Institut National Polytechnique de Grenoble
ENSIMAG

Université Joseph Fourier
U.F.R.
Informatique & Mathématiques Appliquées

I. M. A. G.

DEA D'INFORMATIQUE

Projet présenté par :

Daniel SALBER

TEST DE MACHINES D'ETATS FINIS

Effectué au laboratoire LCSI

Date: 26 juin 1991

Jury:
M. ADIBA
C. BOITET
M. CRASTES DE PAULET
M. KARAM
B. PLATEAU

Remerciements

Je tiens à remercier ici Mme G. Saucier et toute l'équipe du LCSI, et plus particulièrement Margot Karam qui m'a encadré tout au long de ce projet. Je souhaite aussi exprimer ma gratitude aux chercheurs du laboratoire Artémis, qui m'ont fait profiter de leur savoir et de leur documentation sur les graphes.

Table des matières

Introduction

Le présent rapport présente le travail que j'ai réalisé lors de mon projet de DEA Informatique, au Laboratoire de Conception de Systèmes Intégrés (LCSI) de l'INPG.

Ce travail porte sur le test de machines d'états finis. Nous souhaitions développer une méthode de test à partir d'une spécification de haut niveau de la machine d'états finis, telle qu'elle est fournie aux outils de synthèse automatique de contrôleurs. Ce choix permet de ne pas tenir compte de la structure physique du circuit et d'éviter ainsi une explosion combinatoire dans le cas de grandes machines. La méthode de test devait également être indépendante de toute hypothèse à priori sur les fautes du circuit. Mais en général, le taux de couverture obtenu est inférieur à 100%; une phase complémentaire est donc nécessaire, à l'issue de laquelle le taux de couverture est de 100%.

La première partie présente le sujet sur lequel j'ai travaillé ainsi que l'état de l'art dans ce domaine. La seconde partie détaille le travail effectué. Enfin, la troisième partie expose les aspects théoriques de la phase complémentaire du projet.

PARTIE I

Le terme "test des circuits" regroupe trois notions différentes de test. En effet, les erreurs éventuelles de conception sont mises en évidence par des moyens sophistiqués et longs, grâce au test en "fin de conception" qui prouve la bonne fonctionnalité du circuit. Pour la fabrication, existe un test de "fin de fabrication" qui détecte les défauts et élimine les circuits défectueux. Le troisième test concerne l'usager, qui, à la réception du produit, peut tester la conformité de son circuit grâce au "test en réception", et par la suite effectuer des tests de maintenance.

Dans ce rapport, nous nous intéressons uniquement au test en fin de conception.

Pour les tests du prototype en fin de conception, il existe deux types de test : le test aléatoire et le test déterministe (avec vecteurs prédéterminés de test).

Le test aléatoire consiste à générer des vecteurs d'entrée aléatoires de test. 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. Quelles que soient les hypothèses de fautes, il subsiste une certaine probabilité de ne pas découvrir un circuit défectueux. Ce test est peu cher mais peut être très long.

Le test avec vecteurs prédéterminés; on distingue deux méthodes, par distinction ou par identification.

La méthode par identification est une recherche de la conformité du circuit par rapport à une référence, par application de vecteurs d'entrée prédéterminés. C'est le cas du test exhaustif des circuits combinatoires ou de la partie combinatoire d'un automate et le test des chemins d'un organigramme de contrôle d'un contrôleur par exemple. La longueur du test exhaustif est fonction du nombre n des entrées primaires du circuit et est égale à 2^n.

Dans la méthode par distinction, on fait des hypothèses de fautes 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 du test déterministe par distinction permet d'obtenir des tests compacts et performants que l'on peut, par un choix judicieux des vecteurs d'entrée, lier à la fonctionnalité du circuit.

Les circuits séquentiels 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 de temporisation. Le deuxième type implémente 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).

I. Etat de l'art

Ce chapitre présente l'état de l'art dans le domaine de la génération de test pour machines d'états finis.

La littérature sur les méthodes de génération de test des machines d'états finis laisse apparaître deux tendances classiques de test. La première consiste en une génération aléatoire de vecteurs de test. Dans la seconde les vecteurs générés sont déterministes, c'est-à-dire que chacun de ces vecteurs est calculé dans le but de détecter une faute ou un mauvais fonctionnement de la machine dû à une faute.

Qu'il s'agisse de la génération aléatoire ou déterministe, la séquence résultante est évaluée par un simulateur de fautes qui travaille au niveau structurel du circuit et sur des hypothèses de fautes logiques. Dans ce cas, le simulateur sert d'évaluateur de la séquence générée. Toutefois, il existe actuellement une tendance pour que le simulateur de fautes assiste à la génération de la séquence de test. Dans ce cas, les résultats de la simulation d'une séquence de test sont exploités pour générer une séquence de test complémentaire permettant de détecter et tester les fautes résiduelles.

I.1 Approche aléatoire

Cette approche consiste à générer une séquence aléatoire de test où les vecteurs ne sont pas choisis suivant des hypothèses de fautes. Cette approche est utilisée pour le test des circuits combinatoires et séquentiels [Bre71], [Sch75], [Nit85], [Jac89]. La longueur de la séquence de test est choisie à l'avance. Aucune information à priori n'est valable pour déterminer la longueur maximale de la séquence de test nécessaire pour avoir une bonne couverture.
Cette génération a l'avantage d'être rapide. Elle a l'inconvénient de ne pas garantir une bonne couverture. Elle nécessite ainsi une séquence complémentaire afin de se rapprocher du taux de couverture 100%. Cette deuxième séquence peut être aléatoire; la longueur du test peut donc devenir très grande. Elle peut être aussi générée par une approche déterministe pour tester les fautes résiduelles non couvertes. Ainsi la deuxième séquence de [Cho91] est constituée d'une partie déterministe pour la justification de l'état qui contrôle l'effet de la faute et une partie aléatoire pour la propagation de cet effet jusqu'aux sorties primaires.

I.2 Approche déterministe

Dans cette approche les vecteurs de test sont calculés sur hypothèses de fautes. Suivant le niveau d'abstraction du circuit on distingue deux approches déterministes de génération de test, à savoir l'approche structurelle et l'approche fonctionnelle.

I.2.1 Approche structurelle

Dans cette approche, on considère la machine d'états finis au niveau structurel, c'est-à-dire au niveau bascules et portes logiques.
Cette approche propose de générer les vecteurs de test du circuit en utilisant des algorithmes proches du D-algorithme [Rot66] utilisé principalement pour la génération de test des circuits combinatoires. Cet algorithme emploie des processus de sensibilisation de chemin (propagation avant des signaux) et de justification de valeur sur ces chemins (propagation arrière des signaux). Les algorithmes proposés prennent compte des barrières temporelles dues à la présence des bascules dans les circuits séquentiels [Put71], et proposent une solution au problème de stockage dans le cas de gros circuits [Mar78].
Un autre algorithme très connu pour la génération de test pour circuits combinatoires, PODEM [Goe81], dérivé du D-algorithme, est à la base de plusieurs algorithmes de génération de test pour circuits séquentiels en général et pour des machines d'états finis en particulier. Cet algorithme consiste à affecter des valeurs aux entrées du circuit afin de sensibiliser un chemin (propagation avant des signaux). Plusieurs heuristiques d'affectation des valeurs aux entrées des machines sont proposées [Fuj83], [Abr85] pour améliorer le calcul du chemin de sensibilisation. PODEM est appliqué pour le test des machines d'états finis en prenant compte des barrières temporelles. Il est utilisé pour propager l'effet d'une faute jusqu'aux sorties primaires de la partie combinatoire de la machine dans [Ma87], [Agr88], [Ash90], [Che88], [Pat91]. Pour pallier au problème de la grande mémoire de stockage nécessaire à l'exécution de ces programmes, [Che88] propose un algorithme d'amélioration dans lequel on part d'une sortie pour sensibiliser un chemin par une propagation arrière dans le temps et l'espace.

L'approche structurelle a l'avantage de pouvoir détecter les fautes redondantes [Che91], [Abr79], [Ash90] et de générer la séquence de test des autres fautes. Elle a l'inconvénient d'être lente, nécessitant un grand espace de stockage et par suite inexploitable dans le cas de grandes machines.

I.2.2 Approche fonctionnelle

Cette approche de génération de test est fondée sur le graphe ou la table de transition de la machine d'états finis. Dans [Ma87], [Ma88], [Gho89], la justification de l'état initial qui contrôle l'effet d'une faute est faite sur le graphe de la machine, puis validée par simulation de fautes.

I.2.2.a Hypothèse de fautes fonctionnelles

Cette approche ne tient pas compte de la structure du circuit. Les modèles de fautes fonctionnelles utilisés en général sont les fautes sur les transitions. Dans ce type d'approche deux hypothèses sont déclarées. La première est qu'une seule transition peut être fausse à la fois. La deuxième hypothèse est qu'une transition fausse se manifeste obligatoirement par une faute sur l'état final de la transition.
L'approche consiste à justifier l'état initial de la transition fausse et de chercher la séquence de distinction de l'état final de la transition. Une telle approche est utilisée par [Koh78]. La séquence trouvée a l'avantage de fournir une très

bonne couverture de fautes au niveau structurel, mais l'inconvénient d'être plus longue qu'une séquence qui aurait été générée sur hypothèses de fautes structurelles et pour une même couverture [Che90]. Récemment Cheng a pallié à ce problème [Che91], et a proposé une méthode de test où la séquence est minimisée.
L'avantage de cette méthode est de s'affranchir du niveau logique et de fournir une bonne couverture de fautes au niveau structurel.
Le premier inconvénient de cette méthode est que les hypothèses de fautes ne sont pas réalistes. En effet la présence d'une faute dans le circuit peut corrompre plusieurs transitions et pas seulement une seule. Le deuxième inconvénient de cette méthode est la nécessité du calcul d'un nombre non négligeable de séquences de différentiation entre chacun des états du graphe et les autres.

I.2.2.b Hypothèse de fonctionnement erroné

La méthode de Poage [Poa63] travaille sur les tables de transition de la machine juste et des machines fausses. Toutefois les types de fautes ne sont pas considérés à priori. Il suffit de connaître le comportement des machines fausses sans connaître ou chercher à modéliser les fautes qui ont conduit au comportement faux.
Cette méthode consiste à chercher une séquence de distinction de longueur minimale entre la machine juste et chacune des machines fausses et ceci en calculant le produit entre la table de transition de la machine juste et chacune des tables des machines fausses.
L'inconvénient dans cette méthode est la taille de la mémoire nécessaire pour le stockage des tables de transition et des tables produits surtout lorsqu'il s'agit de grandes machines.

I.3 Test assisté par simulateur

Dans cette approche, un simulateur de fautes assiste à la génération de test elle-même [Agr87], [Che87]; elle consiste à calculer une première séquence de test par une méthode quelconque et à envoyer cette séquence à un simulateur de fautes. Les informations fournies par le simulateur sont exploitées dans le but de compléter la première séquence de test.
[Pat91] utilise les informations de la simulation de la phase de propagation des fautes vers les sorties primaires pour justifier l'état qui contrôle l'effet d'une faute. Cette justification se fait avec l'assistance permanente du simulateur.
Dans [Agr88], elles sont utilisées pour calculer une fonction de coût pour les nœuds du graphe. Ce coût traduit la facilité de contrôler et d'observer un nœud et est par suite pris en compte dans la génération de test des fautes résiduelles.

II Modélisation des machines d'états finis

II.1 Définition d'une machine d'états finis

Une machine d'états finis (ou FSM pour Finite State Machine) peut être représentée comme une machine de Moore (figure **1a**) ou comme une machine de Mealy (figure **1b**). Elle est définie par:

FSM = (I,O,ST,□,□) I: entrée (input)
O : sortie (output)
ST : variable d'état (state variable)
□ : fonctions dédiées aux sorties
□ : fonctions dédiées aux états suivants

$ST_{n+1} = \square (ST_n, I_n)$
$O_n = \square (ST_n, I_n)$ pour Mealy ou $O_n = \square (ST_n)$ pour Moore

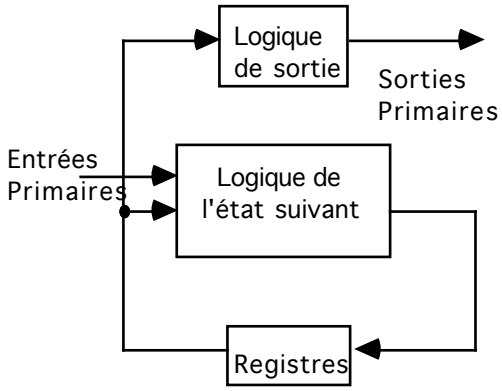

Figure **1a** - Modèle de Moore

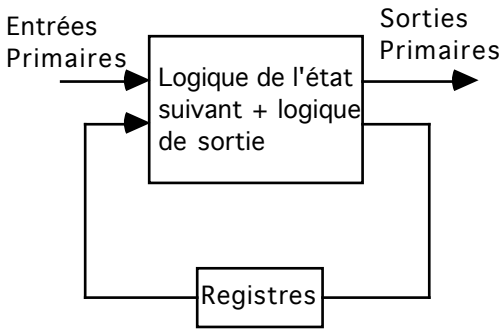

Figure **1b** - Modèle de Mealy

La différence essentielle entre une machine de Moore et de Mealy provient des sorties dépendant soit seulement des sorties des bascules soit des sorties des bascules et des entrées primaires.

II.2 Représentation fonctionnelle

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, codée sur un bit peut prendre l'une des trois valeurs (0, 1, X). Sur une transition, le monôme associé à l'ensemble des valeurs des entrées égales à 0 ou à 1 est appelé prédicat de la transition.

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

Exemple:

Soit l'exemple d'une machine d'états finis à trois états, deux entrées x et y et deux sorties w et z, représentée en graphe Mealy (figure **2a**) et en graphe Moore (figure **2b**).

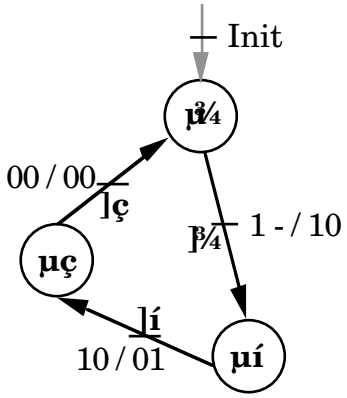

Figure **2a**. Machine Mealy

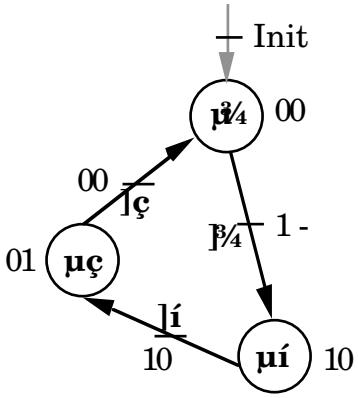

Figure **2b**. Machine Moore

Dans les deux représentations, l'ensemble N des nœuds et l'ensemble A des arcs du graphe associé sont les suivants:
$N = \{n_1, n_2, n_3\}$
$A = \{(n_1, n_2), (n_2, n_3), (n_3, n_1)\} = \{a_1, a_2, a_3\}$
A l'arc a_1 est associée l'entrée 1-, à l'arc a_2 est associée l'entrée 10 et à l'arc a_3 est associée l'entrée 00.
Les prédicats des arcs a_1, a_2 et a_3 sont donnés respectivement par les monômes suivants: x , $x \cdot \overline{y}$, et $\overline{x} \cdot \overline{y}$.

PARTIE II

Cette partie présente la méthode de test développée lors de ce projet. Après quelques rappels sur les graphes, la méthode étudiée est présentée dans ses grandes lignes, puis les différents problèmes rencontrés sont détaillés, et les solutions et algorithmes mis au point sont développés.

I. Rappels de définitions et de théorèmes

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

$d^-(n_i)$ le nombre d'arcs entrants d'un nœud n_i

$d^+(n_i)$ le nombre d'arcs sortants d'un nœud n_i

N_0 le nœud correspondant à l'état d'initialisation

I.1 Définitions générales sur les graphes

Définition 1: Dans un graphe orienté, un *chemin* est une séquence d'arcs [a_1, a_2, ...a_n] dans laquelle l'extrémité terminale d'un arc a_i est l'extrémité initiale de l'arc a_{i+1}. Le nœud extrémité initiale (resp. terminale) de a_i est dit *nœud prédécesseur* (resp. *nœud successeur*) du nœud extrémité terminale (resp. initiale). L'arc a_i est appelé *arc successeur* (resp. *arc prédécesseur*) de son nœud extrémité initiale (resp. extrémité terminale).
Définition 2: Dans un graphe orienté une *chaîne* est une séquence d'arcs [a_1, a_2,...,a_i,...,a_q] dans laquelle les arcs a_i et a_{i+1} ont une extrémité commune.
Définition 3: Une *chaîne élémentaire* est une chaîne telle qu'en la parcourant on ne rencontre pas deux fois le même nœud.

Soit l'exemple du graphe de la figure **3**. La séquence [a_1, a_4, a_5, a_6] constitue le chemin de n_0 à n_3. Les séquences [a_1, a_4, a_5, a_6] et [a_1, a_2, a_3] sont les deux chaînes élémentaires de n_0 à n_3.

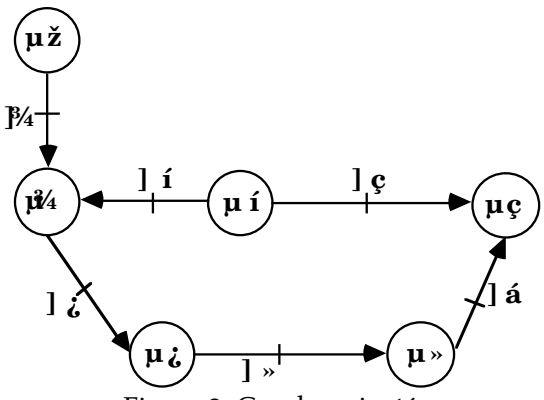

Figure **3**. Graphe orienté

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

$$D(n_i) = d^-(n_i) - d^+(n_i).$$

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

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

Définition 7: Un *arbre* est un graphe connexe sans cycle.

Définition 8: Un *arbre complet d'un graphe* G est un graphe partiel de G connexe et sans cycle.

I.2 Définitions et théorèmes sur les graphes Eulériens

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.

Théorème 2: Un graphe connexe G admet un chemin Eulérien (N_0, N_f) de G si et seulement si

$D(N_0) = -1$,

$D(N_f) = 1$,

$D(n_i) = 0$ pour tout nœud n_i du graphe tel que $n_i \neq N_0$ et $n_i \neq N_f$.

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

I.3 Définitions et théorèmes 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 a_i est associée bijectivement une quantité $f(a_i) \geq 0$ telle que:

1. pour tout nœud n_i du graphe ($n_i \neq S$ et $n_i \neq P$), les arcs prédécesseurs a' et les arcs successeurs a'' vérifient:

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

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

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

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

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 $c(a) \geq 0$ 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 (n_i, n_j) de la chaîne on a:

- si (n_i, n_j) est un arc direct alors
  $c(n_i, n_j) - f(n_i, n_j) > 0$
- si (n_i, n_j) est un arc inverse alors
  $f(n_j, n_i) > 0$

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

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.
$\Delta = \min \Delta(a)$ où $\Delta(a)$ 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 $F' = F + \Delta$.

II. Approche générale de test

L'approche utilisée consiste, à partir du graphe d'états de la machine d'états finis, à parcourir ce graphe en passant au moins une fois par chaque arc; on détermine alors un chemin dans le graphe. La séquence de vecteurs de test ainsi obtenue active donc au moins une fois chaque transition de l'automate.

II.1 Etat d'avancement au début du projet

Lorsque j'ai commencé à travailler sur ce projet, un travail fondé sur cette approche avait déjà été réalisé au LCSI par C. Jay [Jay89]. Ce travail expose une méthode de test à partir du graphe d'états de la machine d'états finis. Le principe de cette méthode est le suivant:

pour déterminer un chemin dans le graphe passant au moins une fois par chaque transition, on transforme le problème en étudiant le graphe dual G* du graphe G considéré. Sur le graphe dual G*, le problème de parcours des arcs du graphe initial G se transforme en une recherche de chemin passant au moins une fois par tous les états du graphe G*. Cette recherche est faite par un algorithme classique "branch and bound". Le chemin trouvé est alors transposé

(transformation des arcs en états et des états en arcs) pour déterminer le chemin correspondant dans le graphe G d'origine; puisque le chemin obtenu dans G* passe par tous les états de G*, le chemin correspondant dans G passe par tous les arcs du graphe G et fournit donc une solution au problème.

D'autre part, certaines entrées peuvent prendre des valeurs indifférentes (notées **X**, et valant indifféremment 1 ou 0); dans la séquence de test définitive, ces valeurs indifférentes doivent être instanciées, et le choix de ces valeurs a une influence sur la qualité de la séquence de test; dans la méthode de C. Jay, les entrées indifférentes sont fixées de façon aléatoire.

Un algorithme fondé sur cette méthode a été implanté dans les outils d'aide à la conception de circuits de la société VLSI Technology, Inc.

Exemple:

Prenons pour exemple la machine "traffic" dont la figure ci-dessous (figure **4**) représente le graphe fonctionnel. Cette machine est un contrôleur de gestion de feux de circulation; elle comporte quatre états (FG, FY, HG, HY), trois entrées (car, tlong et tshort), et cinq sorties (fl_1, fl_0, hl_1, hl_0, start); l'état *reset* est l'état FG.

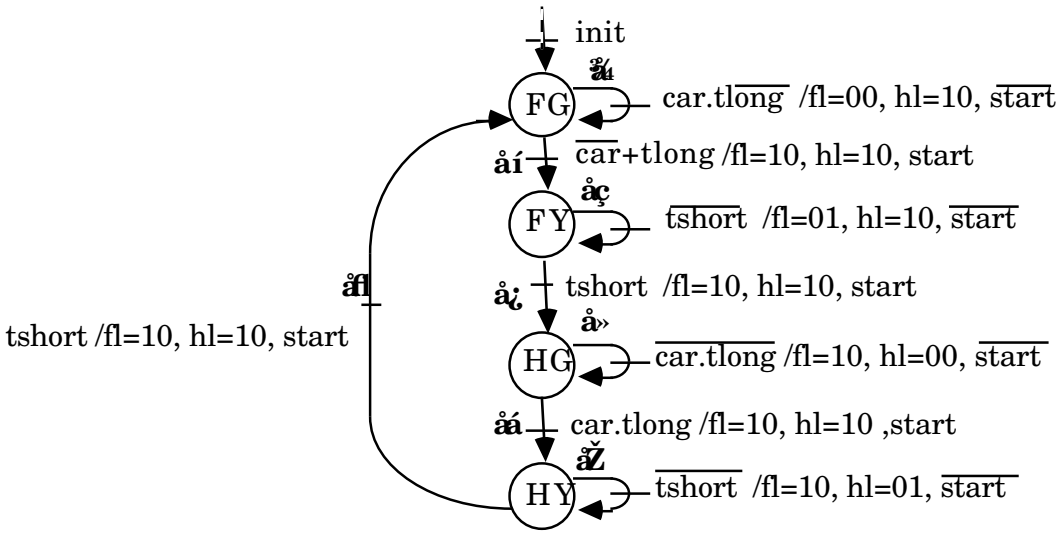

Figure **4** - Graphe fonctionnel G de la machine "traffic"

Le graphe dual G* de traffic est représenté sur la figure **5**:

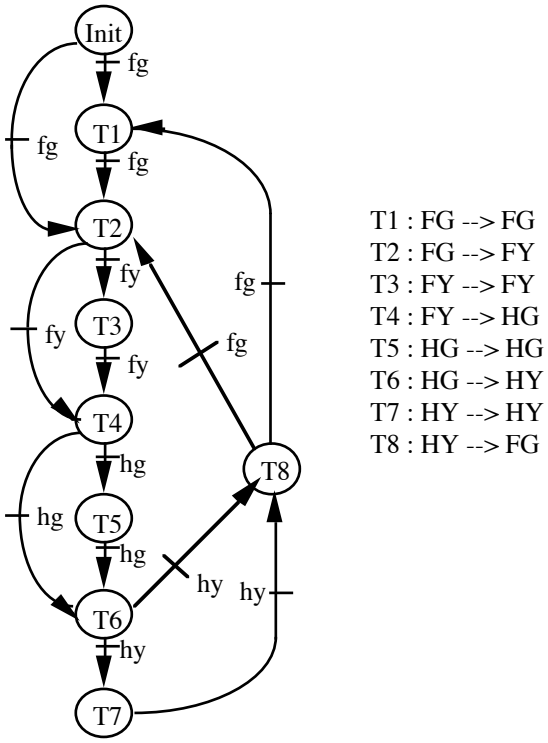

Figure **5** - Graphe dual G* du graphe G

La recherche d'un chemin qui passe par tous les nœuds du graphe dual G* conduit au chemin suivant: [Init, T1, T2, T3, T4, T5, T6, T7, T8]

Ce qui nous fournit le chemin dans G représenté ci-dessous (figure **6**) (les entrées sont, dans l'ordre, car, tlong, tshort, et les sorties, fl_1, fl_0, hl_1, hl_0, start):

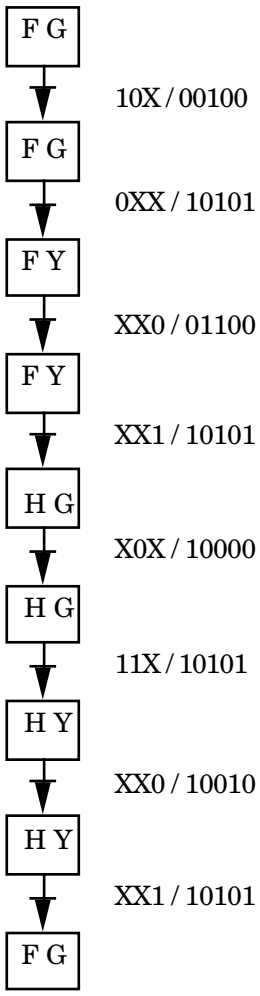

Figure **6** - Parcours de G passant au moins une fois par chaque arc

La séquence d'entrée obtenue est donc:

vecteur 1 10X
vecteur 2 0XX
vecteur 3 XX0
vecteur 4 XX1
vecteur 5 X0X
vecteur 6 11X
vecteur 7 XX0
vecteur 8 XX1

L'affectation des valeurs indifférentes X est faite aléatoirement; soit, par exemple, la séquence de vecteurs de test suivante:

vecteur 1 100
vecteur 2 001
vecteur 3 000
vecteur 4 011
vecteur 5 101
vecteur 6 110
vecteur 7 000

vecteur 8 011

II.2 Inconvénients de la méthode existante

Cette méthode présente plusieurs inconvénients:

- Nécessité du passage au graphe dual, d'où un temps de calcul non négligeable, en particulier pour des graphes de taille importante.
- Le problème de parcours optimal du graphe dual G* en passant au moins une fois par chaque état est analogue au problème, bien connu en recherche opérationnelle, du "voyageur de commerce" [Gib85], qui est np-complet. On peut donc, au mieux, mettre au point une heuristique.

II.3 Améliorations proposées

Afin d'améliorer la méthode de C. Jay, les points suivants ont été retenus:

- Suppression du passage au graphe dual
- Mise au point de nouveaux algorithmes de parcours du graphe en passant au moins une fois par chaque arc, mais en travaillant directement sur le graphe.
- Elaboration d'un algorithme permettant de choisir les valeurs à affecter aux entrées indifférentes.

III. Recherche d'un algorithme de parcours d'un graphe en passant au moins une fois par chaque arc

III.1 Exposé du problème

Les hypothèses générales sur la machine d'états finis à traiter sont les suivantes:

- il existe une commande asynchrone câblée, permettant, à partir de n'importe quel état de la machine, de la faire revenir dans un état d'initialisation *reset*;
- l'état initial *reset* est "robuste" aux fautes, c'est-à-dire qu'il ne peut y avoir de faute sur cet état.

D'après les définitions **9**, **10**, **11**, et les théorèmes **1** et **2**, certains graphes, dits *eulériens*, admettent un chemin passant une et une seule fois par chacun des arcs du graphe. Pour ce type de graphe, il fallait donc mettre au point un algorithme permettant de calculer un chemin eulérien du graphe (cf. **III.2**). Pour un graphe non-eulérien, il a été montré qu'il est possible, en dupliquant certains arcs du graphe, de se ramener au cas eulérien, si le graphe est fortement connexe (théorème **3**). L'algorithme permettant de transformer un graphe non-eulérien en graphe eulérien est détaillé en **III.3**.

III.2 Cas Eulérien

Dans le cas particulier où le graphe considéré est eulérien, on sait, d'après le théorème **2**, qu'il existe au moins un chemin passant une et une seule fois par toutes les transitions du graphe. Par définition, ce chemin est minimal. Deux algorithmes ont été mis au point pour rechercher ce chemin eulérien.

III.2.1 Algorithme de recherche de chemin eulérien par utilisation des matrices d'incidence

Cette solution repose sur certaines propriétés des matrices d'incidence des graphes orientés.
On rappelle que pour un graphe orienté, G (N, A), la matrice d'incidence M est une matrice carrée de dimension n*n (où n est le nombre de nœuds du graphe); chaque élément de la matrice M [i,j] est un entier qui indique le nombre d'arcs directs reliant le sommet i au sommet j.
Cette matrice d'incidence possède la propriété remarquable suivante: si l'on note M^p la puissance $p^{ième}$ de la matrice M, M^p [i,j] a pour valeur le nombre de chemins de longueur p qui relient le sommet i au sommet j.
Dans cette méthode, la stratégie mise en œuvre pour trouver un chemin Eulérien dans un graphe Eulérien est fondée sur le marquage des arcs déjà parcourus et sur la recherche de circuits dans le graphe Eulérien, c'est à dire de chemins ayant le même sommet pour extrémités.
Partant d'un sommet donné, on détermine d'abord s'il existe un circuit partant de ce sommet et qui n'emprunte aucun des arcs déjà marqués. Si oui, on ajoute le circuit trouvé au chemin résultat. Ces circuits sont détectés en élevant la matrice d'incidence à des puissances successives et en considérant les éléments diagonaux. Ces circuits sont ensuite déterminés exactement par un algorithme de recherche d'un chemin de longueur donnée, suivant l'algorithme de Djikstra [Gib85].

Cependant, cet algorithme fait appel à de nombreux calculs matriciels; la taille de la matrice d'incidence étant déterminée par le nombre d'états de la machine, l'algorithme devient très coûteux (à la fois en temps et en mémoire) dès que la machine comporte un grand nombre d'états.

Il aurait été envisageable de mettre au point un algorithme spécifique afin de diminuer le coût des calculs matriciels; en effet, une matrice d'incidence comporte en général de nombreux zéros, et, de plus, ce sont principalement les éléments diagonaux des puissances successives de la matrice qui nous intéressent; le calcul peut donc être optimisé.

Mais un autre algorithme, plus simple, et n'utilisant pas le calcul matriciel a été développé (cf. **III.2.2**).

Exemple:

Prenons pour exemple un graphe eulérien simple, mais qui va permettre de bien mettre en évidence le mécanisme de l'algorithme (sur la figure **7**, on a représenté uniquement le graphe, sans les entrées et sorties de la machine; l'état *reset* est l'état n_1):

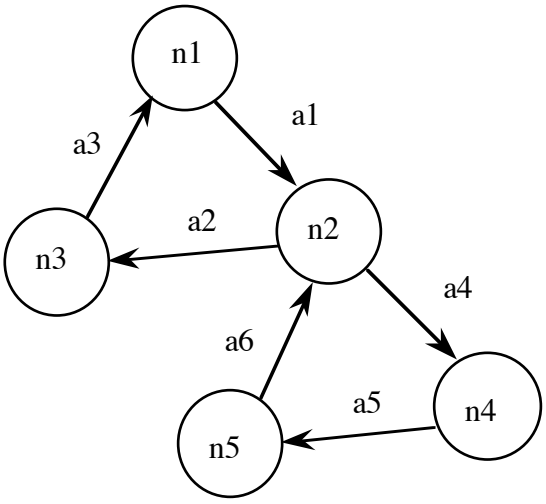

Figure **7** - Exemple de graphe eulérien

L'élément M[1, 1] de la matrice d'incidence M de ce graphe vaut 0 car il n'existe pas d'arc de n_1 à n_1; l'élément M[1, 2] vaut 1 par suite de la présence de l'arc a_1, etc... La matrice d'incidence de ce graphe est:

$$\begin{vmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{vmatrix}$$

L'algorithme va partir de l'état *reset* (n_1), et prendre le seul arc a_1 qui part de n_1; on marque l'arc a_1.
En n_2, deux arcs peuvent être choisis, a_2 ou a_4; l'algorithme va donc essayer de détecter si l'un des deux arcs appartient à un circuit passant par n_2; le cube de la matrice d'incidence fait apparaître qu'il existe deux circuits de longueur 3 passant par n_2; comme le circuit (a_2, a_3, a_1) comporte l'arc a_1 déjà marqué, c'est le circuit (a_4, a_5, a_6) qui est ajouté au chemin.
Les autres étapes du déroulement de l'algorithme sont simples, un seul arc non marqué pouvant être emprunté à chaque étape.
Le chemin déterminé par l'algorithme est donc : [a_1, a_4, a_5, a_6, a_2, a_3].

III.2.2 Algorithme de recherche de chemin eulérien par utilisation des arbres de graphes

Cet algorithme comporte deux étapes:

1. Extraction d'un arbre complet ("spanning tree") du graphe, c'est-à-dire d'un arbre qui contient tous les nœuds du graphe.
2. A chaque nœud, on associe la liste ordonnée de ses arcs prédécesseurs et de ses nœuds prédécesseurs. Les arcs qui appartiennent à l'arbre complet et leurs nœuds correspondants sont placés à la fin de ces listes. Le chemin eulérien

est construit de façon inverse; on part de l'extrémité finale jusqu'à arriver à l'extrémité initiale. A partir du nœud courant, le premier arc qui n'a pas encore été choisi dans la liste des prédécesseurs, est choisi et est ajouté à l'ensemble des arcs sélectionnés. L'origine de cet arc devient le nœud courant. L'algorithme s'arrête lorsque tous les arcs du graphe ont été sélectionnés; on obtient alors un chemin eulérien du graphe.

Cet algorithme a une complexité en O(p), où p est le nombre d'arcs du graphe.

Exemple:

Soit l'exemple du graphe eulérien de la figure **8**. La figure **9** montre un arbre complet du graphe, obtenu par un parcours du graphe en profondeur La figure **10** donne les listes A_{ni} et B_{ni} des nœuds et arcs prédécesseurs du nœud n_i.

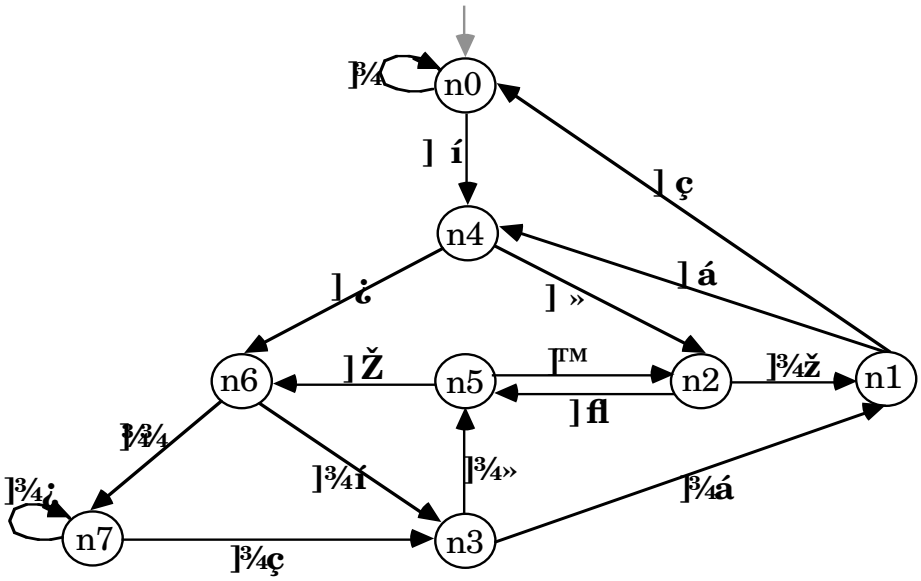

Figure **8** - Exemple de graphe eulérien

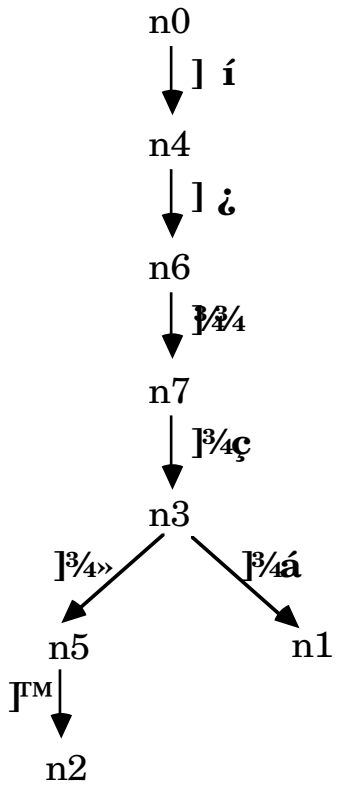

Figure **9** - Un arbre complet du graphe

A_{n0} = [n1, n0]
B_{n0} = [a3 , a1]
A_{u1} = [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]

Figure **10** - Listes des prédécesseurs

Dans cet exemple, le chemin eulérien obtenu est la suite des arcs [a_1, a_2, a_4, a_{11}, a_{14}, a_{13}, a_{15}, a_9, a_8, a_7, a_{12}, a_{16}, a_6, a_5, a_{10}, a_3]

III.3 Cas non-eulérien

Dans le cas où le graphe n'est pas eulérien, on cherche à le rendre eulérien en dupliquant des arcs existants, ce qui revient à les emprunter plusieurs fois dans le chemin final. D'après la théorie du flot dans un réseau de transport [Gib85], il est toujours possible d'obtenir de cette façon un graphe eulérien à partir d'un graphe non-eulérien, à condition que le graphe soit fortement connexe. Cette transformation se fait en quatre étapes:

étape 1: vérifier que le graphe est fortement connexe; sinon, le rendre fortement connexe en ajoutant des arcs d'initialisation asynchrones (algorithme détaillé en annexe **VI**).

étape 2: rajout de deux nœuds virtuels au graphe: une "source" S et un "puits" P; la source est un nœud origine d'un ensemble d'arcs aboutissant à tous les nœuds pour lesquels $d^+ < d^-$. Le puits est un nœud auquel aboutit un ensemble d'arcs ayant pour origine les nœuds pour lesquels $d^+ > d^-$.

étape 3: recherche du flot maximum: un flot dans un graphe associe à chaque arc a une valeur f(a) telle que pour tout nœud n tel que n≠ S et n≠P

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

Le flot maximum d'un graphe, depuis S jusqu'à P a pour valeur:

$$\sum_{a' \in \text{(arcs successeurs de S)}} f(a') = d^+(S) = d^-(P)$$

Dans un tel graphe, le flot de chaque arc est borné par sa capacité. La capacité des arcs (S, n_i) successeurs de S vaut $d^-(n_i) - d^+(n_i)$. La capacité des arcs (n_j, P) prédécesseurs de P vaut $d^-(n_j) - d^+(n_j)$. Les autres arcs du graphe ont une capacité infinie. Une chaîne (S, P) est une séquence d'arcs $[a_1, ..., a_n]$ tels que a_i et a_{i+1} ont un nœud en commun. Une chaîne saturée est une chaîne sur laquelle le flot ne peut être augmenté. Le flot maximum entre S et P est transporté par un ensemble de chaînes saturées. L'algorithme de calcul d'une chaîne saturée est détaillé en annexe **IV**.

Exemple:

Considérons l'exemple du graphe non-eulérien suivant (figure **11**)

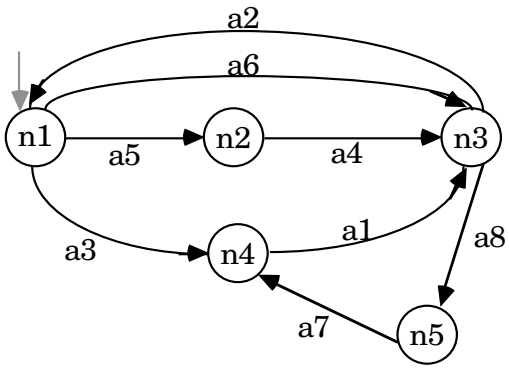

Figure **11** - Un graphe non-eulérien

Dans cet exemple, deux chaînes saturées, $[(S, n_3), (n_3, n_1), (n_1, P)]$ et $[(S, n_4), (n_4, n_3), (n_3, n_1), (n_1, P)]$ transportent chacune un flot de 1 et fournissent donc un flot maximum de 2 (figures **12** et **13**)

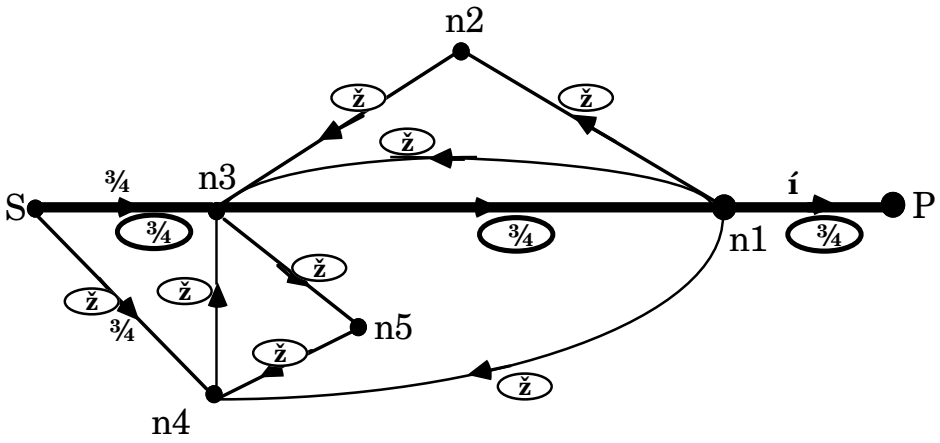

Figure **12** - Première chaîne saturée

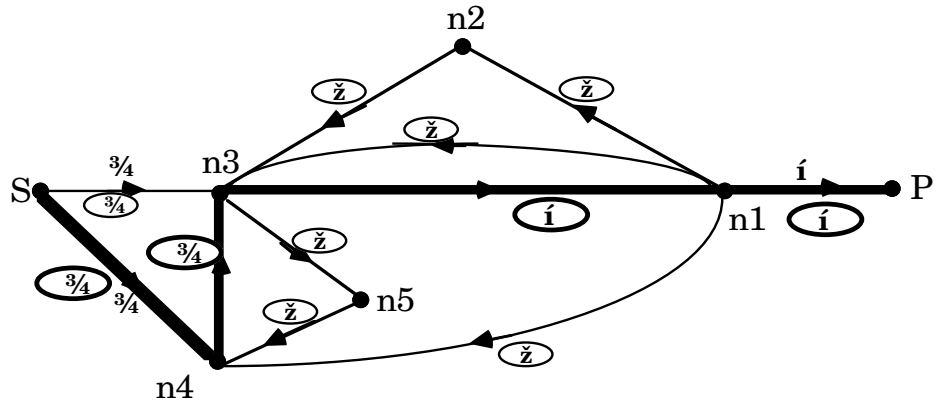

Figure **13** - Deuxième chaîne saturée

étape 4: Duplication des arcs: la source S et le puits P sont supprimés ainsi que les arcs successeurs de S et les arcs prédécesseurs de P. Les arcs pour lesquels on a obtenu un flot non nul sont dupliqués un nombre de fois égal à la valeur de leur flot. Le graphe obtenu est alors eulérien. Dans notre exemple, les arcs a_1 et a_2 sont dupliqués respectivement une et deux fois.

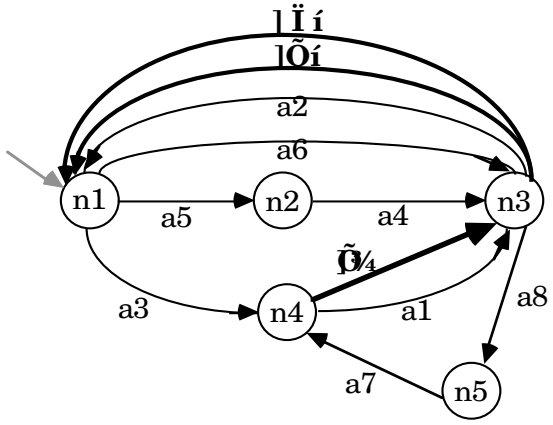

Figure **14** - Le graphe eulérien obtenu

Un graphe non-eulérien étant de cette manière transformé en graphe eulérien, on s'est ramené au cas eulérien, et la méthode exposée plus haut peut être appliquée afin de déterminer un chemin eulérien dans ce graphe.

La complexité de cet algorithme est un polynôme en n et p, où n est le nombre de nœuds, et p le nombre d'arcs du graphe.

III.4 Implantation et résultats

Les algorithmes exposés ci-dessus ont été implantés en C sous Unix (≈ 250 lignes pour l'algorithme utilisant les matrices d'incidence, ≈ 150 lignes pour l'algorithme utilisant les arbres, ≈ 300 lignes pour l'algorithme transformant un graphe non-eulérien en graphe eulérien).

Le programme implantant ces algorithmes a été utilisé pour traiter des circuits de référence (benchmarks) de machines d'états finis. Ces circuits sont fournis au programme sous forme de fichiers au format "kiss", qui donne l'état suivant et la sortie en fonction de l'état courant et de l'entrée (un exemple de fichier au format kiss est donné en annexe **VII**).

Les résultats obtenus, sur station de travail Sun4, pour quelques benchmarks, sont détaillés dans le tableau **1**; on donne pour chaque machine, le nombre d'états et d'arcs de la représentation sous forme de graphe, le nombre d'entrées et de sorties de la machine, et les résultats proprement dits, pour l'ancienne méthode (C. Jay) et pour l'algorithme utilisant la nouvelle méthode, c'est-à-dire le nombre d'arcs dupliqués, la longueur de la séquence de test, et le temps nécessaire à la génération de la séquence de test:

	nb. états	nb. arcs	nb. entrées	nb. sorties	graphe eulérien	nb. arcs dupliqués		longueur de la séquence de test		temps de calcul (secondes)	
						anc. méth.	nouv. méth.	anc. méth.	nouv. méth.	anc. méth.	nouv. méth.
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	59	383	166	357.6	0.46
planet	48	116	9	19	non	613	171	729	287	279.6	2
hyeti	175	455	29	71	non	2416	746	2871	1201	2638	493

Tableau **1** - Résultats de l'algorithme de parcours de graphe

On remarque que la nouvelle méthode diminue de façon considérable le nombre d'arcs dupliqués et donc la longueur de la séquence de test. Le temps de calcul est également bien meilleur.

IV. Détermination des entrées indifférentes, dans le but d'améliorer la couverture

IV.1 Exposé du problème

Dans la séquence de test obtenue, les valeurs à appliquer aux entrées de la machine d'états finis comportent des valeurs indifférentes ("don't care", notées **X**). Nous avons cherché à instancier ces valeurs indifférentes dans le but d'améliorer le taux de couverture. Plusieurs algorithmes ont été envisagés.

On définit le taux de couverture, comme le pourcentage du nombre de fautes détectées sur le nombre total de fautes considérées. Le facteur de qualité est le rapport entre le taux de couverture et la longueur de la séquence de test.

Le taux de couverture est calculé par un simulateur de fautes (de type HiLo ou Mach1000), à partir de la description physique du circuit, de la séquence de vecteurs de test que nous avons obtenue, ainsi que d'hypothèses sur les fautes à détecter. Le simulateur de fautes fournit aussi le nombre et la liste des fautes non détectées.

IV.2 Algorithme aléatoire et autres algorithmes simples

Dans la méthode développée par C. Jay, les valeurs indifférentes sont déterminées de façon entièrement aléatoire. Il est apparu que la façon de fixer ces entrées pouvait avoir un impact important sur le taux de couverture obtenu.

Le premier algorithme essayé, et qui nous sert de référence pour les résultats, consiste simplement à fixer toutes les valeurs indifférentes à zéro. Les résultats sont variables suivant le circuit, pouvant fournir un taux de couverture peu satisfaisant.

Ceci nous a conduit à développer un second algorithme, visant à améliorer ce taux de couverture.

IV.3 L'algorithme de détermination des entrées à valeurs indifférentes

Cet algorithme cherche à obtenir un maximum de valeurs distinctes pour les entrées.

Pour chaque entrée prenant une valeur indifférente, on essaie tout d'abord de lui donner ses deux valeurs possibles (0 et 1). Par exemple, si les trois premiers vecteurs de la séquence de test sont:

	E_3	E_2	E_1
vecteur 1	0	1	1
vecteur 2	0	0	1
vecteur 3	0	0	X

...

Le X présent dans le vecteur 3 est instancié à zéro.

De façon plus générale, si un vecteur contient plusieurs entrées indifférentes:

1. toutes les entrées qui n'ont pas encore, dans les vecteurs précédents, pris les valeurs 1 et 0, sont instanciées, suivant le cas, à 1 ou à 0.

2. Pour les entrées indifférentes restantes $E_{i1}...E_{ip}$ et qui ont déjà pris les deux valeurs 0 et 1, on choisit une des valeurs de $\{E_{i1}, ..., E_{ip}\}$ parmi les 2^p possibles, en évitant toutes les valeurs déjà prises dans les vecteurs précédents par $\{E_{i1}, ..., E_{ip}\}$.

Par exemple, dans la séquence suivante:

	E_3	E_2	E_1
vecteur 1	0	1	0
vecteur 2	0	X_1	1
vecteur 3	0	X_2	X_3

...

X_1 est mis à 0 (E_2 prend déjà la valeur 1 dans le vecteur 1, mais jamais la valeur 0); puis pour (X_2, X_3), comme E_2 et E_1 ont déjà chacun pris les deux valeurs 0 et 1, on choisit une valeur qui n'a pas encore été prise par $\{E_3, E_2, E_1\}$, donc soit (0, 0, 0), soit (0, 1, 1).

IV.4 Résultats

Cet algorithme a été implanté en C sous Unix (≈ 250 lignes), et produit une séquence de vecteurs de test.

On a utilisé ensuite le simulateur de fautes HiFault de GenRad auquel on a fourni le fichier de description du circuit ("netlist", fournie par l'outil de génération automatique de machines d'états finis de VLSI Technology), le fichier contenant la séquence de test générée par notre algorithme, et un fichier définissant les fautes à simuler. Les fautes considérées sont le collage à 0 et à 1 des entrées et sorties des cellules standard du circuit. Le simulateur nous donne le taux de couverture correspondant.

Les résultats obtenus, sur station de travail Sun4, pour l'algorithme aléatoire, l'algorithme de référence, et l'algorithme définitif, sont présentés

dans le tableau suivant qui compare pour les différentes méthodes, la longueur de la séquence de test, le taux de couverture, et le facteur de qualité.

	longueur de la séquence de test		taux de couverture (%)			facteur de qualité		
	anc. méth.	nouv. méth.	anc. méth.	référence	nouv. méth.	anc. méth.	référence	nouv. méth
u802b	124	63	84.5	85.3	87.6	0.68	1.35	1.4
flammand	52	42	85.7	83.1	87.3	1.65	1.98	2.08
mouse	10	5	82	78	84	8.2	15.6	16.8
big3	25	20	88.8	67.5	72.5	3.55	3.37	3.63
traffic	11	9	92.6	80.8	86.3	8.42	8.98	9.59
s1	383	166	85.3	82.8	86.6	0.22	0.5	0.52
planet	729	287	92.3	90.1	95.2	0.13	0.31	0.33
hyeti	2871	1201	92.3	nc	nc	0.03	nc	nc

Tableau **2** - Taux de couverture et facteur de qualité des différents algorithmes

On constate que le nouvel algorithme, avec des séquences de test nettement moins longues, donne le meilleur facteur de qualité; le nombre de fautes non détectées, qui devront être traitées dans la seconde phase, est quasiment équivalent.

PARTIE III

V. Phase complémentaire visant à obtenir un taux de couverture de 100%

La séquence de vecteurs de test déterminée par la première approche ne garantit pas un taux de couverture de 100%. Il faut maintenant déterminer une séquence de test complémentaire pour couvrir les fautes non détectées. Cette séquence pourrait être obtenue par la méthode classique du D-algorithme, mais à condition de revenir à la description physique du circuit.

Nous proposons une approche déterministe, à partir du modèle fonctionnel, fondée sur la méthode de Poage [Poa63]. Cette méthode consiste, à partir de l'état reset, à déterminer les séquences d'entrée permettant de distinguer entre la machine juste et chacune des machines fausses en présence d'une faute.

V.1 La méthode de Poage

La méthode de Poage consiste à générer des séquences de distinction entre la machine juste et d'autres machines fausses en se fondant sur les tables de transition.
Soit T la table de transition de la machine juste (la table de transition donne la sortie et l'état suivant en fonction de l'entrée et de l'état courant). Pour chacune des fautes i, la table de transition T_i de la machine fausse M_i est construite. Cette construction est simple si les expressions des variables internes et des sorties primaires sont données en fonction des entrées et des sorties des blocs (portes logiques et bascules réelles) sur lesquelles se manifestent les fautes. Des produits cartésiens entre la table de transition T de la bonne machine et celle de chacune des autres fausses machines sont construits. A partir d'un état initial produit n n_i et d'une entrée déterminée sont calculés les termes produits ns ns_i / out out_i où n, ns, out et n_i, ns_i, out_i désignent l'état courant, l'état suivant et la sortie de la machine juste M et de la fausse M_i respectivement.
Dans les tables produits construites, un terme ns ns_i / out out_i avec les sorties de la machine juste M et de la fausse M_i différentes (out $\neq$ out_i) permet de distinguer la machine M de la machine M_i.

Exemple:

Illustrons cette méthode sur une machine décrite dans [Cha68]; dans cet exemple, on considère quatre fautes. Les tables de transition de la machine juste T et des machines fausses T_1, T_2, T_3 et T_4 sont données dans le tableau **3**.

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

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

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

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

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

Tableau **3** - Tables de transition pour la machine juste et les machines fausses

Les produits cartésiens T*Ti (pour i=1 à 4) sont donnés dans le tableau **4**. Les sorties soulignées indiquent des sorties différentes pour la machine juste et la machine fausse.

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

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

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

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

Tableau **4** - Produits cartésiens T * Ti

Ces produits sont rassemblés dans une seule table (tableau **5**) qui donne les produits cartésiens T*T1, T*T2, T*T3, et T*T4. Dans cette table, une étoile correspond à des sorties distinctes. Il faut trouver une séquence de longueur minimale permettant de distinguer la machine juste de chacune des machines fausses. La séquence doit donc passer par au moins une étoile de chaque colonne. Ce mécanisme peut être vu comme une simulation descendante en parallèle, partant d'un état reset robuste aux fautes, et guidée par le modèle fonctionnel. Dans cet exemple, la séquence de longueur minimale permettant de distinguer la machine juste des machines fausses est 0 1 0 1 1.

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

Tableau **5** - Produit cartésien général

V.2 Proposition d'amélioration

On souhaite éviter le calcul de toutes les tables de transition, pour éviter l'explosion combinatoire dans le cas d'un nombre de fautes résiduelles important, en se limitant uniquement aux termes intéressants. On a donc étudié

plusieurs heuristiques, utilisant la table de transition de la machine juste (qui est connue d'après le graphe fonctionnel), qui nous permettent de construire la séquence de test pas à pas, avec l'assistance d'un simulateur de fautes. Un exemple de ces heuristiques est fondé sur un parcours en profondeur d'abord, fait en parallèle sur le graphe et la table de transition de la machine juste.
On part de l'état initial, et on choisit la première valeur possible des entrées; puis l'on examine sur le simulateur si cette première valeur nous a permis de détecter une faute.
Si oui, on garde ce premier élément de séquence, et l'on réitère pour l'état suivant jusqu'à ce qu'on ait détecté toutes les fautes.
Si cette première valeur n'a détecté aucune faute, on essaie avec une autre valeur possible des entrées.
Si pour un état, aucune des valeurs possibles des entrées n'a permis de détecter une faute, on passe à l'état suivant, en choisissant la valeur d'entrée de façon à passer dans un état non encore examiné.
Lorsqu'on arrive au dernier état, si il reste encore des fautes non-détectées, on revient en arrière (backtrack) pour examiner les couples (état, entrées) restants.

Exemple:

Dans l'exemple suivant, il reste trois fautes à détecter. La table de transition de la machine juste est donnée dans le tableau **6**.

	000	100	010	110	001	101	011	111
00	00011/10	00010/00	00011/10	00011/10	00011/10	00010/00	00011/10	00011/10
10	10010/10	10010/10	10010/10	10010/10	10011/11	10011/11	10011/11	10011/11
11	01000/11	01000/11	01000/11	01001/01	01000/11	01000/11	01000/11	01001/01
01	01100/01	01100/01	01100/01	01100/01	01101/00	01101/00	01101/00	01101/00

Tableau **6** - Table de transition de la machine juste

On part de l'état initial 00, et on fixe la première entrée à la première valeur possible, 000. Le simulateur de fautes nous indique que cette séquence détecte les fautes 1 et 3.
On passe alors dans l'état suivant, 10; on essaye successivement les valeurs d'entrée possibles (parcours en largeur), mais aucune ne détecte de nouvelles fautes (indication fournie par le simulateur). On cherche alors une valeur d'entrée qui fasse passer à un état non encore visité. La première valeur qui satisfait ce critère est 001 que l'on rajoute à la séquence de test. Le même mécanisme est appliqué aux deux états qui restent à examiner, 11 (aucune faute détectée; on choisit l'entrée 110), et 01 (aucune faute détectée). Dans l'état 01, on a visité tous les états, et il reste des fautes à détecter. On revient donc en arrière; les états 01, 11, 10 ont déjà été examinés pour toutes les valeurs d'entrée possibles, et aucune ne détecte de faute. On revient donc jusqu'à l'état 00, à la première entrée qui n'avait pas été examinée (reprise du parcours en largeur); les entrées 100 et 010 ne détectent aucune faute, mais l'entrée 110 détecte la faute 2 (indiqué par le simulateur).

On a donc détecté toutes les fautes et obtenu la séquence représentée figure **15**:

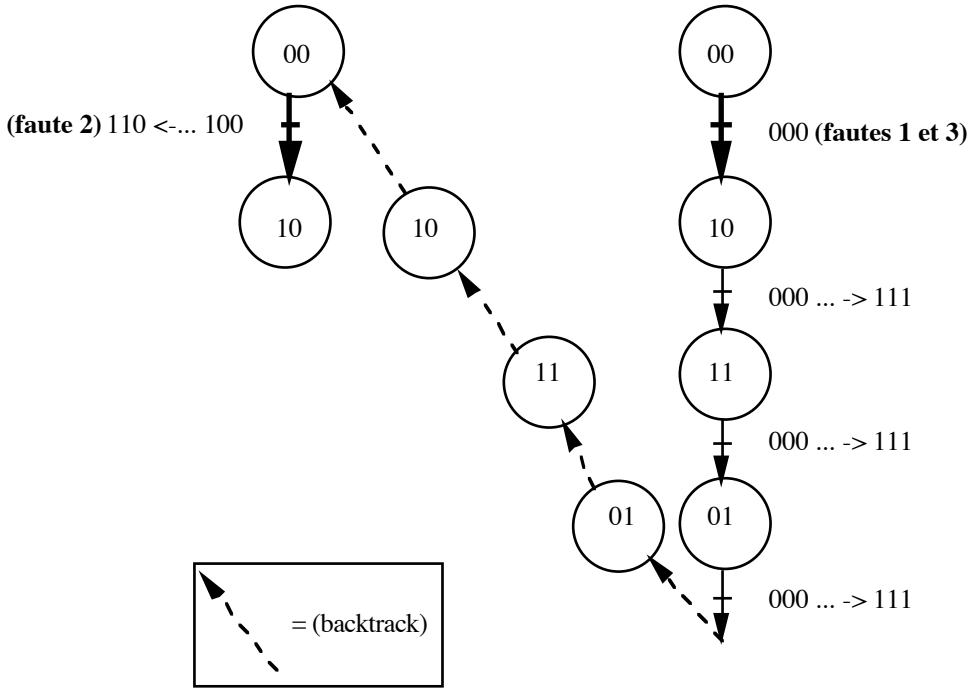

Figure **15** - Séquence de test détectant les 3 fautes restantes

On supprime alors les pas de la séquence qui ne détectent aucune faute; la séquence finale obtenue est donc:

000
reset
110

Conclusion

On a proposé dans ce rapport une méthode de test de machines d'états finis fondée sur deux approches complémentaires.
La première utilise le graphe fonctionnel de la machine; on parcourt ce graphe de façon à activer toutes les transitions, et on utilise un algorithme pour fixer les valeurs aléatoires. Les résultats obtenus sont très encourageants: le taux de couverture est satisfaisant, et la longueur de la séquence de test ainsi que le temps de génération sont particulièrement réduits par rapport aux méthodes existantes.
La deuxième est dérivée de la méthode de Poage. On cherche à compléter la séquence obtenue par la première approche, de façon à détecter toutes les fautes résiduelles, avec l'assistance d'un simulateur de fautes. Cette phase complémentaire semble prometteuse, mais doit encore être approfondie.

Ces travaux ont fait l'objet de deux publications:

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, Grenoble, France, 5-7 novembre 1990.

C. Jay, M. Karam, D. Salber, G. Saucier
"Test generation of controllers using the synthesis specifications"
Euro ASIC '91
Paris, France, 27-31 mai 1991.

Bibliographie

[Abr79] M. Abramovici et M. A. Breuer, "On Redundancy and Fault Detection in Sequential Circuits", *IEEE Transactions on Computers*, Vol. C-28, pp. 864-865, novembre 1979.

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

[Agr87] V. D. Agrawal et K.-T. Cheng, "Threshold-Value Simulation and Test Generation", *Testing and Diagnosis of VLSI and ULSI*, (Proc. NATO Adv. Study Inst., Como, Italy, 1987.

[Agr88] V. D. Agrawal, K.-T. Cheng, P. Agrawal, "Contest: a concurrent test generator for sequential circuits", *25th ACM/IEEE Design Automation Conference*, pp. 84-89, 1988.

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

[Bre71] M. A. Breuer, "A Random and an Algorithmic Technique for Fault Detection Test Generation for Sequential Circuits", *IEEE Transactions on Computers*, Vol. C-20, n° 11, pp. 1366-1370, novembre 1971.

[Cha68] H. Y. Chang, E. Manning, G. Metze, "Fault diagnosis of digital systems", Wiley-interscience, 1968.

[Che87] K.-T. Cheng and V. D. Agrawal, "A Simulation-Based Directed-Search Method for Test Generation, *Proc. Int. Conf. Comp. Des. (ICCD'87)*, Port Chester, NY, pp. 48-51, octobre 1987.

[Che88] K.-T. Cheng, "The back algorithm for sequential test generation", *1988 International Conference on Computer Design*, pp. 66-69, octobre 1988.

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

[Che91] K.-T. Cheng, "An ATPG-Based Approach to Sequential Logic Optimization", *MCNC*, 1991.

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

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

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

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

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

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

[Jac89] R. Jacoby, P. Moceyunas, H. Cho, et G. Hachtel, "New ATPG techniques for logic optimization", *IEEE International Conference on Computer Aided Design*, pp. 548-551, novembre 1989.

[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.

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

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

[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, octobre 1988.

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

[Mar88] R. Marlett, "An Effective Test Generation System for Sequential Circuits", *23rd Design Automation Conference*, pp. 250-256, 1988.

[Nit85] S. Nitta, M. Kawamura et K. Hirabayashi, "Test Generation by Activation and Defect-Drive (TEGAD)", *INTEGRATION*, the VLSI journal 3 3-12, 1985.

[Pat91] J. H. Patel, T. Niermann, "HITEC: A Test Generation Package for Sequential Circuits", *EDAC*, 1991.

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

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

[Rot66] P. Roth, "Diagnosis of automata Failures: a calculus and a method", *IBM journal*, juillet 1966.

[Sch75] H. D. Schnurmann, E. Lindbloom, et R. G. Carpenter, "The Weighted Random Test-Pattern Generator", *IEEE Transactions on Computers*, Vol. C-24, n° 7, pp. 695-700, juillet 1975.

Annexes

I. Algorithme de recherche d'un chemin Eulérien dans un graphe Eulérien (recherche par listes)

II. Algorithme de recherche d'un arbre complet d'un graphe: parcours en profondeur

III. Rendre Eulérien un graphe: Algorithme de recherche d'un flot maximal Fmax

IV. Calcul de la plus petite chaîne saturée de S à P

V. Algorithme de recherche de la longueur de la plus petite chaîne de S à P

VI. Algorithme permettant de rendre fortement connexe un graphe

VII. Exemple de fichier de description de machine d'états finis (format kiss)

Annexe I

Algorithme de recherche d'un chemin Eulérien dans un graphe Eulérien (recherche par listes)

a - Recherche d'un arbre T du graphe G (N, A) ayant le nœud N_0 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
    B_v <-- 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** il existe v_j tel que $D(v_j)>0$ **alors**
      CE <-- [ ] , NC <-- v_j
  Sinon CE <-- [ ] , NC <-- N_0
  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 un arbre T ayant N_0 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'arbre 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) prédécesseurs 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 prédécesseurs de v. Dans ces listes l'arc (u_i, v) qui appartient à l'arbre T et le nœud u_i 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 v_j comme étant le nœud courant NC. Le nœud v_j est le nœud de degré positif $D(v_j)>0$ dans G. Dans le cas de la recherche d'un cycle Eulérien, CE est initialisée à [ ] et on définit le nœud N_0 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 prédécesseurs à NC qui n'ont pas encore été traversés.

Pas f, g, h
Un arc AT prédécesseur au NC est alors choisi pour être traversé. Le nœud prédécesseur 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 prédécesseur 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 prédécesseur 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 prédécesseur AT à la tête de la liste CE (pas h).

Annexe II

Algorithme de recherche d'un arbre complet d'un graphe: parcours en profondeur

Notre algorithme de recherche d'un arbre 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'arbre recherché a une racine N_0 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 <-- n_i
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 **alor**s
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'arbre. 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'un arbre 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 n_i racine de l'arbre (pas e).

Pas f, g
La procédure de la recherche de l'arbre (RA) est appelée avec le nœud courant n_i 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 égal au numéro de l'itération précédente (pas i). Ainsi la racine n_i 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 III

Rendre Eulérien un graphe: Algorithme de recherche d'un flot maximal Fmax

a - Ajouter au graphe G, une source S et un puits P. Soit G' le graphe obtenu. Dans G', S est un nœud prédécesseur 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, n_i) successeur à S est égale au degré de son nœud successeur D(n_i). La capacité d'un arc (n_j, P) prédécesseur à P est égale au degré de son nœud prédécesseur D(n_j). Les autres arcs du graphe G' ont une capacité infinie. La valeur du flot des arcs de G' est initialisée 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: $\Delta = \min \Delta(n_i, n_j)$ où (n_i, n_j) sont les arcs de la chaîne.
h - **Pour tout** (n_i, n_j) de la chaîne **faire**
i - **si** (n_i, n_j) est direct dans la chaîne **alors**
j - $f(n_i, n_j) \leftarrow f(n_i, n_j) + \Delta$
k - **sinon** $f(n_j, n_i) \leftarrow f(n_j, n_i) - \Delta$
l - $F \leftarrow F + \Delta$
fin
fin
m - **si** path = true **alors**
début
n - effacer les arcs successeurs de S et prédécesseurs de P
o - **Pour tout** a_i tel que $f(a_i) \neq 0$ **faire**
p - ajouter un nombre d'arcs égal à $f(a_i)$
fin

Annexe IV

Calcul de la plus petite chaîne saturée de S à P

a - Calculer la longueur L de la plus petite chaîne: recherche en largeur d'abord
b - **si** $L \leq 2$ **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 prédécesseurs de n et
augmentant le flot */
d - Chaîne <-- [ ]
e - NC <-- P
f - **tant que** $NC \neq S$ **faire**
début
g - rechercher l'arc (n_i, NC) appartenant à B'(NC) et
tel que
$L(n_i) = L(NC) - 1$
h - rajouter (n_i, NC) à la tête de Chaîne
i - NC <-- n_i
fin
fin

Annexe V

Algorithme de recherche de la longueur de la plus petite chaîne de S à P

```
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
        fin
q - L <-- L(P) - 1
```

Annexe VI

Algorithme permettant de rendre fortement connexe un graphe

a- Chercher les composantes fortement connexes du graphe G
b- Construire le graphe réduit où les nœuds sont les composantes fortement connexes C_i; un arc (u, v) de C_i à C_j existe s'il existe au moins un arc (u_0, v_0) dans G allant d'un état de C_i à un état de C_j. (on note que l'arc (v, u) n'existe pas, car le graphe réduit est un graphe sans cycle.
c- Les arcs "reset" (arcs d'initialisation) à rajouter sont des arcs allant d'un état u de C_i tel que $d^-(C_i)=0$; u choisi dans C_i est un état tel que D(u) dans G est positif strictement.

Annexe VII

VII. Exemple de fichier de description de machine d'états finis (format kiss)

Fichier au format kiss décrivant la machine d'états finis "traffic":

```
.i 3
.o 5
.p 10
.s 4
0-- HG HG 00010
-0- HG HG 00010
11- HG HY 10010
--0 HY HY 00110
--1 HY FG 10110
10- FG FG 01000
0-- FG FY 11000
-1- FG FY 11000
--0 FY FY 01001
--1 FY HG 11001
.e
```