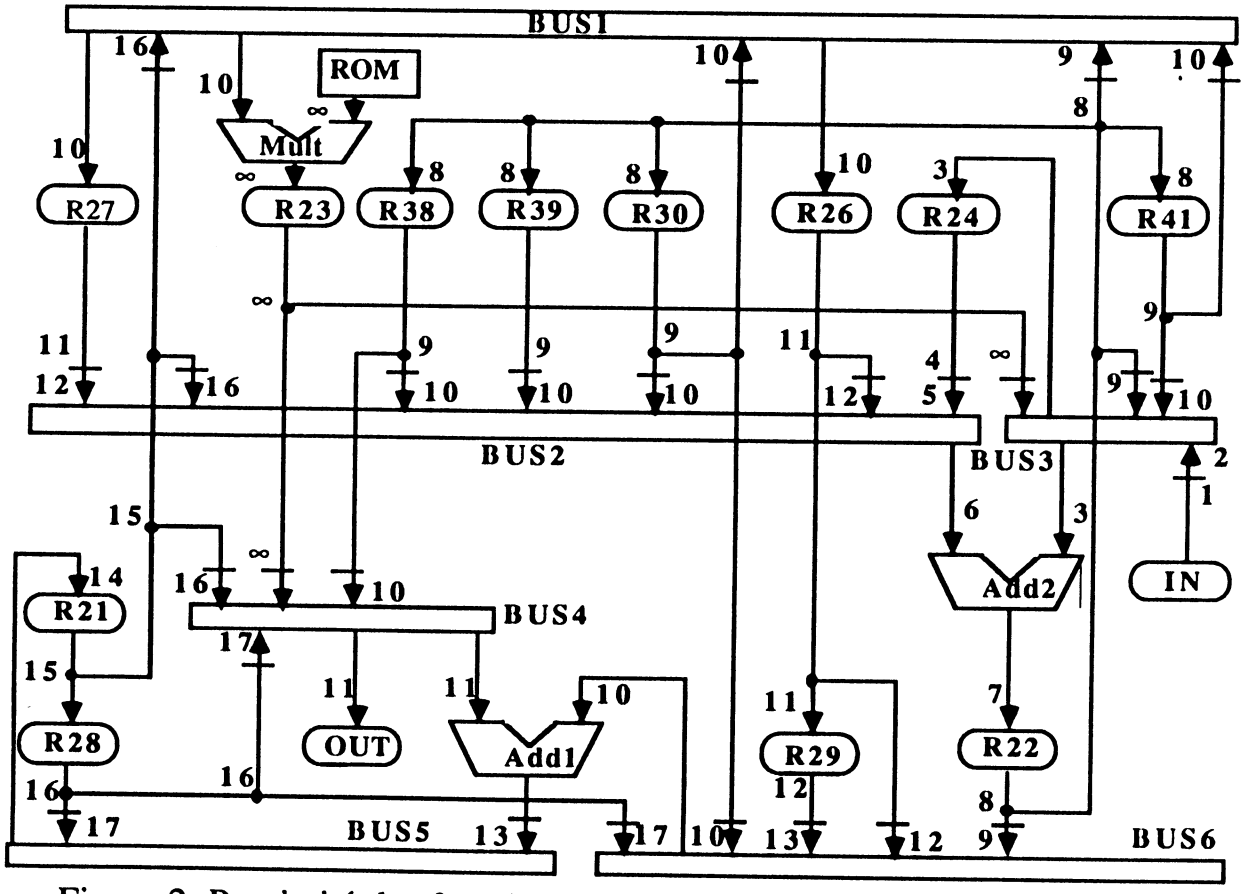


Figure 2. Proximité des fonctions de base par rapport aux entrées primaires

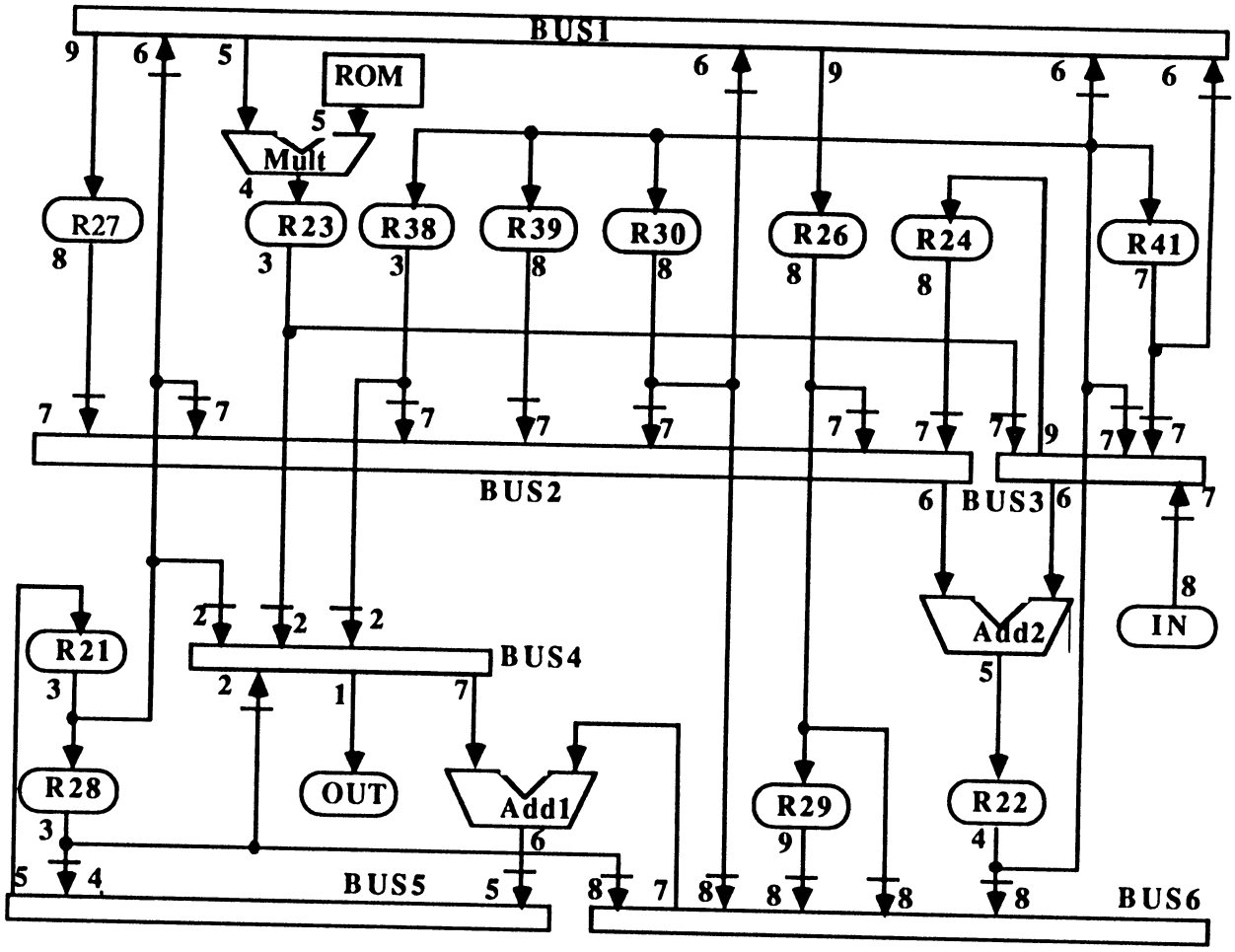


Figure 3. Proximité des fonctions de base par rapport aux sorties primaires

Table des matières

NSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

INPG

Grenoble, le 14 Octobre 1991

ÉPARTEMENT 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 Bernard COURTOIS Directeur de Recherche CNRS TIM3

Monsieur Christian LANDRAULT Directeur CNRS (LAM)

Mademoiselle KARAM Margot

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 : Signal-Image-Parole

Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
M. GARNIER