Diagnostic des erreurs de conception dans les circuits digitaux : le cas des erreurs simples

Ayman Wahba

Thèse

présentée par

Ayman MOHAMED MOHAMED HASSAN WAHBA

pour obtenir le titre de Docteur

de l'Université Joseph Fourier – Grenoble 1

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

(spécialité Informatique)

—

DIAGNOSTIC DES ERREURS DE CONCEPTION DANS LES CIRCUITS DIGITAUX: LE CAS DES ERREURS SIMPLES

—

Thèse soutenue le 7 mai 1997

Composition du jury :

Président :	M. Guy MAZARÉ
Rapporteurs :	M. François ANCEAU
	M. Charles TRULLEMANS
Directrice :	Mme. Dominique BORRIONE
Examinateurs :	M. Einar AAS
	M. Jean-François AGAESSE

Thèse préparée au sein du laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architecture (TIMA)

Remerciements

Je tiens à exprimer ma très profonde reconnaissance à Madame Dominique Borrione, directrice de ma thèse, pour l'aide scientifique et morale qu'elle m'a toujours apportée, et pour l'intérêt qu'elle a bien voulu me porter au cours de mes années d'études; qu'elle soit vivement remerciée.

J'adresse également des remerciements à Monsieur Guy Mazaré, Directeur de l'ENSIMAG, qui a bien voulu me faire l'honneur de présider le jury de cette thèse.

Je tiens aussi à remercier Messieurs François Anceau et Charles Trullemans qui ont eu la gentillesse de bien vouloir être rapporteurs de cette thèse. Je les remercie très chaleureusement pour leurs commentaires judicieux et pour le temps qu'ils m'ont accordé.

Des remerciements très spéciaux pour Monsieur Einar Aas qui n'a pas hésité à venir de la Norvège pour faire partie de ce jury. Je lui suis extrêmement reconnaissant pour toutes les discussions fructueuses et tous les conseils pertinents au cours de ce travail.

Je voudrais aussi remercier M. Jean-François Agaesse pour m'avoir fait l'honneur de participer au jury de ma thèse, et pour le temps qu'il m'a accordé et l'intérêt qu'il a manifesté à l'égard de mon travail.

Je tiens aussi à exprimer ma gratitude à Monsieur et Madame Le Faou pour leurs encouragements, et leur soutien moral pendant des périodes difficiles. Je remercie également Monsieur Hakim Bouamama qui m'a apporté son savoir et son amitié, et pour nos échanges de vue enrichissants.

Un grand merci à tous les membres de l'équipe et en particulier Adam Morawiec, Julia Dushina, Pierre Pomes, et Gérard Vitry pour l'amitié qu'ils m'ont montrée. Merci également à toutes les secrétaires de l'ancien laboratoire ARTEMIS et du laboratoire TIMA, notamment Madame Claudine Meyrieux, Madame Corinne Durand-Viel, et Madame Isabelle Essalhiene pour leur gentillesse et pour l'aide qu'elles m'ont apportée.

À mes parents ...

Rêve de grandes choses,
cela te permettra d'en faire au moins de toutes petites.

Table des matières

Abréviations

ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BDD	**B**inary **D**ecision **D**iagram
C	**C**omposant
CAG	**C**olumn **A**ddress **G**enerator
Classe-R	**Classe** de **R**emplacement
EDR	**E**space **D**e **R**echerche
esp	Ensemble d'**é**tats **s**uivants **p**ossibles
FSM	**F**inite **S**tate **M**achine
I_{bon}	Ensemble de nœuds desquels la bonne connexion ne peut pas provenir
I_m	Ensemble de nœuds desquels la connexion manquante ne peut pas provenir
I_x	Ensemble des entrées insuppressibles
IMPL	**Impl**émentation
P	**P**orte
P_{bon}	Ensemble de nœuds desquels la bonne connexion peut provenir
P_m	Ensemble de nœuds desquels la connexion manquante peut provenir
$P_{mauvais}$	Ensemble de mauvaises entrées d'une porte
P_x	Ensemble des entrées d'une porte potentiellement en trop
PODEM	**P**ath **O**riented **D**ecision **M**aking
S_x	Ensemble des entrées sûrement en trop
SPEC	**Spéc**ification
Sus	Fonction de **Sus**picion
V_{comp}	**V**aleur **Comp**atible
VCourante	**V**aleur **Courante**

VDE	**V**ecteur **D**étectant l'**E**rreur
VNE	**V**ecteur **n**e Détectant pas l'**E**rreur
VRequise	**V**aleur **Requise**
VT	**V**ecteur de **T**est
W	La sortie de la spécification
Y	La sortie de l'implémentation

Chapitre 1

Introduction

1.1 Position du problème

Le progrès rapide dans le domaine de la technologie VLSI a mené à des systèmes digitaux à un très haut niveau de complexité. La détection tardive d'erreurs de conception est très coûteuse, surtout si le produit est déjà commercialisé. Il est donc absolument nécessaire de découvrir toutes les erreurs pendant les toutes premières phases du processus de conception, et d'effectuer une vérification après chaque étape de conception, et non pas seulement après la phase finale.

L'utilisation d'outils de synthèse automatisés, qui génèrent des produits *corrects par construction*, peut rendre la vérification inutile. Cependant, ces outils ne sont pas aujourd'hui encore suffisamment avancés pour produire des circuits efficaces sous certaines contraintes, quand la vitesse ou la surface, par exemple, sont des facteurs critiques. Les circuits générés par ces outils sont souvent modifiés manuellement pour améliorer leurs performances temporelles ou pour obtenir des structures plus compactes. Dans d'autres cas, les circuits sont modifiés manuellement pour effectuer de petits changements dans la spécification.

Après l'application des procédures de synthèse automatique, le concepteur a un petit aperçu des détails internes de la conception qui en résulte. Les modifications manuelles dans cette conception ont une très grande chance d'insérer des erreurs involontaires.

Par ailleurs, la sûreté de fonctionnement des outils de synthèse n'est pas toujours garantie, à cause de leur haut niveau de complexité et des modifications

apportées sans cesse à ces outils pour faire face à la technologie qui évolue très rapidement. C'est pourquoi, dans la pratique, un effort remarquable est fourni pour vérifier que la conception produit, pour toute combinaison d'entrées, la sortie prévue conformément à la spécification.

Citons ici quelques exemples de l'industrie de la VLSI où la phase de vérification a permis de trouver des erreurs dans des circuits industriels.

Pendant la conception du système DPS 7000 de BULL [86], une phase de simulation a été utilisée avant toute fabrication des VLSI et toute réalisation physique du systéme pour valider la conception logique. Cette phase a permis de trouver 452 erreurs dans le système. Ces erreurs ont été introduites au cours du développement par suite de modifications. Dans le processeur UltraSPARC-I, développé par *SUN's SPARC's Technology*, des bogues très subtiles ont été décelées en utilisant un *model checker* [154]. Dans le processeur EWSD-CCS7E, développé par *SIEMENS*, 320 erreurs ont été trouvées pendant une phase de simulation [9]. Plus de 150 problèmes sérieux ont été détectés dans les ASIC's, et environ 35 problèmes dans les modules de la bibliothèque de composants autres que les ASIC's. Le quatrième exemple est le processeur PowerPC d'*IBM*. La phase de vérification a permis de découvrir 450 bogues dans le processeur PowerPC1, 480 bogues dans le processeur PowerPC2, et 600 bogues dans le processeur PowerPC3 [7]. Dans tous les cas que nous venons de citer, les erreurs ont été trouvées pendant les premières phases de la conception. Ce n'était pas le cas pour la fameuse bogue trouvée dans le processeur Pentium d'*Intel*. Il s'agissait d'une erreur de conception dans l'implémentation de l'algorithme de division à virgule flottante (FDIV). L'erreur n'a été découverte qu'après la commercialisation du processeur, ce qui a coûté à *Intel* plus de 400 millions de dollars [64]. De même pour le processeur Cyrix 6x86. Cyrix a récemment corrigé une erreur dans la version 2.6 du processeur. L'erreur est maintenant corrigée dans la version 2.7 mais pas avant que la version 2.6 soit déjà commercialisée. Cyrix refuse de déclarer la nature de l'erreur mais elle est reliée à l'écriture dans la mémoire cache d'après Microsoft [87].

Les méthodes de vérifications existantes, comme la simulation ou la preuve de tautologie par exemple, ont pour rôle de décider seulement si l'implémentation est correcte ou non; mais elles ne peuvent rien dire sur l'emplacement ou le type de l'erreur, s'il en existe. C'est le rôle du concepteur de localiser et de corriger l'erreur.

Dans le meilleur des cas, les outils de vérification peuvent seulement générer des contre-exemples, sous la forme de vecteurs de test qui mettent en évidence l'erreur dans le cas où l'implémentation n'est pas conforme à la spécification.

Il est important de clarifier ici la différence entre les fautes de fabrication et les erreurs de conception. Dans le premier cas, la structure du circuit est correcte: tous les composants sont correctement connectés et leurs types sont corrects. Si une erreur existe, c'est souvent à cause du fonctionnement défectueux d'un ou de plusieurs composants, ou à cause de la coupure de fils. La faute la plus connue dans cette classe est la faute de *collage à 1* ou *collage à 0*: la sortie d'un composant a toujours la valeur 1 ou la valeur 0 indépendamment des valeurs d'entrée. Ce type de faute survient pendant les phases de fabrication ou même après la production finale. Dans la littérature, il existe de nombreuses techniques de génération de vecteurs de test permettant la détection de ce genre de fautes [76, 3, 118, 126, 42]. Le but du diagnostic dans ce cas est de dépanner une implémentation réelle après sa fabrication. Le diagnostic repose souvent sur des bases de données contenant des couples observation-diagnostic. Ces couples sont normalement obtenus d'une implémentation correcte en utilisant les techniques de simulation de fautes.

Dans le cas d'erreurs de conception, seul le comportement correct du circuit est connu. L'erreur peut être due à l'utilisation d'un ou de plusieurs composants de type incorrect ou à de connexions déplacées. Les erreurs de conception surviennent souvent pendant les premières phases de conception, avant la fabrication. Dans cette thèse nous nous consacrons au diagnostic de ce dernier type d'erreurs, quand la non-équivalence entre une spécification et son implémentation est détectée par une phase de vérification.

La place du diagnostic dans le processus de conception est montrée dans la figure 1.1. Une spécification *SPEC* est donnée et validée par simulation et/ou par des techniques formelles. Après une étape de synthèse, soit manuelle soit automatique, soit les deux, une description initiale de l'implémentation est générée. Cette implémentation initiale est ensuite vérifiée par rapport à la spécification. Si une erreur est détectée, l'outil de preuve génère des contre-exemples que les concepteurs utilisent souvent pour simuler leur implémentation et pour essayer de trouver l'erreur manuellement et la corriger. L'implémentation initiale est modifiée et le cycle de vérification-diagnostic-correction est répété jusqu'à ce qu'une

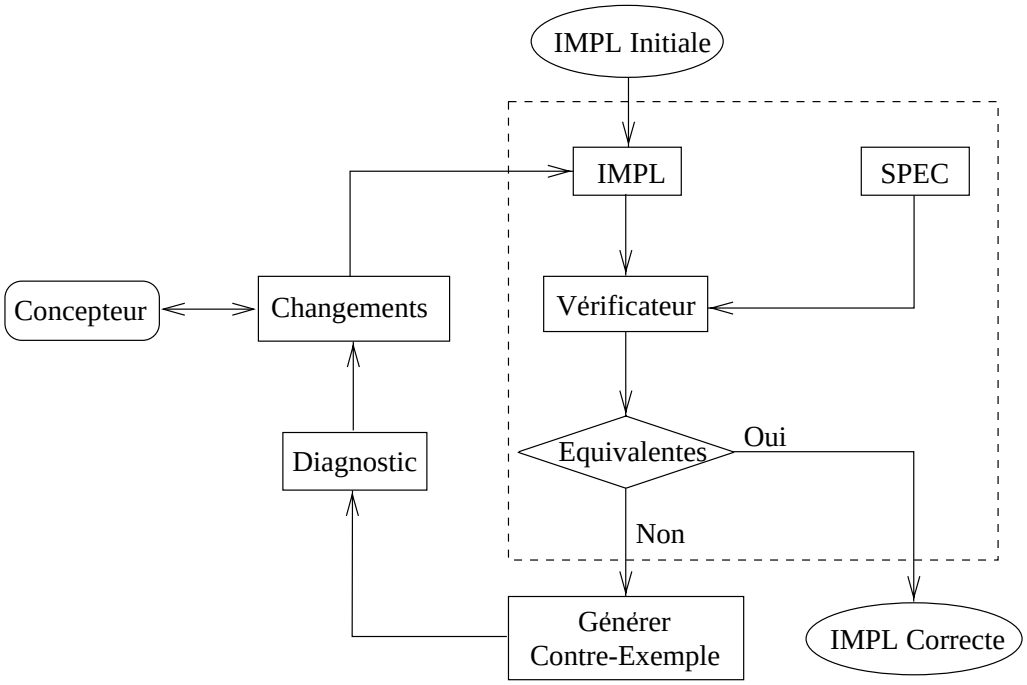

FIG. 1.1 - *Place du diagnostic dans le processus de conception*

implémentation correcte soit générée. Le processus de diagnostic manuel prend un temps très grand qui peut être égal, voire supérieur, au temps de conception lui-même. C'est ici qu'un système de diagnostic automatisé est nécessaire.

1.2 État de l'art

Si une grande littérature existe sur la détection des fautes de fabrication, très peu a été publié dans le domaine du diagnostic d'erreurs de conception.

Abadir [3] a classé les erreurs simples de conception en deux catégories: *i) les erreurs de portes*; un inverseur manquant ou excédentaire, une porte manquante ou excédentaire, ou le remplacement d'une porte par une autre; *ii) les erreurs de connexions*; une connexion manquante ou excédentaire, ou une connexion déplacée. Les erreurs de fausses connexions à des valeurs constantes représentent aussi une sorte d'erreurs de connexion, comme il nous l'a été rapporté par les ingénieurs de Thomson-TCS. Ces erreurs sont précisées dans la figure 1.2.

Odawara [123] a présenté un système nommé VERIFIER pour le diagnostic d'erreurs d'inverseurs. Ce système utilise les vecteurs de test incompatibles pour

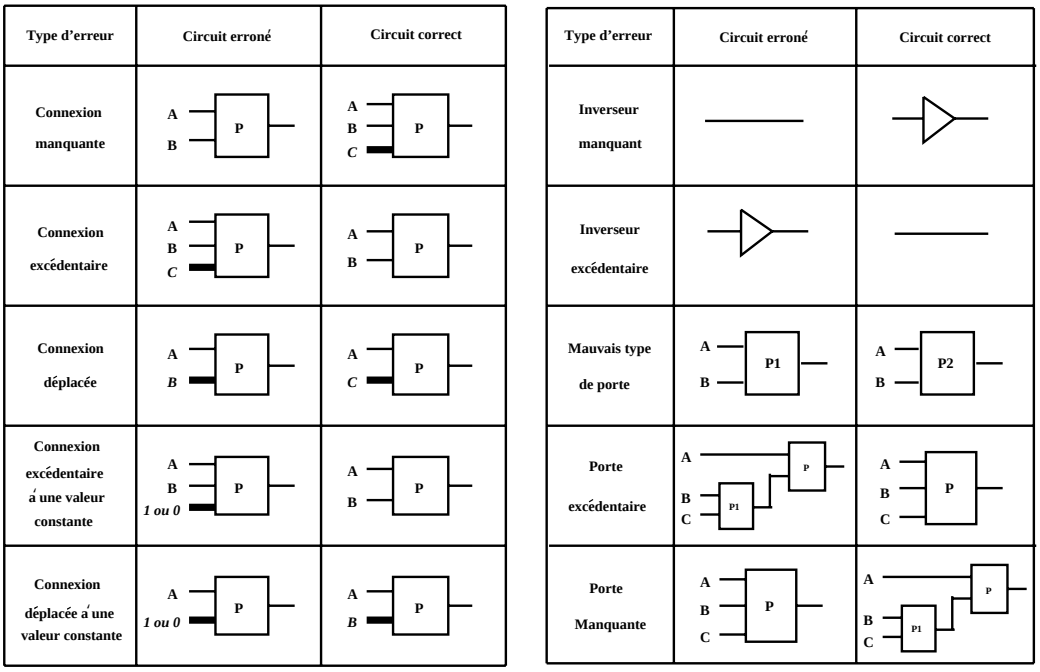

FIG. 1.2 - *Erreurs simples de conception*

trouver l'emplacement de l'erreur. Ces vecteurs de test incompatibles produisent aux sorties de l'implémentation des valeurs différentes des valeurs prévues selon la spécification. Cette méthode ne permet pas de déterminer précisément l'emplacement de l'erreur, mais de déterminer seulement la zone du circuit où l'erreur peut se trouver. L'implémentation est simulée sous l'application de vecteurs de test incompatibles, et elle est parcourue de ses sorties à ses entrées. Le parcours correspond aux chemins qui propagent les valeurs erronées. Le parcours s'arrête quand une porte où il est impossible de définir un chemin erroné est atteinte. L'erreur se trouve donc quelque part dans le cône d'influence de cette porte. En pratique, une telle porte est souvent trouvée près des sorties primaires d'où le parcours démarre, et donc une grande zone du circuit est susceptible de contenir l'erreur. Dans certaines expériences, la taille de la zone aurait représenté 37 % de la taille du circuit.

La technique discutée dans [129] est utilisée pour diagnostiquer les erreurs de porte. Elle détermine l'ensemble des portes qui peuvent être erronées en énumérant les vecteurs d'entrée qui détectent l'erreur. Un inconvénient essentiel de

cette méthode est que l'énumération des vecteurs d'entrée qui détectent l'erreur est pratiquement impossible pour les circuits ayant un grand nombre d'entrées.

Une autre méthode a été présentée dans [140], également pour trouver la zone du circuit contenant une erreur. Le circuit est divisé en sous-circuits, et chaque sous-circuit est vérifié séparément d'après la sous-fonction correspondante de la spécification. Cette technique exige la possibilité de trouver une correspondance entre les sous-circuits de l'implémentation et les sous-fonctions de la spécification, ce qui n'est pas toujours évident. Les résultats expérimentaux ont concerné seulement les erreurs d'inverseur, mais rien n'a été dit sur l'application à d'autres types d'erreurs, ni sur la taille des zones suspectes trouvées.

Madre [108] a présenté un système automatisé pour le diagnostic et la rectification de circuits logiques. Ce système est bâti autour du démonstrateur propositionnel de PRIAM, dont il exploite les capacités à résoudre les équations logiques. Pour chaque porte soupçonnée, le système calcule la fonction qui doit être générée à la sortie de cette porte afin de corriger le circuit, et il vérifie la possibilité de générer cette fonction à partir des fonctions d'entrée de cette porte. Cette technique est applicable seulement pour les erreurs de porte, mais pas pour les erreurs de connexion. Liaw [103] a apporté quelques modifications à cette méthode pour améliorer sa performance, mais les hypothèses essentielles sont restées les mêmes.

Une autre technique a été présentée dans [141]. En partant des vecteurs de test incompatibles, d'autres vecteurs appelés IPLDE (*Input Patterns for Locating Design Errors*) sont générés. Ces IPLDE sont ensuite utilisés pour le diagnostic. Les résultats expérimentaux montrent que le nombre de candidats d'erreur générés par cette méthode est grand, y compris pour de petits circuits. Cette technique a été modifiée pour être applicable aux cas comprenant des erreurs multiples [142]. Selon les résultats fournis, cette technique a permis de trouver les erreurs avec un taux de réussite variant entre 61.3 % et 100 % dans le cas où deux erreurs existaient, et entre 50.9 % et 100 % dans le cas où trois erreurs existaient. Le temps d'exécution augmente de façon exponentielle avec le nombre d'erreurs. Dans un des exemples, ce temps va de 26.7 secondes (cas de deux erreurs) à 4254.6 secondes (cas de trois erreurs). Un inconvénient de ces méthodes est qu'elles utilisent un grand nombre de IPLDE (généralement 100-200). L'existence de ces IPLDE n'est pas toujours garantie; or, un vecteur de test qui n'est pas un IPLDE est inutile

pour le diagnostic par ces méthodes.

Le système ACCORD a été présenté par Chung et Hajj dans [54, 55]. Ce système définit la zone d'erreur par l'intersection des cônes d'influence des sorties erronées. Une équation d'erreur est formée pour chaque ligne située dans cette zone. Ces équations d'erreur sont ensuite utilisées pour décider si telle ou telle ligne peut être un emplacement possible de l'erreur ou non. Ce système est basé sur l'utilisation de graphes de décision binaires (BDD), et beaucoup de jeux d'essai n'ont pas pu être testés à cause de l'immensité de la taille de la mémoire nécessaire pour construire les BDD. Il est bien connu que la taille des BDD augmente d'une manière exponentielle pour quelques circuits, comme les multiplicateurs d'entiers par exemple [31]. La méthode présentée dans [96] a aussi le même inconvénient.

Les méthodes présentées par Zhang [155, 156] utilisent aussi les BDD et les techniques de simulation de fautes. Un grand nombre de vecteurs de test sont nécessaires avant que l'erreur ne soit localisée (il a fallu, par exemple, 1087 vecteurs pour localiser une erreur dans un circuit de 546 portes).

Toutes les méthodes précédentes s'appliquent seulement aux circuits combinatoires. Dans [72], un algorithme est présenté pour le diagnostic d'une classe restreinte de circuits séquentiels, à savoir les circuits séquentiels sans boucles ou les circuits séquentiels ayant un comportement répétitif: le comportement du circuit se répète après un nombre fixe de cycles d'horloge. Les compteurs, les registres à décalage, et les convertisseurs parallèle-série sont autant d'exemples de circuits séquentiels appartenant à cette classe. L'algorithme de Fujita [72] peut aussi traiter les autres types de circuits séquentiels si la spécification et l'implémentation ont le même codage et le même nombre de variable d'états, ou si au moins une relation logique peut être trouvée entre les variables d'états des deux descriptions. Dans ce cas, le problème est réduit à un problème de diagnostic combinatoire.

A ce point l'utilisation des BDD mérite quelque discussion. Nous avons fait des essais préliminaires en 1993, en utilisant le paquetage de S. Höreth [84]. Ces essais ont montré que certains circuits combinatoires ne pouvaient pas être représentés, et que les vecteurs de test générés étaient moins efficaces que les vecteurs que nous utilisons actuellement [145]. Tout récemment, E. Sentovich a

publié une étude systématique des performances des paquetages de BDD les plus modernes [134]. Sur une station Sparc-20 avec 128 Mega Octets de mémoire, aucun des paquetages n'a eu assez de mémoire pour représenter 4 des circuits que notre système est capable de traiter sur une station beaucoup moins puissante. Ceci nous a conforté dans notre choix de ne pas utiliser de BDD pour le diagnostic.

1.3 Contribution de la thèse

Toutes les méthodes citées aux paragraphes précédents sont fondées soit sur l'emploi de vecteurs de test *binaires* [1], soit sur l'exploitation des techniques de représentation et de résolution d'équations logiques.

D'une part, l'emploi de vecteurs de test binaires est souvent très coûteux en temps à cause du grand nombre de vecteurs de test utilisés. D'autre part, les techniques de résolution d'équations logiques sont beaucoup plus complexes que les méthodes reposant sur l'utilisation de vecteurs de test.

La technique que nous avons développée est basée sur l'utilisation de vecteurs de test *ternaires* spécialement conçus pour le diagnostic. L'utilisation des vecteurs de test ternaires est exploitée dans [30], où Bryant a établi les fondements théoriques de la vérification formelle en utilisant la simulation selon la logique à trois valeurs.

Le problème du diagnostic, formulé de manière générale, est un problème NP-complet [73]. Afin de réduire la complexité du problème, nous employons une méthodologie de *diagnostic par hypothèses d'erreur* (voir la figure 1.3). Nous recherchons une erreur d'un type particulier, et un algorithme spécialisé dans ce type d'erreur est exécuté. Cet algorithme spécialisé est beaucoup plus efficace qu'un algorithme de diagnostic général, qui à notre connaissance n'a jamais été mis en œuvre dans un logiciel. Si une correction n'est pas possible selon l'hypothèse choisie, une autre hypothèse est choisie et un autre algorithme spécialisé dans le diagnostic de cette nouvelle hypothèse est appliqué, et ainsi de suite. Les hypothèses d'erreurs considérées dans cette thèse sont: les erreurs de composants

1. Seule la technique de Tomita [141, 142] fait intervenir des vecteurs contenant exactement une valeur ternaire 'X', qui se propage jusqu'aux sorties de la spécification et non de l'implémentation. Un tel vecteur peut ne pas exister, et dans ce cas la méthode échoue.

et les erreurs de connexions (connexion excédentaire, connexion manquante, et connexion déplacée). Nous distinguons dans ce manuscrit les différentes erreurs de connexions parce que les algorithmes utilisés pour chacune d'elles sont assez différents. Parmi ces types d'erreurs, certains sont rencontrés plus fréquemment que les autres. Dans l'étude statistique présentée dans [1], il a été montré que les erreurs de composants représentent 67% des erreurs de conception, suivies par les erreurs de connexions excédentaires qui représentent 17%, et les erreurs de connexions manquantes (9%). Dans notre système de diagnostic, ces types sont examinés successivement par ordre de probabilité décroissante,

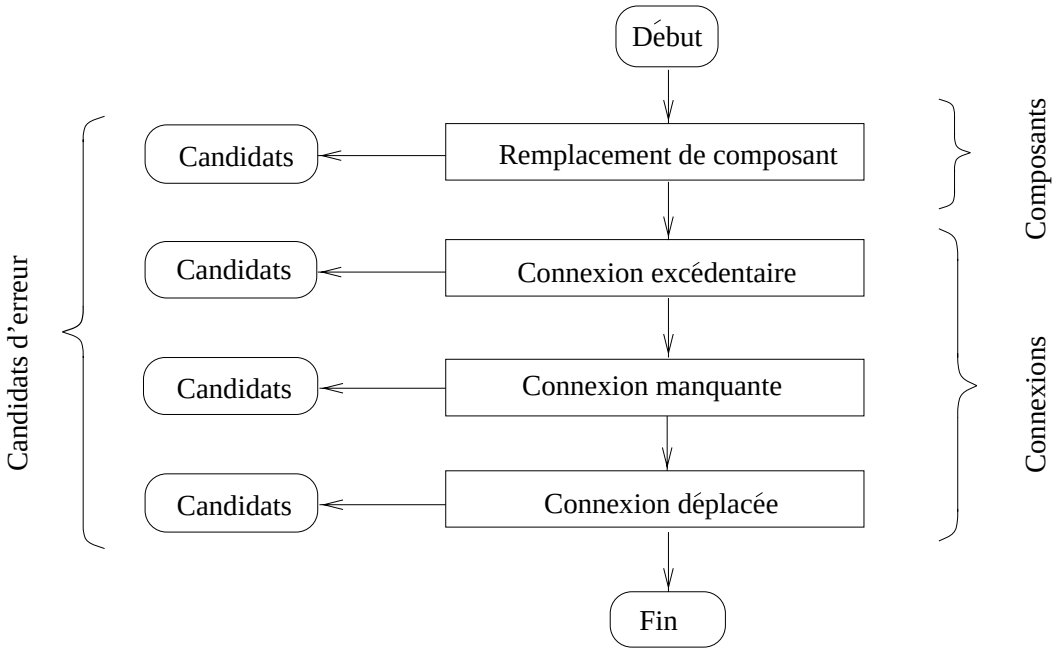

FIG. 1.3 - *Diagnostic par hypothèses d'erreur*

Notre système de diagnostic prend en entrée une forme interne issue de la compilation des descriptions, dans un langage standard, de la spécification et de l'implémentation. En pratique nous écrivons en VHDL (voir fiqure A.1 dans l'annexe A) mais tout ce qui suit s'appliquerait de la même manière à Verilog ou à tout autre langage de description de matériel. En particulier la syntaxe et la sémantique statique des descriptions ont déjà été vérifiées par le compilateur, ce qui permet de ne pas prendre en compte les erreurs que le compilateur sait détecter (entrée de composant non connectée, plusieurs sources pour une même

entrée ... etc).

Le logiciel de diagnostic est composé de trois modules essentiels qui coopèrent entre eux: un générateur de vecteurs de test ternaires, un simulateur selon la logique à trois valeurs, et un module de diagnostic. Le flux d'informations entre ces trois modules est présenté dans la figure 1.4.

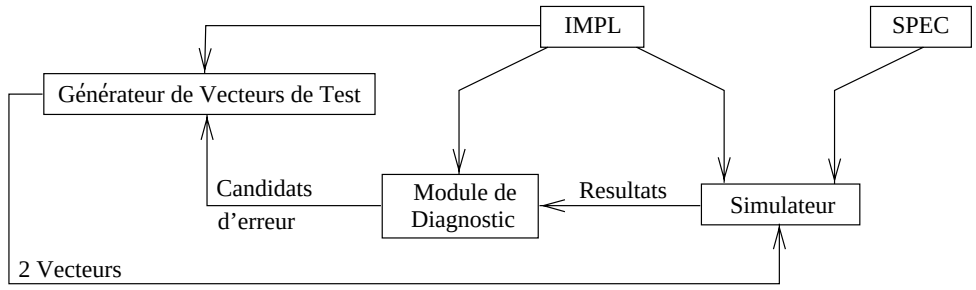

FIG. 1.4 - *Système de diagnostic*

Le générateur de vecteurs de test génère des vecteurs de test spécialisés dans le diagnostic. Ces vecteurs sont envoyés au simulateur qui simule l'implémentation et la spécification et envoie les résultats de la simulation au module de diagnostic. Ce module utilise ces résultats pour délimiter la zone du circuit où l'erreur est susceptible de se trouver. Le générateur de vecteurs de test génère des vecteurs supplémentaires pour les composants de cette zone, et la même opération est répétée jusqu'à ce que l'erreur soit trouvée.

Par rapport aux autres méthodes, les techniques présentées dans cette thèse leur sont supérieures sous quatre aspects:

1. Nos techniques ne sont pas limitées aux implémentations décrites par des portes simples de type AND, OR, NAND, etc. Des composants plus complexes peuvent également être utilisés.

2. Dans la plupart des cas, l'emplacement *exact* de l'erreur est trouvé. Dans d'autre cas, un petit nombre de candidats d'erreur est proposé. Si l'erreur est due au remplacement d'un composant par un autre, le composant erroné est localisé ainsi que le type du composant correct qui doit le remplacer. Si l'erreur est une erreur de connexion, la connexion fautive est identifiée.

3. Grâce aux vecteurs de test spécialement générés pour le diagnostic, l'erreur

est localisée après l'application d'un petit nombre de ces vecteurs (10-20 vecteurs pour un circuit de 2500 composants).

4. Nous présentons pour la première fois un outil de diagnostic des circuits séquentiels. La spécification et l'implémentation n'ont besoin d'avoir ni le même codage, ni le même nombre de variables d'états.

1.4 Plan de la thèse

Dans la suite de cette thèse, nous commençons par un chapitre traitant le problème de diagnostic d'erreurs de composants dans les circuits combinatoires (chapitre 2). Dans le chapitre suivant, nous nous attaquons au problème des erreurs de connexion, toujours dans les circuits combinatoires (chapitre 3). Enfin, après avoir donné les principes et les algorithmes propres au diagnostic combinatoire, nous abordons le problème du diagnostic dans les circuits séquentiels (chapitre 4).

Chacun de ces trois chapitres commence par une section consacrée aux définitions. Cette section présente les principes de base et les terminologies nécessaires pour comprendre le reste des chapitres. Quand besoin est, d'autres définitions sont présentées au fur et à mesure de notre discussion. Enfin, nous donnons notre conclusion et nos perspectives dans le chapitre 5.

L'annexe A donne un aperçu général de l'environnement de preuve PREVAIL dans lequel notre prototype de l'outil de diagnostic est intégré. L'annexe B est un bref manuel d'utilisation de cet environnement.

Chapitre 2

Diagnostic des erreurs de composants dans les circuits logiques combinatoires

Dans ce chapitre nous traitons les erreurs de composants. Une erreur de composant est due au remplacement d'un composant par un autre de type différent, parmi les composants disponibles dans une bibliothèque prédéfinie. Cette bibliothèque est un paramètre de l'outil de diagnostic; et d'une implémentation à une autre la bibliothèque peut être différente. En pratique, cette bibliothèque est donnée sous la forme d'un ensemble de descriptions VHDL (une entité et une architecture par composant) eventuellemnt regroupées dans un paquetage.

Le principe général du diagnostic repose sur la notion d'*espace de recherche*. Au départ n'importe quel composant est potentiellement erroné et donc l'espace de recherche contient tous les composants du circuit. A l'aide de vecteurs de test spécifiques, l'algorithme de diagnostic élimine progressivement des composants de cet espace de recherche jusqu'à ce que l'erreur soit localisée. L'objectif de cet algorithme consiste à déterminer non seulement le composant erroné mais aussi le type de composant qui doit être utilisé pour corriger le circuit.

2.1 Définitions et terminologie

Au travers de cette thèse, nous considérons une spécification SPEC et son implémentation *IMPL*, toutes deux au niveau booléen. La sortie de la spécification est dénotée $W = \{w_1, w_2, ...w_m\}$, et celle de l'implémentation est dénotée $Y = \{y_1, y_2, ...y_m\}$, où m est le nombre de sorties. Le style de description de la spécification n'est pas restreint tandis que l'implémentation est décrite par un réseau de composants. Chaque composant réalise une fonction logique combinatoire à une seule sortie. L'ensemble des types de composants qui peuvent être utilisés dans l'implémentation sont décrits dans une bibliothèque contenant un nombre arbitraire mais fini de ces types de composants. Ainsi, les portes élémentaires et complexes sont décrites par les types de composant AND2, AND3, AND4 (portes ET), OR2, OR3, ... etc. Dans un circuit, deux exemplaires de porte de type OR2, par exemple, sont deux composants distincts.

Nous retenons dans ce chapitre l'hypothèse d'une seule erreur de composant: si l'implémentation est erronée, l'erreur est due au remplacement d'*un et un seul* composant par un composant de type différent de la bibliothèque.

Définition 2.1 *: Espace de recherche.*
L'espace de recherche d'une implémentation est un sous-ensemble de ses composants et il contient le composant où l'erreur existe. ◇

Dans la suite il sera noté: *Espace-de-Recherche.*

Définition 2.2 *: Classes de remplacement.*
Deux types de composant $T_1 = Type(C_1)$ et $T_2 = Type(C_2)$ appartiennent à la même classe de remplacement, Classe-R, si et seulement si C_1 et C_2 ont le même nombre d'entrées et de sorties. ◇

Définition 2.3 *: Ensemble-P.*
A chaque composant $C \in$ Espace-de-Recherche est associé un ensemble nommé Ensemble-P(C) contenant tout type de composant T susceptible de corriger l'implémentation si C est remplacé par un autre composant de type T. ◇

Définition 2.4 *: Vecteurs discriminants.*
Un vecteur V est dit vecteur discriminant de deux composants C_1 et C_2, si

type(C_1) et type(C_2) appartiennent à la même classe de remplacement, et si la valeur binaire générée à la sortie de C_1 est le complément de la valeur générée à la sortie de C_2, quand V est appliqué aux entrées de C_1 et C_2. $\diamond$

Nous noterons ce prédicat: *Discriminant*(C_1, C_2).

Considérons par exemple que C_1 est une porte AND à trois entrées, et que C_2 est un multiplexeur ayant comme sélecteur a_0 et comme entrées a_1 et a_2. Le vecteur $(a_0a_1a_2) = (01X)$ génère 0 à la sortie de C_1, alors qu'il génère 1 à la sortie de C_2. Le vecteur $(01X)$ est donc un vecteur *Discriminant*(C_1, C_2).

Définition 2.5 *: Vecteurs non discriminants.*
Un vecteur V est dit vecteur non discriminants de deux composants C_1 et C_2, si type(C_1) et type(C_2) appartiennent à la même classe de remplacement, et si la valeur générée à la sortie de C_1 est égale à la valeur générée à la sortie de C_2, quand V est appliqué aux entrées de C_1 et C_2. $\diamond$

Nous noterons ce prédicat: *NonDiscriminant*(C_1, C_2).

Considérons encore la porte AND, C_1, et le multiplexeur C_2. Le vecteur $(a_0a_1a_2) = (00X)$ génère 0 à la sortie de C_1 et C_2. Le vecteur $(00X)$ est donc un vecteur *NonDiscriminant*(C_1, C_2).

a	$not(a)$
1	0
0	1
X	X

$not(a)$

$a \backslash b$	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

$a.b$

TAB. 2.1 - *Fonctions AND et NOT dans la logique à trois valeurs*

La Logique à trois valeurs :
Une variable dans la logique à trois valeurs peut avoir une valeur $v \in \mathcal{T}$, où

$\mathcal{T} = \{0, 1, X\}$. Le 0 et le 1 sont les valeurs booléennes conventionnelles, et X représente une valeur inconnue ou non spécifiée. La table 2.1 définit les fonctions *AND* et *NOT* dans la logique à trois valeurs. Toute autre fonction logique peut être dérivée à partir de ces deux fonctions.

Définition 2.6 *: Vecteur de test.*
Un vecteur de test d'un circuit à n entrées est un vecteur n-bit qui peut être binaire $\mathcal{B}^n$ ou ternaire $\mathcal{T}^n$, où

$\mathcal{B} = \{0, 1\}$ - *domaine binaire;*

$\mathcal{T} = \{0, 1, X\}$ - *domaine ternaire.* $\diamond$

Les vecteurs de test sont générés de telle façon qu'ils produisent une valeur binaire (0 ou 1) sur une ou plusieurs sorties de l'implémentation quand ils sont appliqués sur ses entrées.

Ordonnancement d'informations :
Bryant [30] a introduit le concept d'ordonnancement d'informations pour les valeurs logiques ternaires. L'ordre partiel $X < 0$ et $X < 1$ ordonne les valeurs ternaires selon leur teneur en information car X indique l'absence d'informations tandis que 0 et 1 représentent des valeurs complètement définies.

Une valeur a est dite *compatible* avec une valeur b si $a \leq b$ ou $b \leq a$. Sinon, a et b sont *incompatibles.*

Une valeur a est dite *plus faible* qu'une autre valeur b si $a < b$.

Le concept d'ordonnancement d'informations s'étend aussi aux vecteurs de valeurs ternaires.
Un vecteur de valeurs ternaires $A = \{a_1, a_2, \ldots a_n\}$ est compatible avec un autre vecteur $B = \{b_1, b_2, \ldots b_n\}$ si $\forall i$, a_i est compatible avec b_i.

Par exemple, les vecteurs '$X0XX$' et '$001X$' sont compatibles, ainsi que les

vecteurs '$X0X1$' et '$1X0X$'. Les vecteurs '$1X10$' et '$1X00$' ne sont pas compatibles car les troisièmes bits des deux vecteurs ne sont pas compatibles.

Un vecteur $A = \{a_1, a_2, \ldots a_n\}$ est dit plus faible qu'un autre vecteur $B = \{b_1, b_2, \ldots b_n\}$ si $(\exists i, (a_i < b_i)) \wedge (\forall j \neq i, (a_j \leq b_j))$.

Par exemple, '$1X0X$' < '$1X01$' et '$XX0X$' < '1001'.

Proposition 2.1 [30]*:*
Soit f une fonction logique de $\mathcal{T}^n$ dans $\mathcal{T}$, et soient A et B deux vecteurs de test, $A, B \in \mathcal{T}^n$. Si $A \leq B$, alors $f(A) \leq f(B)$. $\diamond$

Preuve :

La preuve de cette proposition provient directement de la propriété de monotonicité des fonctions logiques par rapport à l'ordonnancement d'informations [30], obtenue par étude de cas sur les deux operateurs de la table 2.1. $\diamond$

Définition 2.7 *: Vecteur de test minimal.*
Soit f une fonction logique de $\mathcal{T}^n$ dans $\mathcal{T}$. Un vecteur de test $V = \{v_1, v_2, ...v_n\} \in \mathcal{T}^n$ tel que $f(V)$ est une valeur binaire (0 ou 1) est appelé vecteur de test minimal par rapport à f s'il n'existe pas de vecteur $V' \in \mathcal{T}^n$ plus faible que V vérifiant $f(V')$ est binaire. $\diamond$

En d'autres termes, le remplacement de la valeur binaire d'un bit v_j, $1 \leq j \leq n$, par un X force $f(V)$ à la valeur X.

Un vecteur de test minimal, par rapport à une sortie d'un circuit réalisant une fonction f, est un vecteur correspondant à un des impliquants premiers de f.

Soit f une fonction de $\mathcal{T}^n$ dans $\mathcal{T}$. L'image réciproque de 1 par f, $(f^{-1}(1))$, est l'ensemble des vecteurs $V_1 \in \mathcal{T}^n$ tels que $f(V_1) = 1$ (*"on-set"* en anglais).

L'image réciproque de 0 par f, $(f^{-1}(0))$, est l'ensemble des vecteurs $V_0 \in \mathcal{T}^n$ tels que $f(V_0) = 0$ (*"off-set"* en anglais).

En l'absence de termes consacrés équivalents en français, nous utiliserons *"on-set"* et *"off-set"* dans ce mémoire.

Nous nous intéressons en particulier à la borne inférieure du *"on-set"* et du *"off-set"* par rapport à la teneur en information des vecteurs, c'est-à-dire aux vecteurs minimaux de ces deux ensembles.

Définition 2.8 *: "on-set" minimal et "off-set" minimal.*
Soit f une fonction de $\mathcal{T}^n$ dans $\mathcal{T}$. Le "on-set" (resp. "off-set") minimal de f est le sous-ensemble du "on-set" (resp. "off-set") de f constitué des seuls vecteurs minimaux par rapport à f. $\diamond$

Définition 2.9 *: Vecteur propageant D.*
Un vecteur $V = \{v_1, v_2, ...v_n\}$ est appelé vecteur propageant la valeur D dans un composant C à n entrées si $\exists i \mid v_i = D$ ou $\overline{D}$ et si la valeur D ou $\overline{D}$ est générée à la sortie de C quand V est appliqué à ses entrées. D est une valeur symbolique. $\diamond$

Par exemple, le vecteur $D11$ est un vecteur propageant D dans une porte AND à trois entrées, car la simulation symbolique d'une porte AND sous l'application de ce vecteur génère D à sa sortie. Ce vecteur propage aussi D dans une porte NAND, car il génère $\overline{D}$ à sa sortie. Pour un multiplexeur ayant un sélecteur a_0, et deux entrées a_1 et a_2, le vecteur '$a_0a_1a_2$' = '$1X\overline{D}$' propage D car il génère $\overline{D}$ à sa sortie, alors que le vecteur '$1D1$' ne propage pas D dans le multiplexeur car il génère 1 à sa sortie.

Définition 2.10 *: La frontière de D.*
Étant donnée une implémentation, la frontière de D est l'ensemble des composants dont une entrée au moins a la valeur D ou la valeur $\overline{D}$ et dont la sortie a la valeur X. $\diamond$

Définition 2.11 *: Un **circuit arborescent** est un circuit qui a un chemin unique de chaque entrée primaire jusqu'à chaque sortie primaire.* $\diamond$

La notion de *circuits arborescents* sera utilisée dans la section suivante pour démontrer un théorème sur ce type de circuits, qui sera ensuite étendu pour les circuits non-arborescents.

Définition 2.12 *: Le **cône d'influence** d'un composant C est l'ensemble de tous les composants et des entrées primaires qui se trouvent sur un chemin orienté des entrées primaires vers* C. $\diamond$

Nous le noterons $Cône(C)$.

La notion de *cône d'influence* est tout à fait centrale dans la suite de ce chapitre.

2.2 Génération des vecteurs de test

La méthode de génération des vecteurs de test que nous utilisons ici est une version modifiée de l'algorithme PODEM [76]. Cet algorithme est largement utilisé dans la littérature et il s'est montré très efficace dans la génération des vecteurs de test pour des circuits à haut niveau de complexité dans le cadre de la génération de vecteurs de test pour détécter les fautes à l'issue de la fabrication du ciruit. Le modèle de faute utilisé en test de fabrication est souvent le modèle de collage à 1 ou de collage à 0. Nous avons repris le schéma général de la méthode, toutefois notre algorithme est spécifique de la détection d'erreurs de conception. Dans ce chapitre, par exemple, ce que nous cherchons à mettre en évidence n'est pas une entrée ou une sortie de composant qui reste collée à une valeur constante mais une sortie de composant qui réalise une autre fonction.

La génération des vecteurs de test est effectuée en trois phases: la phase d'activation, la phase de justification et la phase de différenciation. Avant l'exécution de ces trois phases, les valeurs de tous les signaux dans le circuit sont non spécifiées, X.

2.2.1 La phase d'activation

La phase d'activation consiste à associer quelques valeurs à quelques nœuds dans le circuit. Si par exemple le vecteur est destiné à détecter (resp. ne pas détecter) le remplacement d'un composant C_1 par un autre composant C_2, nous associons les valeurs correspondantes à un des $Discriminant(C_1, C_2)$ (resp. $NonDiscriminant(C_1, C_2)$) aux entrées du composant C_1.

Pour accélérer le processus de génération et pour éviter autant que possible le *"backtracking"* qui pourrait se produire à cause de la détection tardive de conflits entre les valeurs associées, les vecteurs $Discriminant(C_1, C_2)$ (resp. $NonDiscriminant(C_1, C_2)$) possibles sont ordonnés selon une fonction de coût exprimant la difficulté d'associer chacun de ces vecteurs aux entrées de C_1. Le vec-

teur *Discrimiunant*(C_1, C_2) (resp. *NonDiscriminant*(C_1, C_2)) ayant le coût le plus petit est tenté d'abord. Si un conflit se produit, le vecteur *Discriminant*(C_1, C_2) (resp. *NonDiscriminant*(C_1, C_2)) suivant est tenté, et ainsi de suite. L'emploi de fonctions de coût dans la génération des vecteurs de test est utilisé dans le système *CONTEST* [6]. Cette méthode a permis d'accélérer considérablement la génération des vecteurs de test, comparée à celle d'autres systèmes tels que *STG* [110].

Les fonctions de coût présentées dans [6] s'appliquent seulement aux circuits contenant des portes simples (AND/NAND/OR/NOR/NOT). Nous donnons ici une définition plus générale des fonctions de coût, qui s'applique également aux circuits contenant des composants combinatoires plus complexes.

Le coût de justification exprime la difficulté de mettre une certaine valeur v sur un nœud n dans le circuit. Si n a déjà la valeur v, alors le coût est égal à 0. Sinon, le coût depend de la nature de n. Si n est une entrée d'un composant, le coût est égal au coût pour mettre la valeur v sur le nœud duquel n provient (une entrée primaire ou une sortie d'un composant). Si n est une entrée primaire, alors il suffit d'associer la valeur v à cette entrée, et dans ce cas le coût est égale à 1. Par ailleurs, si n est la sortie d'un composant, il faut trouver un vecteur d'entrées de ce composant qui génère v à sa sortie, et le coût dans ce cas est égal à la somme des coûts associés aux éléments du vecteur. Nous donnons dans la suite une définition formelle du coût de justification.

Dans tout ce qui suit, un nœud est soit une entrée primaire, soit la sortie ou l'entrée d'un composant du circuit.

Définition 2.13 *: La fonction de coût de justification.*
Soit V un ensemble de valeurs ternaires $\{v_1, v_2, ...v_m\}$ compatible avec l'ensemble de valeurs courantes $VC = \{vc_1, vc_2, ...vc_m\}$ d'un ensemble de nœuds $N = \{n_1, n_2, ...n_m\}$. Si la valeur v_i doit être associée au nœud n_i, $1 \leq i \leq m$, alors la fonction de coût CoûtJustification(N, V) est définie comme suit:

$$CoûtJustification(N, V) = \sum_{i=1}^{m} coût(n_i, v_i) \mid v_i \neq X$$

où $coût(n_i, v_i)$ est calculé comme suit:

---

if $vc_i = v_i$ ***then***

coût$(n_i, v_i) = 0$;
elseif n_i *est une entrée d'un composant* ***then***
m = *la sortie d'un composant, ou l'entrées primaire de laquelle* n_i *provient;*
coût$(n_i, v_i) =$ *coût*(m, v_i);
elseif n_i *est une entrée primaire* ***then***
coût$(n_i, v_i) = 1$;
else
C = *le composant dont la sortie est* n_i;
Valeurs = valeurs courantes des entrées de C;
if $v_i = 1$ ***then***
E = *"on-set" minimal de* C;
else
E = *"off-set" minimal de* C;
endif
$P = \{P_j \mid P_j \in \mathrm{E} \wedge P_j$ *compatible avec Valeurs*$\}$;
if $P = \phi$ ***then***
coût$(n_i, v_i) = \infty$
else
coût$(n_i, v_i) = min_j$ $\{$*CoûtJustification*$(entrées(C), P_j)\}$, $P_j \in P$
endif
endif $\diamond$

2.2.2 La phase de justification

La phase de justification consiste à associer des valeurs particulières à quelques entrées primaires afin de générer aux entrées du composant C_1 les valeurs déterminées pendant la phase d'activation.

Pour justifier une valeur v à un nœud n dans le circuit, il y a deux possibilités:

1. Soit n est une entrée primaire, et dans ce cas la justification est immédiate: nous mettons la valeur de cette entrée à v.

2. Soit n est la sortie d'un composant C, et dans ce cas il faut fixer une ou plusieurs entrées de C à des valeurs qui imposent v en sortie. S'il existe

plusieurs associations possibles, la plus facile à justifier est tentée d'abord (selon la fonction de coût correspondante). Plus la fonction de coût est petite, plus la justification est facile. La même démarche est ensuite répétée aux entrées de C, et ainsi de suite jusqu'à ce que les entrées primaires soient atteintes.

2.2.3 La phase de différenciation

Dans la phase de différenciation la valeur de sortie du composant C_1 doit être propagée jusqu'à une ou plusieurs sorties primaires. La valeur de sortie de ce composant est remplacée par la valeur D, et le circuit est simulé sous l'application des valeurs des entrées primaires obtenues dans la phase précédente. La frontière de D est ensuite calculée.

S'il y a plusieurs composants par lesquels la valeur D peut se propager (c'est-à-dire si la frontière de D contient plus d'un composant), alors le composant ayant la *fonction de coût de propagation* la plus petite est tenté d'abord. Cette fonction exprime la difficulté de propager une valeur d'un nœud à au moins une sortie primaire.

Si le nœud n dont la valeur D est à propager est une sortie primaire, alors la valeur D existe déjà sur une sortie primaire et le coût de propagation est égal à 0.

Si le nœud n est une entrée d'un composant C, les autres entrées de C doivent être associées à des valeurs qui propagent D à la sortie de C. Le coût de propagation est donc égal au coût de justification des entrées de C à un vecteur propageant D dans C. Ce vecteur doit être compatible avec les valeurs courantes des entrées de C. Par exemple, si les valeurs courantes des entrées d'une porte AND sont $(DX11)$, le seul vecteur de test propageant D et compatible avec $(DX11)$ est $(D111)$. Donc le coût de propagation de D à partir de l'entrée de cette porte jusqu'à une sortie primaire est égal au coût de fixation de la valeur de la deuxième entrée à 1 plus le coût de propagation de D de la sortie de cette porte jusqu'à une sortie primaire.

Si le nœud n est une patte de sortance (*"fanout stem"*), alors la propagation de D peut être effectuée via n'importe laquelle de ses branches de sortance. La branche ayant le coût le moins élevé est choisie.

Après la propagation de la valeur D jusqu'à la sortie du composant sélectionné, une nouvelle frontière de D est générée. La même démarche est ensuite répétée jusqu'à ce que la valeur D atteigne une sortie primaire.

La définition 2.14 ci-après présente une méthode pour calculer la fonction de coût de propagation.

Définition 2.14 *: La fonction de coût de propagation.*
La fonction de coût de propagation CoûtPropagation(n) d'une valeur D d'un nœud n dans l'implémentation jusqu'à une sortie primaire est calculée comme suit:

---

if *n est une sortie primaire* ***then***
  $CoûtPropagation(n) = 0;$
elseif *n est une entrée d'un composant C ayant la sortie c* ***then***
  Valeurs = *valeurs courantes des entrées de C;*
  $P = \{P_j \mid P_j$ *est un vecteur minimal propageant D dans le composant* $C \wedge P_j$ *est compatible avec* Valeurs$\}$;
  if $P = \phi$ ***then***
    $CoûtPropagation(n) = \infty$
  else
    $CoûtPropagation(n) = CoûtPropagation(c) +$
      $min_j\ CoûtJustification(entrées(C), P_j),\ P_j \in P;$
  endif
elseif *n est une patte de sortance à m branches* ***then***
  $CoûtPropagation(n) = min_{1 \leq j \leq m}\ CoûtPropagation(i_j);$
  où i_j *est la* $j^{ème}$ *branche de sortance de n;*
endif $\diamond$

2.3 L'algorithme de diagnostic

Le diagnostic est fait en utilisant une technique de rétro-propagation. La spécification et l'implémentation sont simulées sous l'application de vecteurs de test ternaires, et leurs sorties sont comparées. Les composants de l'implémentation sont ensuite analysés, en commençant par les composants connectés aux sorties

primaires et en poursuivant vers les entrées primaires. En commençant par une sortie S et sous l'application d'un vecteur de test VT, à chaque composant analysé est associé un des deux ensembles suivants: un ensemble de types de composants qui peuvent remplacer C pour corriger la valeur de S si elle est erronée; ou un ensemble de types de composants qui ne peuvent pas remplacer C, faute de quoi la valeur correcte de S changera. Ces deux ensembles sont appelés respectivement l'*ensemble des types possibles*, *Types-Possibles(C,VT)*, et l'*ensemble des types impossibles*, *Types-Impossibles(C,VT)*.

Définition 2.15 *: Valeur requise à la sortie d'un composant.*
Soit $Y_m(C, z, VT)$ la valeur générée à la sortie de l'implémentation sous l'application d'un vecteur de test VT, si la valeur courante à la sortie d'un composant C, $VCourante(C, VT)$, est remplacée par une autre valeur z.
La valeur requise à la sortie d'un composant C, $VRequise(C, VT)$, sous l'application du vecteur de test VT, est la valeur qui satisfait $Y_m(C, VRequise(C, VT), VT) = W(VT)$. $\diamond$

Proposition 2.2 *:*
Un type de composant T est mis dans l'ensemble Types-Possibles(C,VT) si le remplacement de C par un autre composant de type T, sous l'application d'un vecteur de test VT qui détecte l'erreur, génère la valeur requise $VRequise(C, VT)$ à la sortie de C. $\diamond$

Proposition 2.3 *:*
Un type de composant T est mis dans l'ensemble Types-Impossibles(C,VT) si le remplacement de C par un autre composant de type T, sous l'application d'un vecteur de test VT, génère à la sortie de C une valeur différente de la valeur requise $VRequise(C, VT)$. $\diamond$

Il est à noter que le remplacement de C par un autre composant de type $T \in$ *Types-Possibles(C,VT)* corrigera la valeur de la sortie quand le vecteur VT est appliqué aux entrées primaires. La notion d'*Ensemble-P(C)* donnée par la définition 2.2 n'est pas liée à un seul vecteur de test VT. Un Ensmble-P(C) est égal à l'intersection de tous les ensembles *Types-Possibles* moins l'union de tous

les ensembles *Types-Impossibles* génères pour le composant C après l'application de plusieurs vecteurs de test.

Les vecteurs de test que nous utilisons dans notre méthode sont générés de telle façon qu'ils produisent une valeur binaire, (0 ou 1), sur au moins une sortie de l'*implémentation*, tandis que les autres sorties pourraient avoir la valeur X. Le diagnostic est fait à partir des sorties ayant des valeurs binaires.

Si l'erreur est détectée sur une sortie y_i sous l'application d'un vecteur de test VT, deux possibilités peuvent se produire: soit $w_i(VT) = \overline{y}_i(VT)$, soit $w_i(VT) = X$.

Premier cas : $w_i(VT) = \overline{y}_i(VT)$.

Le diagnostic commence au composant C dont la sortie est y_i. Un moyen de corriger la valeur de cette sortie est de remplacer C par un autre composant de type différent. Cette étape génère l'ensemble *Types-Possibles(C,VT)*. Mais ce n'est pas le seul moyen de rendre $y_i(VT)$ égal à $w_i(VT)$ car le même résultat peut être obtenu en gardant le composant C invariable et en *inversant* la valeur d'au moins une de ces entrées e. Il est impossible, selon la proposition 2.1, d'inverser la valeur de sortie de C sans inverser la valeur d'au moins une entrée. Si une telle entrée e existe, l'algorithme de diagnostic examine aussi le composant duquel e provient. L'analyse continue de la même façon jusqu'à ce que les entrées primaires soient atteintes.

Exemple 2.1 *:*

Le circuit présenté dans la figure 2.1-a est une implémentation erronée de la spécification donnée par $w = e(d(b + c) + f)$. La bibliothèque de composants utilisée pour la conception comporte les types {AND2,NAND2,AO21,OA21,MUX2}. Les fonctions que ces composants réalisent sont données comme suit:

Composant	*Entrées*	*Fonction de sortie*
AND2	$\{a, b\}$	$a.b$
NAND2	$\{a, b\}$	$\overline{a.b}$
AO21	$\{a, b, c\}$	$a.b + c$
OA21	$\{a, b, c\}$	$(a + b).c$
MUX2	$\{s, a, b\}$	$\overline{s}.a + s.b$

L'erreur est causée par le composant g qui devrait être un OA21 à la place de AO21. En fait, il nous a été rapporté par les ingénieurs de Thomson-TCS que ce genre d'erreurs est commun à cause de la ressemblance entre les noms de composants.

Sous l'application du vecteur $abcdef = X00110$*, la sortie de la spécification est* 0*. L'implémentation, comme nous pouvons la voir sur la figure 2.1-a, génère* 1 *à sa sortie. Un moyen de corriger la valeur de la sortie est de remplacer le composant y par un autre composant ayant un type dans la classe de remplacement de AO21. Nous avons donc le choix entre MUX2 et OA21. Le remplacement de y par un MUX2 ou par un OA21 corrigera la valeur de la sortie, et donc Types-Possibles*$(y, X00110) = \{MUX2, OA21\}$*. Un autre moyen de corriger la sortie est d'inverser les valeurs de quelques entrées du composant y. La sortie d'un composant AO21 est égale à* 0 *si ses entrées sont* $0X0$ *ou* $X00$*. Alors, soit* $i = 0$ *soit* $j = 0$ *corrige la sortie, et les composants i et j sont aussi examinés. Le résultat de cet examen est Types-Possibles*$(i, X00110) = \{\}$*, et Types-Possibles*$(j, X00110) = \{\}$*. La poursuite de l'analyse passe aussi par le composant g avec comme résultat Types-Possibles*$(g, X00110) = \{OA21\}$*. Les lignes en trait épais dans la figure 2.1-a correspondent aux chemins suivis par l'algorithme de diagnostic par rétro-propagation.* $\diamond$

Deuxième cas : $w_i(VT) =$ **'X'.**

Dans ce cas, la sortie y_i de l'implémentation a une valeur binaire, tandis que la sortie correspondante de la spécification w_i a comme valeur X, due à l'existence de X sur une ou plusieurs entrées primaires. La raison pour laquelle X n'apparaît pas sur la sortie de l'implémentation est que sa propagation est bloquée par un composant ayant X à ses entrées et une valeur binaire à sa sortie.

Le diagnostic est fait comme dans le cas précédent mais l'analyse commence à partir des composants qui empêchent la propagation de la valeur 'X', et non plus à partir de la sortie primaire. Ceci permet d'accélérer grandement le processus de diagnostic car seul un petit nombre de composants sont examinés. Le but est de changer les valeurs binaires existant aux sorties de ces composants pour qu'elles

deviennent X. Trois moyens existent pour ce faire:

1. Affaiblir la valeur d'une entrée j de C, ce qui signifie propager un X bloqué jusqu'à ce qu'il parvienne à j. Nous ne considérerons pas ce moyen car il est obtenu automatiquement par le moyen numéro 2 ou 3.

2. Inverser la valeur d'une entrée k de C ayant une valeur binaire. L'analyse est alors faite comme dans le premier cas où $w_i(VT) = \overline{y}_i(VT)$.

3. Remplacer le composant qui bloque la propagation de X par un autre composant d'un autre type tel que X est généré à sa sortie.

Exemple 2.2 *:*
Le circuit de la figure 2.1-b est le même que celui de l'exemple précédent, mais dans ce cas il réalise la spécification donnée par $w = e(ad + bc + bd + f)$. *L'implémentation est erronée à cause du composant i qui aurait dû être un OA21. Sous l'application du vecteur* $X00110$ *l'implémentation génère* 1 *à sa sortie tandis que la spécification génère* X. *Ici, le diagnostic ne commence pas à partir de la sortie primaire y, mais à partir des composants qui bloquent la propagation de* X. *Dans notre exemple, il n'y a que le composant i qui bloque la propagation de* X. *La valeur* X *à l'entrée de ce composant peut être propagée à sa sortie si i est remplacé par un autre composant de type OA21. Il n'y a pas d'autre moyen de faire propager la valeur* X. $\diamond$

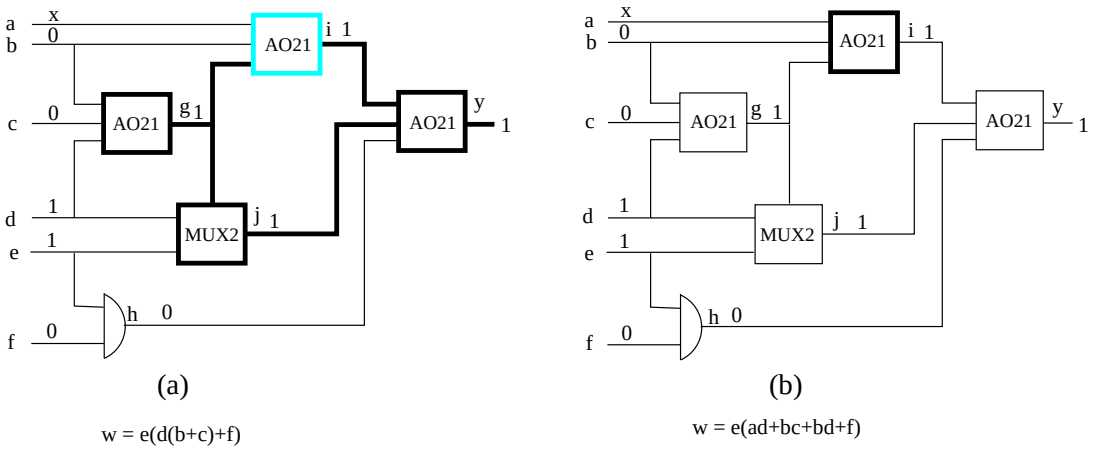

FIG. 2.1 - *Deux exemples du mode de calcul de Types-Possibles*

Nous expliquerons par la suite comment les chemins par lesquels l'analyse se poursuit sont trouvés (c'est-à-dire les entrées d'un composant C ou les nœuds dans le circuit qui doivent changer leurs valeurs pour que $VRequise(C, VT)$ soit générée à la sortie de C).

Définition 2.16 *: Frontière d'un ensemble de composants.*
Un composant C_2 est dit successeur *d'un composant C_1, ($C_1 \neq C_2$), si la sortie de C_1 est connectée à une (ou plusieurs) entrée de C_2. Soit R un ensemble de composants de l'implémentation. La* frontière *de R sous l'application d'un vecteur de test VT, Frontière(R, VT), est un sous-ensemble de R contenant les composants n'ayant pas de successeurs dans R, et dont la valeur de la sortie est '0' ou '1'.* $\diamond$

Proposition 2.4 *: Les nœuds modifiables.*
Soit VT un vecteur de test qui détecte l'erreur sur une sortie y_i de l'implémentation. Soit C le composant en cours d'examen, dont les entrées sont $\{e_1, e_2, ...e_n\}$. Les valeurs des entrées de C sont $\{v_1, v_2, ...v_n\}$ sous l'application de VT aux entrées primaires du circuit.

Soit $z = VRequise(C, VT)$ la valeur requise à la sortie de C afin de corriger y_k.

$$Soit\ Set = \begin{cases} \text{“on-set”}\ minimal\ de\ C & si\ z = 1 \\ \text{“off-set”}\ minimal\ de\ C & si\ z = 0 \\ \text{“X-set”}\ minimal\ de\ C & si\ z = X \end{cases}$$

où le “X-set” minimal *de C est défini comme suit:*

$\{P = \{p_1, p_2, ...p_n\} \mid P$ *est un vecteur propageant D depuis l'entrée e_j de C, et* $v_j = X\}$, $1 \leq j \leq n$;

Pour chaque $P = \{p_1, p_2, ...p_n\} \in$ Set
Cône Commun $= \bigcap_{(v_k \neq p_k) \wedge (p_k \neq X)} Cône(e_k)$, $1 \leq j, k \leq n$
Si $\exists j \mid (v_j = \overline{p_j}) \wedge$ *(Cône Commun* $\neq \phi$*)*, $1 \leq j, k \leq n$
alors *la valeur requise z peut être générée à la sortie de C si et seulement si la valeur de sortie de n'importe quel composant dans Frontière(Cône Commun,VT) est inversée.*

Nœuds-Modifiables(C) = Frontière(Cône Commun,VT) $\diamond$

Preuve :

- $z = 1$ **ou** 0 :

  Dans ce cas, la valeur de sortie du composant C doit être inversée. Selon la proposition 2.1, la valeur d'au moins une entrée de C doit être inversée. Considérons $z = 1$ (l'argument est symétrique pour $z = 0$). If faut appliquer aux entrées de C un vecteur de son *"on-set"*.

  Dans la proposition, la condition "$\exists j \mid v_j = \overline{p_j}$" caractérise l'existence d'une entrée à inverser, pour obtenir un vecteur du *"on-set"*.

  Toutefois, il se peut que l'inversion d'une seule entrée ne produise pas un vecteur du *"on-set"*. S'il faut en inverser plusieurs, sous l'hypothèse d'une seule erreur, ceci n'est possible que si elles sont toutes influencées par un nœud commun dans le circuit. Les nœuds qui affectent plusieurs entrées de C simultanément existent dans le cône d'influence commun de ces entrées. Donc, la condition *Cône Commun* $\neq \phi$ garantit l'existence d'un tel nœud.

  Pour générer la valeur 1 à la sortie de C, les valeurs de ses entrées doivent être *plus fortes* ou *égales* à un vecteur P du *"on-set"* minimal. Si le $j^{ème}$ bit du vecteur P, p_j est égal à X, la valeur de la $j^{ème}$ entrée de C, v_j, peut être fixée à n'importe quelle valeur sans affecter la sortie de C. Donc, il n'y a pas besoin de changer la valeur de la $j^{ème}$ entrée de C. C'est pour cette raison que l'intersection faite pour calculer *Cône Commun*, est faite seulement sur les entrées de C où $p_k \neq x$. (Voir l'indice $p_k \neq x$ de $\bigcap$).

  Cependant, si p_j est une valeur binaire, v_j doit être changée en p_j. C'est pour cette raison que l'intersection faite pour calculer *Cône Commun*, est faite sur les entrées de C où $v_k \neq p_k$. (Voir l'indice $v_k \neq p_k$ de $\bigcap$).

- $z = X$:

  Afin de générer X à la sortie d'un composant C, l'une au moins des entrées de C doit valoir X, et les autres entrées doivent propager ce X. Si $v_j = p_j = X$, alors pour générer $z = X$ les entrées de C doivent être plus fortes ou égales à un vecteur P qui propage la valeur de e_j. Un tel vecteur existe dans l'ensemble *"X-set"* le plus faible. La preuve est ensuite identique à celle du cas précédent. $\diamond$

La proposition 2.4 dirige le processus de rétro-propagation. Comme nous l'avons dit auparavant, pour changer la sortie d'un composant, soit nous changeons le type du composant, soit nous changeons les valeurs de certains nœuds affectant les entrées du composant. Dans ce dernier cas, la rétro-propagation est faite à travers les nœuds modifiables, qui sont déterminés par la proposition 2.4.

Parfois, l'erreur n'est pas détectée sur une certaine sortie y_i sous l'application d'un vecteur de test VT, (c'est-à-dire $y_i(VT) = w_i(VT)$), et donc $y_i(VT)$ ne doit pas changer sa valeur après la correction du circuit. Le circuit est alors parcouru en rétro-propagation à partir de y_i comme précédemment. Mais dans ce cas, nous extrayons pour chaque composant analysé C les types de composant qui ne peuvent pas remplacer C et nous les mettons dans *Types-Impossibles(C,VT)*. Nous déterminons aussi les entrées de C qui doivent conserver leur valeur, et la rétro-propagation se poursuit à travers ces entrées que nous appelons *entrées fixes*. La proposition suivante est utilisée pour déterminer les entrées fixes.

Proposition 2.5 *: Entrées Fixes d'un composant.*
Soit $V = \{v_1, v_2, ...v_n\}$ *le vecteur de valeurs des entrées* $I = \{i_1, i_2, ...i_n\}$ *d'un composant* C *dans l'implémentation sous l'application d'un vecteur de test* VT. *Et soit* z *la valeur courante à la sortie de* C. *Si* z *doit être inchangée, alors*

Pour chaque vecteur $P = \{p_1, p_2, ...p_n\} \in$ Set, P *compatible avec* V, *où*

$$Set = \begin{cases} \text{“on-set”} \ \textit{minimal de } C \ \textit{si } z = 1 \\ \text{“off-set”} \ \textit{minimal de } C \ \textit{si } z = 0 \end{cases}$$

Les entrées fixes de C *sont données par:*

$$\text{Entrées fixes} = \{i_j \mid (i_j \in I) \wedge (p_j = v_j \neq X)\}$$

Preuve :

Puisque le but est de garder la valeur *binaire* de la sortie d'un certain composant inchangée, le cas où $z = X$ ne sera pas envisagé.

Si la sortie d'un composant C est à fixer à 1 (resp. 0), V doit être plus fort ou égal à un vecteur P dans le *“on-set” minimal* (resp. *“off-set” minimal*)

de C. Il existe deux possibilités pour les valeurs des bits de P. Soit $p_j = X$, ce qui signifie que la valeur de l'entrée i_j n'a pas d'influence sur la sortie de C sous l'application de P. Il n'est donc pas nécessaire de garder la valeur de i_j inchangée, et i_j n'est pas une entrée fixe. Soit $p_j = 1$ ou 0. Puisque P est un vecteur du *"on-set" minimal* (resp. *"off-set" minimal*), alors la valeur de i_j est indispensable pour générer la sortie de C, et i_j est donc une entrée fixe. $\diamond$

L'algorithme de diagnostic par rétro-propagation résulte directement des définitions et des propositions données ci-dessus. Nous présentons ici le squelette de cet algorithme. La procédure *diagnostic-combinatoire* prend comme entrées le vecteur de test VT, l'espace de recherche *Espace-de-Recherche*, un tableau contenant l'*Ensemble-P(C)* de chaque composant $C \in$ *Espace-de-Recherche* et les valeurs de sorties de l'implémentation Y et de la spécification W sous l'application de VT. Au départ, tous les composants du circuit sont susceptibles d'être erronés, et donc *Espace-de-Recherche* contient tous les composants du circuit. Cet espace de recherche est ensuite diminué progressivement au fur et à mesure que l'algorithme de diagnostic est appliqué.

Dans cet algorithme, l'implémentation et la spécification sont simulées sous l'application d'un vecteur de test donné VT, et leurs sorties y_i, w_i $(1 \leq i \leq m)$ sont comparées. Si $\exists i$, $y_i \neq w_i$, le vecteur de test est classé comme étant *un Vecteur détectant l'erreur, VDE,* pour la sortie y_i. Sinon, il est classé comme étant *un vecteur ne détectant pas l'erreur, VNE,* pour y_i. Le circuit est ensuite parcouru en commençant par ses sorties et en se poursuivant vers les entrées primaires. Cela est fait par la procédure récursive *analyser*. Le résultat de cette procédure est d'associer à chaque composant parcouru un ensemble *Types-Possibles(C,VT)* et un ensemble *Types-Impossibles(C,VT)* qui sont utilisés pour mettre à jour l'*Ensemble-P(C)*.

---

procedure diagnostic-combinatoire(*VT, Espace-de-Recherche, Tableau-Ensembles-P, Y, W*);
begin
  for tout composant C_{y_i} dont la sortie est $y_i \neq X$ **do**
    Nouvel Espace de Recherche $= \phi$;
    if $w_i = X$ **then**
      Composants = Composants bloquant la propagation de X;

```
        else
            Composants = C_{y_i};
        endif;
        if y_i ≠ w_i then
            Type-Vect = VDE;
        else
            Type-Vect = VNE;
        endif;
        for tout C ∈ Composants do
            analyser(Type-Vect,C);
            if Type-Vect = VDE then
                Espace de Recherche = Nouvel Espace de Recherche;
            endif;
        endfor;
    endfor;
end.
```

```
procedure analyser(Type-Vect,C);
begin
    if C est une entrée primaire then
        exit
    endif;
    if Type-Vect = VDE then
        if C ∈ Espace de Recherche then
            Types = Types-Possibles(C, VT);
            mettre-à-jour(Type-Vect,C, Types);
        endif
        for tout nœud i ∈ Nœuds-Modifiables(C) do        % proposition 2.4
            analyser(Type-Vect,i);
        endfor
    else
        Types = Types-Impossibles(C, VT);
        mettre-à-jour(Type-Vect,C, Types);
        for toute entrée i ∈ Entrées-Fixes(C) do        % proposition 2.5
            analyser(Type-Vect,i);
        endfor;
    endif;
end.
```

```
procedure mettre-à-jour(Type-Vect,C, Types);
begin
   if Type-Vect = VDE then
      Ensemble-P(C) = Ensemble-P(C) ∩ Types;
      if Ensemble-P(C) ≠ φ then
         Nouvel Espace de Recherche = Nouvel Espace de Recherche ∪ {C};
      endif
   else
      Ensemble-P(C) = Ensemble-P(C) - Types;
      if Ensemble-P(C) = φ then
         Espace de Recherche = Espace de Recherche − {C};
      endif
   endif;
end.
```

2.3.1 Utilisation de vecteurs de test spécialisés

L'algorithme présenté ci-dessus accomplit le diagnostic en utilisant n'importe quel vecteur de test donné. Cependant, le processus de diagnostic peut être accéléré en utilisant des vecteurs de test spéciaux orientés-diagnostic.

Théorème 2.1 *:*
Soit un circuit arborescent soumis à l'hypothèse d'une seule erreur due au remplacement d'un composant. Deux vecteurs de test sont suffisants pour préciser le cône d'influence du composant erroné C_e, quand l'algorithme de diagnostic est utilisé. Ces vecteurs sont celui qui génère aux entrées de C_e un vecteur caractéristique distinguant le composant erroné C_e du composant correct C_c, Discriminant(C_e, C_c), et celui qui génère aux entrées de C_e un vecteur NonDiscriminant(C_e, C_c), pourvu que:

1. *Discriminant(C_e, C_c) et NonDiscriminant(C_e, C_c) génèrent la même valeur H à la sortie de C_e, où $H \in \{0, 1\}$;*

2. *Discriminant*(C_e, C_c) *et NonDiscriminant*(C_e, C_c) *sensibilisent le même chemin de la sortie de* C_e *aux sorties primaires, et que les valeurs associées aux entrées primaires pour sensibiliser ce chemin soient les mêmes pour Discriminant*(C_e, C_c) *et pour NonDiscriminant*(C_e, C_c). $\diamond$

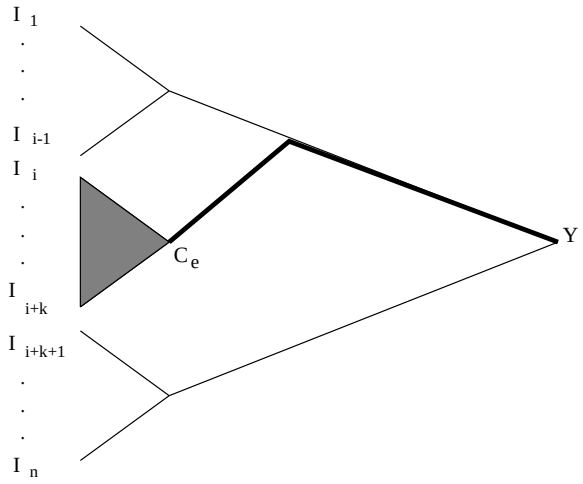

FIG. 2.2 - *Cône d'influence de* C_e *(partie ombrée)*

Preuve :

Sous l'application de *Discriminant*(C_e, C_c) ou de *NonDiscriminant*(C_e, C_c) les entrées $I_1, I_2, \ldots, I_{i-1}$ et $I_{i+k+1}, I_{i+k+2}, \ldots, I_n$ sont fixées à des valeurs telles que la valeur H obtenue à la sortie de C_e peut se propager et atteindre la sortie Y (voir Figure 2.2). Les valeurs de ces entrées sont les mêmes sous *Discriminant*(C_e, C_c) et *NonDiscriminant*(C_e, C_c). La différence est seulement dans le cône d'influence de C_e, *Cône*(C_e).

Quand le diagnostic par rétro-propagation est effectué sous l'application de $VT_1 =$ *Discriminant*(C_e, C_c), l'ensemble *Types-Possibles*(C, VT_1) de chaque composant examiné, C, contient les types de composant tels que la valeur de Y soit corrigée si C est remplacé par un autre composant de type $T \in$ *Types-Possibles*(C, VT_1).

Sous l'application de $VT_2 =$ *NonDiscriminant*(C_e, C_c), l'ensemble *Types-Impossibles*(C, VT_2) de chaque composant examiné, C, contient les types de composant tels que la valeur de Y change si C est remplacé par un autre composant de type $T \in$ *Types-Impossibles*(C, VT_2).

Puisque les valeurs de $I_1, I_2, \ldots, I_{i-1}, I_{i+k+1}, I_{i+k+2}, \ldots I_n$ sont les mêmes sous l'application de $VT_1 = Discriminant\ (C_e, C_c)$ et $VT_2 = NonDiscriminant(C_e, C_c)$, et puisque la valeur H générée à la sortie de C_e est aussi la même, alors les valeurs de tous les signaux à l'extérieur du cône d'influence de C_e sont les mêmes sous l'application des deux vecteurs. Par conséquent tous les types dans *Types-Possibles*(C, VT_1) d'un certain composant C existent aussi dans *Types-Impossibles*(C, VT_2). Puisque l'ensemble *Types-Possibles*(C, VT_1) est intersecté avec *Ensemble-P(C)*, et l'ensemble *Types-Impossibles*(C, VT_2) est enlevé de *Ensemble-P(C)*, alors tous les composants à l'extérieur de *Cône*(C_e) ont *Ensemble-P(C)* $= \phi$, et donc il ne sont pas suspects. $\diamond$

Remarque :

Dans le cas général, le vecteur *Discriminant*(C_e, C_c) et le vecteur *NonDiscriminant*(C_e, C_c) ne sont pas uniques.

Corollaire 2.1 *:*

Le théorème qui vient d'être présenté est aussi valable pour les circuits non arborescents, pourvu que le cône d'influence du composant erroné C_e soit arborescent. $\diamond$

Preuve :

Si le cône d'influence du composant erroné C_e est arborescent, les valeurs des signaux dans *Cône*(C_e) peuvent affecter les autres signaux à l'extérieur de ce cône seulement au travers d'un chemin passant par C_e. Puisque la valeur générée à la sortie de C_e est la même pour *Discriminant*(C_e, C_c) et *NonDiscriminant*(C_e, C_c), et puisque le chemin sensibilisé par *Discriminant*(C_e, C_c) est le même que celui sensibilisé par *NonDiscriminant*(C_e, C_c), alors tous les composants hors de *Cône*(L) auront les mêmes valeurs d'entrées et de sorties sous l'application de *Discriminant*(C_e, C_c) et de *NonDiscriminant*(C_e, C_c). Ce qui s'applique aux composants hors de *Cône*(C_e) dans la preuve du théorème s'applique également ici, et le théorème reste valable. $\diamond$

Corollaire 2.2 :

Soit un circuit non arborescent; le théorème est toujours valable si les trois conditions suivantes sont satisfaites:

1. Discriminant(C_e, C_c) *et* NonDiscriminant(C_e, C_c) *génèrent la même valeur* H *à la sortie de* C_e*, où* $H \in \{0, 1\}$.

2. Discriminant(C_e, C_c) *et* NonDiscriminant(C_e, C_c) *sensibilisent le même chemin de la sortie de* C_e *jusqu'aux sorties primaires, et les valeurs associées aux entrées primaires pour sensibiliser ce chemin sont les mêmes dans les deux vecteurs.*

3. *Chaque composant dans* Cône(C_e) *affecte les autres composants, à l'extérieur de* Cône(C_e) *seulement au travers d'un chemin passant par* C_e. $\diamond$

Preuve :

Si le lien entre *Cône*(C_e) et les autres parties du circuit est fait seulement à travers la sortie de C_e, les conditions sous lesquelles la preuve du théorème est faite sont toujours valable, et donc le théorème est toujours valable dans ce cas aussi. $\diamond$

Dans le cas particulier où l'implémentation est construite seulement en utilisant des portes simples (AND/NAND/OR/NOR/NOT), le diagnostic par rétro-propagation en utilisant *Discriminant*(C_e, C_c) et *NonDiscriminant*(C_e, C_c) peut identifier l'emplacement exact de l'erreur, et non pas seulement son cône d'influence. Cela est prouvé dans [150].

L'algorithme de diagnostic complet, présenté ci-dessous, s'ensuit du théorème précédent et de ses corollaires. Il commence par sélectionner le composant de l'implémentation ayant la plus grande sortance. Nous faisons appel à l'heuristique suivante: la génération des vecteurs de test du composant ayant la plus grande sortance est plus facile, et l'analyse de cette porte permet de diminuer plus rapidement l'*espace de recherche* car elle touche une grande partie du circuit. Les vecteurs *Discriminant* et *NonDiscriminant* sont générés pour ce composant, puis le diagnostic par rétro-propagation est appliqué en faisant appel à la procédure

diagnostic-combinatoire décrite auparavant. La taille de l'*espace de recherche* diminue progressivement, au fur et à mesure que les vecteurs de test sont appliqués. Si l'*espace de recherche* devient vide, l'erreur n'est pas due au replacement d'un composant, et peut être due à un autre modèle d'erreur (une erreur de connexion par exemple). Quand l'exécution de l'algorithme s'achève, l'*espace de recherche* contient les composants qui sont candidats d'erreurs. L'*Ensemble-P* de chacun de ces candidats contient les types corrects qui peuvent être utilisés à la place du type erroné. Les résultats expérimentaux, que nous présenterons dans la section suivante, montrent que dans la plupart des cas l'algorithme se termine avec un seul élément ayant un seul type dans l'*Ensemble-P*.

---

algorithm diagnostic-composant-combinatoire;
begin
  Testé $\leftarrow \phi$;
  Espace-de-Recherche $\leftarrow$ tous les composants dans *IMPL*;
  for tout composant $C \in$ *Espace-de-Recherche* **do**
    $T = type(C)$;
    Ensemble-P(C) $= \{T_1 \mid (T_1 \in$ classe de remplacement de $T) \wedge (T \neq T_1)\ \}$;
  endfor
  while ($\mid$ *Espace-de-Recherche* $\mid > 1$) **and** (*Espace-de-Recherche* $\not\subseteq$ *Testé*) **do**
    Soit C_e un composant dans *Espace-de-Recherche* ayant la plus grande sortance,
    et $C_e \notin Testé$
    $Testé \leftarrow Testé \cup \{C_e\}$;
    for tout composant C_c tel que $type(C_c) \in$ *Ensemble-P*(C_e) **do**
    begin
      générer *Discriminant*(C_e, C_c);
      Simuler *SPEC* et *IMPL* sous l'application de *Discriminant*(C_e, C_c);
      if $\forall i, 1 \leq i \leq m, y_i = w_i$ **then**
        Ensemble-P(C_e) = *Ensemble-P*(C_e) $-$ $type(C_c)$;
      endif;
      diagnostic-combinatoire(*Discriminant*(C_e, C_c),*Espace-de-Recherche*,
                                Tableau-Ensembles-P,Y,W);

      générer *NonDiscriminant*(C_e, C_c);
      Simuler *SPEC* et *IMPL* sous l'application de *NonDiscriminant*(C_e, C_c);
      if $\exists i, 1 \leq i \leq m, y_i \neq w_i$ **then**;
        Ensemble-P(C_e) = *Ensemble-P*(C_e) $-$ $type(C_c)$;
      endif;

diagnostic-combinatoire(*NonDiscriminant*(C_e, C_c),*Espace-de-Recherche*, *Tableau-Ensembles-P*,*Y*,*W*);

endfor

endwhile

for tout composant $C \in$ *Espace-de-Recherche* **do**

begin

print(*C*,*Ensemble-P(C)*);

endfor

end.

2.4 Résultats expérimentaux

Pour valider l'algorithme précédent, nous avons réalisé un système de diagnostic prototype. Ce système se compose de trois parties toutes écrites en PROLOG: un générateur de vecteurs de test, un simulateur et une partie de diagnostic. Le système est testé sur l'ensemble des jeux d'essai ISCAS'85 [27]. Ces circuits sont décrits par des portes simples: AND, OR, NOR, NAND, XOR, XNOR, et NOT. Nous avons écrit un programme qui reconnaît des sous-ensembles dans la description originale et qui les remplace par des composants plus complexes. Nous avons ainsi généré pour chacun de ces circuits une autre version décrite par des composants plus complexes.

Les quatres premières colonnes de la table 2.2 donnent, pour chacun des circuits, son nom, le nombre d'entrées, le nombre de sorties, et le nombre de portes dans la description originale. La cinquième colonne donne le nombre de composants dans la nouvelle description.

Dans nos tests nous avons pris comme spécification la version originale, et comme implémentation la nouvelle version. Nous avons écrit un programme qui insère automatiquement des erreurs aléatoires à des emplacements choisis aléatoirement avant que l'algorithme de diagnostic ne soit exécuté. Des milliers de tests ont été effectués de cette manière, et dans tous les cas l'erreur a été trouvée parmi les candidats proposés.

Les résultats obtenus après exécution sur une station SUN SPARC-10 à 128 Mega Octets de mémoire sont donnés dans le tableau 2.3.

La première colonne de ce tableau donne le nom du circuit; la seconde colonne donne le nombre d'expériences effectuées (Exp.), chacune pour une erreur

Circuit	Entrées	Sorties	# Portes	# Composants
C432	36	7	160	141
C499	41	32	202	170
C880	60	26	383	229
C1355	41	32	546	370
C1908	33	25	880	683
C2670	233	140	1193	897
C3540	50	22	1669	1365
C5315	178	123	2307	1758
C6288	32	32	2406	1456
C7552	207	108	3512	2584

TAB. 2.2 - *Description des jeux d'essai ISCAS'85*

aléatoire. Les trois colonnes suivantes donnent le nombre moyen de vecteurs de test (Vec.) utilisés dans chaque expérience, le temps CPU moyen en secondes (ce dernier inclut le temps de génération des vecteurs de test, le temps de simulation et le temps de diagnostic), et le nombre moyen de candidats d'erreur proposés par l'algorithme. Les écarts types de ces valeurs sont montrés dans les trois dernières colonnes.

Pour montrer les extrêmes, nous donnons aussi ces valeurs dans le meilleur et le pire des cas, dans le tableau 2.4 sous les colonnes intitulées "Max de" et "Min de".

Il est à noter que l'erreur a été trouvée dans tous les cas (le taux de réussite est toujours de 100 %), et que le nombre de candidats proposés est égal à '1' dans presque tous les cas (cinquième colonne du tableau 2.3). L'écart type du temps CPU est relativement grand. Ceci est directement relié à l'écart type du nombre de vecteurs de test, qui dépend à son tour de la nature de l'erreur et de la structure du circuit. Cependant, l'écart type du nombre de candidats d'erreur reste toujours petit, ce qui est le plus important du point de vue du diagnostic.

Le nombre de vecteurs de tests utilisés avant que l'erreur n'ait été localisée est petit par rapport aux autres méthodes. Par exemple, ce nombre s'élève à (64-

Circuit	Exp.	Moyenne de			Ecart type de		
		Vec.	CPU(sec)	Cand.	Vec.	CPU(sec)	Cand.
C432	185	6.37	27.13	1.30	3.45	20.42	0.70
C499	344	8.51	38.24	1.10	3.75	24.13	0.31
C880	568	7.30	27.35	1.30	3.68	24.53	0.60
C1355	148	6.90	196.63	2.77	1.95	72.62	1.06
C1908	257	6.07	254.43	1.47	5.21	361.63	1.40
C2670	124	7.56	411.08	1.35	4.54	592.22	1.09
C3540	77	7.94	872.52	1.16	2.98	846.71	0.40
C5315	149	5.44	528.72	1.06	2.72	471.98	0.24
C6288	16	3.88	607.54	2.12	0.89	737.99	1.63
C7552	43	6.42	1229.94	1.19	10.54	2720.03	0.66

TAB. 2.3 - *Résultats des tests effectués sur les jeux d'essai ISCAS'85*

256) pour les mêmes jeux d'essai dans la méthode présentée en [141]. Dans notre méthode, les résultats expérimentaux montrent que ce nombre est en moyenne entre 6 et 10 vecteurs de test, et ceci n'augmente presque pas avec la taille du circuit. Dans le pire des cas ce nombre n'a pas depassé 54 vecteurs de test (le circuit C7552).

Nous avons aussi testé nos algorithmes sur un circuit qui nous a été fourni par THOMSON-TCS. C'est un circuit de CAG (générateur d'adresse de colonne), qui utilise une bibliothèque de composants contenant 44 types de composant.

Selon notre modèle d'erreur, un composant de l'implémentation peut être remplacé par un autre composant, provenant de la bibliothèque de composants, ayant le même nombre d'entrées et de sorties (pour le moment, nous nous sommes cantonnés aux composants à une seule sortie). Par exemple, une erreur peut être due au remplacement d'un composant AND à 4 entrées par un composant AO22, ou au remplacement d'un composant AO21 par un MUX2, etc. Nous avons effectué 100 expériences de diagnostic sur ce circuit, dont les résultats sont donnés dans le tableau 2.5. Les quatres premières colonnes décrivent le circuit; elles donnent son nom, le nombre d'entrées (E.), le nombre de sorties(S.), et le nombre de compo-

Circuit	Max. de			Min. de		
	Vec.	CPU(sec)	Cand.	Vec.	CPU(sec)	Cand.
C432	18	98.45	4	2	2.76	1
C499	22	141.37	2	4	10.34	1
C880	22	196.14	6	2	2.82	1
C1355	10	445.15	8	2	82.43	1
C1908	28	2402.87	9	2	26.16	1
C2670	27	3537.35	6	2	31.81	1
C3540	25	4734.26	3	3	63.65	1
C5315	17	3060.43	2	2	107.73	1
C6288	6	2590.39	6	2	16.94	1
C7552	54	11244.18	5	2	117.71	1

TAB. 2.4 - *Les extrêmes des résultats des tests effectués sur les jeux d'essai ISCAS'85*

sants (Comp.). La cinquième colonne donne le nombre d'expériences effectuées.

Dans une seule expérience le nombre de candidats d'erreur était de 4, tandis que dans tous les autres cas ce nombre était toujours de 1 ou de 2. Dans les cas où il y avait plus d'un candidat, nous avons corrigé l'implémentation en effectuant le changement proposé. L'implémentation corrigée a ensuite été vérifiée par rapport à sa spécification. Dans presque tous les cas, toutes les propositions données par les candidats d'erreur menaient à une implémentation correcte. Cela est dû au fait que l'implémentation d'une fonction donnée n'est pas unique.

Dans le pire des cas, le temps CPU a été de 14.96 secondes et le nombre de paires de vecteurs de test a été de 6. Dans le meilleur des cas, le temps CPU a été de 0.93 secondes, et l'erreur a été trouvée après l'application d'une seule paire de vecteurs de test.

Les temps CPU donnés dans ce tableau sont basés sur notre réalisation expérimentale de l'algorithme en utilisant le langage de programmation PROLOG sur une station de travail SUN SPARC-10. Une grande amélioration de la performance est possible si les algorithmes sont reprogrammés dans un langage qui

Nom du Circuit	Nombre de				Nombre moyen de vecteurs	Temps CPU moyen(sec)	Nombre moyen de candidats
	E.	S.	Comp.	Exp.			
CAG	5	14	71	100	2.68	3.74	1.18

TAB. 2.5 - *Résultats des tests effectués sur le circuit CAG*

génère un code exécutable plus efficace, comme le langage C par exemple.

Borne supérieure du temps de diagnostic :

Soit une implémentation utilisant une bibliothèque de composants contenant $k \times l$ types de composants qui peuvent être classés en k classes de remplacement. Chaque classe contient l types différents de composants. Pour un composant donné, l'algorithme de diagnostic a besoin de générer $l-1$ paires de vecteurs de test dans le pire des cas. Si l'implémentation est composée de c composants, alors dans le pire des cas où chaque paire de vecteur de test élimine un seul composant de l'espace de recherche, le nombre de paires de vecteurs de test est $c \times (l-1)$ paires. Si sous l'application d'un vecteur de test l'algorithme de diagnostic par rétro-propagation examine tous les composants du circuit (ce qui n'arrive jamais en pratique), le temps d'exécution sera proportionnel à $c^2 \times (l-1)$. Donc, dans le pire des cas le temps d'exécution augmente d'une manière quadratique avec la taille du circuit. Il à noter que cette limite est une limite théorique, et qu'elle n'est jamais atteinte en pratique.

Chapitre 3

Diagnostic de fautes de connexion dans les circuits logiques combinatoires

Nous présentons dans ce chapitre de nouveaux algorithmes pour la localisation et la correction des erreurs de connexion dans les circuits combinatoires. Les types d'erreurs que nous considérons dans ce chapitre sont les erreurs de connexions excédentaires, les erreurs de connexions manquantes, et les erreurs de connexions déplacées.

Nous donnons les définitions et la terminologie dans la section 3.1, puis nous consacrons une section pour chaque type d'erreur. Dans chacune de ces sections nous commençons par présenter la méthode de diagnostic. Cette méthode est valide sous l'application de n'importe quel vecteur de test. Ensuite nous présentons une stratégie de génération de vecteurs de test spécifiques pour accélérer le diagnostic.

3.1 Définitions et terminologie

Une *connexion* de la sortie d'une porte P_1 à l'entrée d'une porte P_2 est dénotée $C(P_1, P_2)$.

Hypothèses de base :

Les algorithmes présentés dans ce chapitre reposent sur le fait que l'implémentation ne contient que des portes simples: AND, NAND, OR, NOR, NOT et BUF. Toutefois, cela n'impose aucune contrainte sur l'utilisation des bibliothèques de composants plus complexes car nous remplaçons chacun de ces composants par sa représentation en termes de portes simples avant de faire le diagnostic.

Le modèle d'erreurs de connexion que nous considérons dans ce chapitre couvre les cinq types d'erreurs évoqués dans la figure 1.2. Ces erreurs sont expliquées de façon plus détaillée ci-après:

1. **Une connexion manquante à l'entrée d'une porte:** une porte à $n-1$ entrées est employée à la place d'une autre porte à n entrées. Toutes les $n-1$ entrées sont correctement connectées.

2. **Des connexions excédentaires à l'entrée d'une porte:** une porte à $n+m$ entrées est employée à la place d'une autre porte à n entrées. Toutes les n entrées sont correctement connectées, et les m connexions excédentaires sont connectées à des nœuds arbitraires dans le circuit.

3. **Une connexion déplacée à l'entrée d'une porte:** une connexion $C(P_1, P_e)$ est remplacée par une autre connexion $C(P_2, P_e)$, où $P_1 \neq P_2$.

Une hypothèse essentielle dans notre travail est que le circuit est combinatoire, et qu'il reste combinatoire après l'insertion de l'erreur. Cela signifie que l'erreur n'introduit pas de boucles dans le circuit. Il est à noter que notre travail concerne le diagnostic de la conception logique avant la fabrication de circuit. Dans un circuit fabriqué une faute de connexion peut introduire des effets mémoire. Ceci ne rentre pas dans le cadre de notre travail.

Les connexions déplacées et les connexions excédentaires à des valeurs constantes sont des cas particuliers d'erreurs de connexions déplacées et de connexions excédentaires. Nous les désignons explicitement parce que, comme il nous l'a été précisé par les ingénieurs de Thomson TCS, les erreurs de connexions à V_{cc} ou V_{dd} arrivent souvent en pratique. Nous représenterons la valeur constante '1' par la sortie d'une porte de type OR ayant comme entrées une entrée primaire i quelconque et son complément. La valeur '0' sera représentée par une construction semblable en utilisant une porte de type AND. Lors de la génération des vecteurs

de test cette entrée primaire est toujours fixée à une valeur binaire, '1' ou '0'. Cela n'implique aucune contrainte supplémentaire sur le générateur de vecteurs de test. La génération se fait comme d'habitude, et si dans le vecteur de test résultant, la valeur de i est '1' ou '0', rien de plus n'est fait. Par contre, si la valeur de i n'est pas spécifiée, c'est-à-dire 'X', cela signifie que la valeur de i n'a pas d'influence sur le chemin sensibilisé ni sur les conditions d'excitation de l'erreur; et donc, cette valeur est remplacée par une valeur binaire arbitraire.

La méthode de diagnostic que nous utilisons est basée sur *l'hypothèse d'erreur:* un type d'erreur est sélectionné (voir Table 1.2), puis le diagnostic est fait sous l'hypothèse que l'erreur est de ce type. Si l'erreur n'est pas trouvée, un autre type est sélectionné et ainsi de suite [146].

Dans une version précédente de ce travail [151], nous avons établi, pour chaque type d'erreurs de connexions et pour chaque type de porte, un ensemble de règles permettant d'identifier les portes suspectées et aussi les chemins aux travers lesquels le parcours du circuit s'effectuera pendant l'analyse sous l'application d'un vecteur de test. Pour faciliter la description et pour la rendre plus compacte, nous avons défini le nouvel opérateur logique (*) comme suit:

Définition 3.1 *: L'opérateur logique (*).*
La fonction réalisée par l'opérateur (*) est définie dans la table 3.1. ◇

$_b\backslash^a$	1	0	X
'1'	X	0	0
'0'	1	X	1
'X'	1	0	X

TAB. 3.1 - *Opérateur (*). a * b*

Voici un exemple montrant comment l'utilisation de cet opérateur peut faciliter la description de nos algorithmes. Supposons que nous sommes en train d'analyser une porte P de type AND, dans un circuit C, sous l'hypothèse d'une connexion manquante quand un vecteur de test détectant l'erreur VT est appliqué aux entrées primaires de C. Une telle porte est suspectée dans les trois cas

suivant:

1. $VCourante(P, VT) = 1$, et $VRequise(P, VT) = 0$
2. $VCourante(P, VT) = 1$, et $VRequise(P, VT) = X$
3. $VCourante(P, VT) = X$, et $VRequise(P, VT) = 0$

Ce sont les trois seuls cas où l'ajout d'une connexion ayant la valeur $VRequise(P, VT)$ à l'entrée de P produira $VRequise(P, VT)$ à sa sortie.

A l'aide de l'opérateur (*) nous pouvons exprimer le même fait en disant que P sera suspectée si, et seulement si, $VRequise(P, VT) * VCourante(P, VT) = 0$.

Définition 3.2 *: La fonction de suspicion.*
Nous définissons la fonction de suspicion d'une porte P, de type Type(P) quand un vecteur de test VT est appliqué aux entrées primaires de l'implémentation, comme suit:

$$Sus(P, VT) =$$

$$\begin{cases} VRequise(P, VT) * VCourante(P, VT) & si\ Type(P) = AND/NOR \\ not(VRequise(P, VT) * VCourante(P, VT)) & si\ Type(P) = OR/NAND \\ 0 & si\ Type(P) = NOT/BUF \end{cases}$$

Pour chaque type de porte, il existe une valeur booléenne produite à la sortie de la porte si une de ses entrées est fixée à une valeur booléenne donnée, quelles que soient les valeurs des autres entrées. Cette valeur s'appelle *valeur forcée* et la valeur d'entrée correspondante s'appelle *valeur forçante*. Pour une porte P, nous dénotons ces valeurs *Forcée(P)* et *Forçante(P)*. Table 3.2 donne ces valeurs pour les différents types de porte.

Définition 3.3 *: Valeur compatible avec la valeur de sortie d'une porte.*
Soit V_P la valeur de sortie d'une porte P, $V_P \in \mathcal{T}$. Une valeur $V_{comp}(P, V_P) \in \mathcal{T}$ est dite compatible avec V_P s'il est nécessaire d'associer la valeur $V_{comp}(P, V_P)$ à une ou plusieurs entrées de P pour que sa sortie prenne la valeur V_P. Pour tous les types de porte considérés dans ce chapitre, nous avons:

$V_{comp}(P, V_P) = Forçante(P) \oplus Forcée(P) \oplus V_P$.

L'ensemble des entrées de P qui ont une valeur compatible avec V_P est dénoté $compatible(P, V_P)$. $\diamond$

Type de la porte	*Forçante(P)*	*Forcée(P)*
AND	0	0
OR	1	1
NAND	0	1
NOR	1	0
NOT	$v \in \{1, 0\}$	$\overline{v}$
BUF	$v \in \{1, 0\}$	v

TAB. 3.2 - *Valeurs forcées et valeurs forçantes*

3.2 Diagnostic des fautes de connexions excédentaires

3.2.1 Analyse avec des vecteurs de test détectant l'erreur

Proposition 3.1 *: Portes suspectes sous l'hypothèse d'erreur de connexions excédentaires.*
Une porte P dans IMPL sous l'application d'un vecteur de test détectant l'erreur VT est suspecte d'avoir des connexions excédentaires à ses entrées si et seulement si les conditions suivantes sont réunies:

1. $P \in$ *Espace de Recherche.*
2. $compatible(P, VRequise(P, VT)) \neq \phi$.

Preuve :

La première condition est évidente: seules les portes qui sont dans l'espace de recherche seront examinées. La deuxième condition garantit que $VRequise(P, VT)$ peut être générée à la sortie de P en supprimant une ou

plusieurs de ses entrées, ce qui par conséquent peut corriger la sortie de l'implémentation. ◇

Si une porte P dans *IMPL* est suspecte sous l'application d'un vecteur de test détectant l'erreur VT, alors ses entrées seront classées en trois catégories:

1. L'ensemble des entrées sûrement excédentaires $S_x(P, VT)$:
   $S_x(P, VT)$ est le sous-ensemble des entrées de P qui *doivent* être supprimées, faute de quoi $VRequise(P, VT)$ ne pourra pas être obtenue à la sortie de P.

2. L'ensemble des entrées insuppressibles $I_x(P, VT)$:
   $I_x(P, VT)$ est le sous-ensemble des entrées de P qui *ne peuvent pas* être supprimées *simultanément*, faute de quoi $VRequise(P, VT)$ ne pourra pas être obtenue à la sortie de P.

3. L'ensemble des entrées potentiellement excédentaires $P_x(P, VT)$:
   $P_x(P, VT)$ est le sous-ensemble des entrées de P qui ne sont pas sûrement excédentaires.

Exemple 3.1 *:*
Soit P une porte de type AND, et soient $(A_1, A_2, A_3, A_4, A_5) = (0, 0, 1, X, X)$ ses entrées quand le vecteur de test VT_1 est appliqué aux entrées de l'implémentation. Si $VRequise(P, VT_1) = X$, alors $S_x(P, VT_1) = \{A_1, A_2\}$, $P_x(P, VT_1) = \{A_3, A_4, A_5\}$, et $I_x(P, VT_1) = \{A_4, A_5\}$. ◇

La proposition suivante montre comment ces ensembles sont calculés.

Proposition 3.2 :
Si une porte $P \in IMPL$ est suspectée d'avoir des connexions excédentaires (prop. 3.1) sous l'application d'un vecteur de test détectant l'erreur VT, alors:

$$S_x(P, VT) = compatible(P, VCourante(P, VT))$$

$$I_x(P, VT) = compatible(P, VRequise(P, VT))$$

$$P_x(P, VT) = entrées(P) - S_x(P, VT)$$

Preuve :

Les entrées forçant la sortie de P à avoir la valeur *VCourante(P,VT)* doivent être supprimées pour changer cette valeur. Donc, les entrées de l'ensemble *compatible(P,VCourante(P,VT))* doivent être supprimées, faute de quoi la sortie de la porte ne changera pas sa valeur courante.

Si toutes les entrées compatibles avec *VRequise(P,VT)* sont supprimées simultanément, *VRequise(P,VT)* ne sera pas obtenue. Donc, les entrées de l'ensemble *compatible(P,VRequise(P,VT))* représentent l'ensemble des entrées insuppressibles $I_x(P,VT)$.

Après la suppression de $S_x(P,VT)$, toute entrée ou toute combinaison d'entrées (exceptée $I_x(P,VT)$) des entrées qui restent peut être supprimée sans affecter la valeur de sortie obtenue, et donc ces entrées constituent $P_x(P,VT)$. $\diamond$

Si la porte P est analysée sous l'application de plusieurs vecteurs de test détectant l'erreur $VT_1, VT_2, \cdots VT_n$, les ensembles des entrées sûrement excédentaires, insuppressibles ou potentiellement excédentaires seront donnés par:

$$S_x(P) = S_x(P,VT_1) \cup S_x(P,VT_2) \cup \cdots S_x(P,VT_n)$$

$$I_x(P) = \{I_x(P,VT_1), I_x(P,VT_2), \cdots I_x(P,VT_n)\}$$

$$P_x(P) = entrées(P) - S_x(P)$$

Il est à noter que $S_x(P)$ et $P_x(P)$ sont des sous-ensembles des *entrées* de la porte, tandis que $I_x(P)$ est l'ensemble des *combinaisons d'entrées* qui ne peuvent pas être supprimées simultanément. (c'est-à-dire $I_x(P)$ est un ensemble d'ensembles).

Exemple 3.2 *:*

Pour la même porte que dans l'exemple 3.1, si sous l'application d'un autre vecteur de test VT_2*, les entrées de* P *sont* $(A_1, A_2, A_3, A_4, A_5) = (0, 1, 0, 1, X)$*, et si* $VRequise(P, VT_2) = 'X'$*, alors* $S_x(P, VT_2) = \{A_1, A_3\}$*,* $P_x(P, VT_2) = \{A_2, A_4, A_5\}$*, et* $I_x(P, VT_2) = \{A_5\}$*. En combinant les résultats de* VT_1 *et* VT_2 *nous obtenons*

$S_x(P) = \{A_1, A_2, A_3\}$, $P_x(P) = \{A_4, A_5\}$, et $I_x(P) = \{\{A_4, A_5\}, \{A_5\}\}$). Cela veut dire que si P est la porte erronée, alors les entrées $\{A_1, A_2, A_3\}$ sont sûrement excédentaires, tandis que l'entrée A_4 peut être *une entrée excédentaire.* $\diamond$

3.2.2 Analyse avec les vecteurs de test ne détectant pas l'erreur

Dans certains cas, les vecteurs de test utilisés ne détectent pas l'erreur, ou ils détectent l'erreur sur quelques sorties tandis que les autres sont correctes. L'analyse des circuits sous l'application de ces vecteurs de test est toujours utile car elle permet d'éliminer quelques connexions suspectes.

Proposition 3.3 :
Si la valeur courante de la sortie d'une porte P, $VCourante(P, VT)$, doit être fixée sous l'application d'un vecteur de test VT pour que la valeur d'une sortie primaire reste invariable, alors l'ensemble $compatible(P, VCourante(P, VT))$ est un élément de $I_x(P)$. $\diamond$

Preuve :

Puisque la valeur courante de la porte P doit être gardée invariable pour ne pas changer la valeur de la sortie primaire, alors les entrées de P qui génèrent $VCourante(P, VT)$ ne peuvent pas être enlevées simultanément. Ces entrées sont données par $compatible(P, VCourante(P, VT))$. $\diamond$

3.2.3 L'algorithme de diagnostic

L'algorithme de diagnostic que nous présentons dans cette section est fondé sur les propositions données ci-dessus. Soit VT un vecteur de test donné; l'algorithme parcourt le circuit des sorties primaires vers les entrées primaires. Ceci est fait par l'appel récursif à la procédure *analyser-x-cnct* décrite par la suite. Cette procédure calcule pour une porte P, donnée comme paramètre, les ensembles $S_x(P, VT)$, $P_x(P, VT)$, et $I_x(P, VT)$. Au départ, l'*espace de recherche* contient toutes les portes du circuit qui ont plus d'une entrée. L'algorithme *diagnostiquer-x-cnct* est ensuite exécuté à plusieurs reprises, avec des vecteurs de test différents, jusqu'à ce que l'erreur soit localisée, ou jusqu'à ce qu'il ne reste plus de vecteurs

de test à générer. La génération des vecteurs de test sera discutée dans la section suivante.

---

algorithm diagnostiquer-x-cnct(VT);
begin
  Simuler *SPEC* et *IMPL* sous l'application de VT;
  for toute porte P_{y_i} dont la sortie est la sortie primaire $y_i \neq X$ **do**
    Nouvel Espace de Recherche $= \phi$;
    if $y_i \neq w_i$ **then**
      Type-Vct:= VDE; % *vecteur détectant l'erreur*
    else
      Type-Vct:= VNE; % *vecteur ne détectant pas l'erreur*
    endif;
    analyser-x-cnct(*Type-Vct*,P_{y_i});
    if *Type-Vct* = VDE **then**
      Espace de Recherche = *Nouvel Espace de Recherche*;
    endif;
  endfor;
end.

Dans la procédure suivante nous faison appel aux notions de *nœuds-modifiables* et d'*entrées-fixes* introduites dans le chapitre précédent.

---

procedure analyser-x-cnct(*Type-Vct*,P);
begin
  if P est une entrée primaire **then**
    exit;
  endif;
  if *Type-Vct* = VDE **then**
    if ($P \in$ *Espace de Recherche*) $\wedge$ (*compatible*(P, *VRequise*(P, VT)) $\neq \phi$) **then**
      % proposition 3.1.
      Nouvel Espace de Recherche = *Nouvel Espace de Recherche* $\cup \{P\}$;
      $S_x = S_x(P, VT)$;
      $P_x = P_x(P, VT)$;
      $I_x = I_x(P, VT)$;
      mettre-à-jour-x-cnct(*Type-Vct*,P, S_x, P_x, I_x);
    endif;
    for toute entrée $i \in$ *Nœuds-modifiables*(P) **do** % proposition 2.4

```
        analyser-x-cnct(Type-Vct,i);
    endfor;
  else        % Type-Vct = VNE
    I_x = I_x(P,VT);
    mettre-à-jour-x-cnct(Type-Vct,P,−,−,I_x);      % '-': paramètre non utilisé.
    for toute entrée i ∈ Entrées-fixes(P) do        % proposition 2.5
        analyser-x-cnct(Type-Vct,i);
    endfor;
  endif; end.
```

Si une porte P a déjà été examinée sous l'application d'autres vecteurs de test, il lui sera associés les ensembles de connexions sûrement excédentaires $S_x(P)$, insuppressibles $I_x(P)$, et potentiellement excédentaires $P_x(P)$. Si cette porte est examinée sous l'application d'un autre vecteur de test VT, il lui sera associés les nouveaux ensembles $S_x(P,VT)$, $P_x(P,VT)$ et $I_x(P,VT)$. Les ensembles qui existent déjà sont mis à jour par la procédure *mettre-à-jour-x-cnct* décrite ci-dessous.

```
procedure mettre-à-jour-x-cnct(Type-Vct,P,S_x,P_x,I_x);
begin
  if P est déjà examinée then
    if Type-Vct = VDE then
      S_x(P) := S_x(P) ∪ S_x;
      I_x(P) := I_x(P) ∪ {I_x};
      P_x(P) := entrées(P) - S_x(P);
      if ∃Z ∈ I_x(P), Z ⊆ S_x(P) then        % P n'est pas suspecte.
        Nouvel Espace de Recherche = Nouvel Espace de Recherche - P;
      endif;
    else        % Type-Vct = VNE
      if I_x ⊆ S_x(P) then      % P n'est pas suspecte.
        Espace de Recherche = Espace de Recherche - P;
      else
        I_x(P) = I_x(P) ∪ {I_x};
      endif;
    endif;
  else        % Premier examen de la porte P.
    if Type-Vct = VDE then
      S_x(P) := S_x;
```

$I_x(P) := \{I_x\}$;
$P_x(P) := entrées(P)$ - $S_x(P)$;
else
$S_x(P) := \phi$;
$I_x(P) := \{I_x\}$;
$P_x(P) := entrées(P)$;
endif;
endif;
if $\mid I_x \mid = 1$ **then**
$P_x(P) := P_x(P)$ - I_x;
endif;
end.

Exemple 3.3 :
Dans cet exemple, nous montrons quelles sont les différentes phases de l'application de l'algorithme de diagnostic pour trouver une faute de connexion excédentaire dans le circuit c17 du jeu d'essais ISCAS'85 [27]. Le circuit erroné est représenté dans la Figure 3.1, où la ligne pointillée représente une connexion excédentaire.

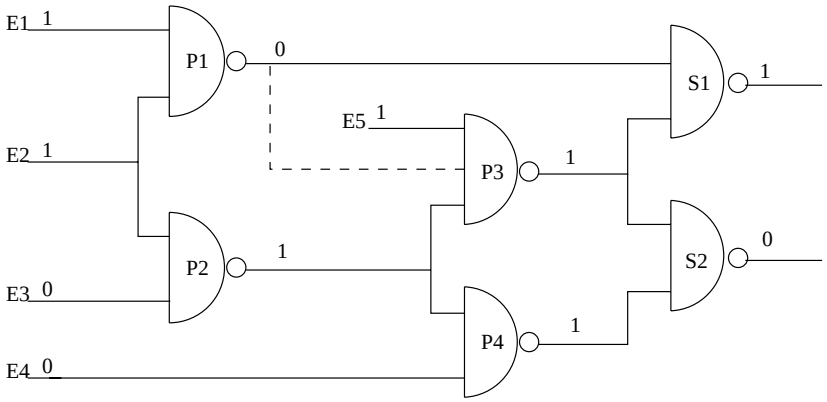

FIG. 3.1 - *Circuit c17 du jeu d'essai ISCAS'85 avec une connexion excédentaire*

$\underline{VT_1 \text{ est } E_1E_2E_3E_4E_5 = 11001}$:
$W(11001) = S_1S_2 = 11$
$Y(11001) = S_1S_2 = 10$
L'analyse commence à la sortie erronée S_2, et se poursuit en marche arrière

vers les entrées primaires. Le vecteur de test est un vecteur détectant l'erreur (VDE). Au départ, l'*espace de recherche* contient toutes les portes du circuit.

A la porte S_2:

$VRequise(S_2, VT_1) = 1$, $Type(S_2) = NAND$.

$compatible(S_2, 1) = \phi$, ce qui signifie que S_2 n'est pas suspectée d'avoir des connexions excédentaires. (Proposition 3.1).

Les nœuds modifiables de S_2 sont P_3 et P_4, et donc l'analyse se dirige vers ces portes.

A la porte P_3:

$VRequise(P_3, VT_1) = 0$, $Type(P_3) = NAND$.

$compatible(P_3, 0) = \{E_5, P_2\}$ et donc P_3 est suspectée d'avoir des connexions excédentaires.

En utilisant la proposition 3.2 nous obtenons:

$S_x(P_3, VT_1) = \{P_1\}$
$I_x(P_3, VT_1) = \{E_5, P_2\}$
$P_x(P_3, VT_1) = \{E_5, P_2\}$

La signification de ce résultat est que, si la porte P_3 est la porte erronée, alors il est certain que son entrée P_1 doit être supprimée, tandis que soit E_5 soit P_2 *peut être* supprimée, mais pas les deux simultanément.

Le seul nœud modifiable de P_3 est P_1, et l'analyse se poursuit vers P_1.

A la porte P_1:

$VRequise(S_2, VT_1) = 1$, $Type(S_2) = NAND$.

$compatible(P_1, 1) = \phi$, et donc P_1 n'est pas suspecte.

P_1 n'a aucune entrée modifiable.

Retournons maintenant à la porte P_4.

A la porte P_4:

$VRequise(P_4, VT_1) = 0$, $Type(P_4) = NAND$.
$compatible(P_4, 0) = \{P_2\}$ et donc P_4 est suspectée d'avoir des connexions excédentaires.

En faisant appel à la proposition 3.2 nous obtenons:

$S_x(P_4, VT_1) = \{E_4\}$
$I_x(P_4, VT_1) = \{P_2\}$
$P_x(P_4, VT_1) = \{P_2\}$

Si la porte erronée est la porte P_4, alors E_4 est une connexion excédentaire, et elle doit être supprimée. Ici, $I_x(P_4, VT_1)$ contient un seul élément plutôt qu'une combinaison d'entrées; et donc cet élément ne pouvant pas être la fausse connexion, il est enlevé de $P_x(P_4, VT_1)$.

P_4 n'a aucun nœud modifiable et donc l'analyse s'arrête à ce point.

Le résumé du résultat obtenu jusqu'à présent est donné dans la table suivante:

Porte	Sûres	Potentielles	Insuppressibles
P_3	$\{P_1\}$	$\{E_5, P_2\}$	$\{\{E_5, P_2\}\}$
P_4	$\{E_4\}$	$\{\}$	$\{\{P_2\}\}$

Cette table montre seulement les portes suspectes. Les autres ne peuvent pas être erronées.

Nous allons à présent faire l'analyse en commençant par la sortie S_1. VT_1 ne détecte pas l'erreur sur cette sortie.

A la sortie S_1:

Cette porte n'est pas suspecte, et donc l'analyse se dirige directement vers ses entrées fixes.

S_1 a une seule entrée fixe qui est P_1.

A la porte P_1:

P_1 n'est pas suspecte et pareillement l'analyse se dirige vers ses entrées fixes E_1 et E_2 qui sont des entrées primaires, et donc l'analyse s'arrête ici.

Utilisons maintenant un autre vecteur de test VT_2. Un tel vecteur est généré automatiquement par l'outil de diagnostic comme nous le montrerons dans la section suivante. Le but ici est de montrer la methode de diagnostic sous l'application de n'importe quel vecteur de test.

VT_2 est $E_1E_2E_3E_4E_5$= 0X0X0:

$$W(0X0X0) = O_1O_2 = 0X$$
$$Y(0X0X0) = O_1O_2 = 0X$$

La spécification et l'implémentation génèrent les mêmes valeurs de sortie. L'analyse est faite seulement à partir de la sortie S_1 qui a une valeur différente de 'X'. Nous obtenons le résultat suivant:
$I_x(P_3, VT_2) = \{E_5\}$
$I_x(P_4, VT_2) = \{E_4\}$

E_4 a déjà été classée comme étant une entrée sûrement excédentaire à l'entrée de la porte P_4. Maintenant, sous l'application de VT_2, E_4 est classée comme étant insuppressible à l'entrée de la porte P_4. Cette contradiction signifie que P_4 ne peut pas être la porte erronée, et elle est donc enlevée de l'*espace de recherche*.

La table suivante résume ces résultats.

Porte	Sûres	Potentielles	Insuppressibles
P_3	$\{P_1\}$	$\{P_2\}$	$\{\{E_5, P_2\},\{E_5\}\}$

Utilisons à ce stade un troisième vecteur de test.

$\underline{VT_3 \text{ est } E_1E_2E_3E_4E_5 = 011\text{X}1}$:

$$W(011X1) = O_1O_2 = 00$$
$$Y(011X1) = O_1O_2 = 00$$

La spécification et l'implémentation génèrent les mêmes valeurs de sortie. En poursuivant la même démarche que précédemment nous obtenons:

$$I_x(P_3, VT_3) = \{P_2\}$$

Ce résultat est utilisé pour mettre à jour les anciens résultats et ainsi aboutir au résultat final suivant:

Porte	Sûres	Potentielles	Insuppressibles
P_3	$\{P_1\}$	$\{\}$	$\{\{E_5, P_2\},\{E_5\},\{P_2\}\}$

Ce résultat final signifie que l'entrée de P_3 provenant de P_1 est une entrée excédentaire, et qu'il faut donc la supprimer. C'est en fait la véritable erreur.

3.2.4 Génération de vecteurs de test

L'algorithme de diagnostic présenté dans la section précédente peut effectuer le processus de diagnostic sous l'application de n'importe quel vecteur de test. Toutefois, l'emploi de vecteurs de test détectant l'erreur accélère le processus pour les raisons suivantes: chaque vecteur détectant l'erreur VT_i détermine un ensemble de portes-suspectes $PS(VT_i)$ auxquelles les connexions excédentaires pourraient exister. Si l'erreur est détectée sous l'application de n vecteurs de test $VT_1, VT_2 \cdots VT_n$, l'erreur doit exister dans l'intersection de $PS(VT_1), PS(VT_2), \cdots PS(VT_n)$. Dans la plupart des cas, cette intersection permet de réduire rapidement la zone suspecte du circuit, surtout quand l'erreur est détectée sur des sorties différentes dans une implémentation à plusieurs sorties. En outre, les vecteurs ne détectant pas l'erreur, comme il l'a été démontré auparavant, peuvent seulement déterminer quelques connexions correctes parmi celles

qui ont déjà été trouvées comme suspectes.

Il existe plusieurs méthodes pour générer ces vecteurs détectant l'erreur. La méthode la plus directe est de calculer la fonction $YW(VT) = Y(VT) \oplus W(VT)$, et de trouver les valeurs des vecteurs d'entrées VT qui rendent $YW(VT) = 1$. Ce calcul pourrait être fait par une des méthodes de manipulation symbolique des fonctions booléennes, comme les BDDs (*Binary Decision Diagrams*). Les BDDs ont pour inconvénient que leur taille augmente d'une manière exponentielle avec certaines fonctions (e.g. les multiplicateurs d'entiers), ou quand l'ordre des variables n'est pas bien choisi. Dans notre travail, nous utilisons l'algorithme présenté dans le chapitre précédent pour générer les vecteurs de test.

Proposition 3.4 *:*

Un vecteur de test VT est capable de détecter si une entrée i d'une porte P est excédentaire ou non, si en appliquant VT aux entrées de IMPL, la valeur Forçante(P) est générée sur i, tandis qu'une autre valeur $V = \overline{Forçante(P)}$ est générée sur les autres entrées de P, et si un chemin est sensibilisé de la sortie de P jusqu'au moins une des sorties primaires. $\diamond$

Preuve :

Pour que la valeur de la sortie de P soit attestée à l'une des sorties primaires, un chemin doit être sensibilisé de la sortie de P jusqu'à cette sortie. Pour que cette valeur soit erronée, l'entrée excédentaire i doit générer une valeur erronée à la sortie de P indépendamment des autres entrées de P. Donc, i doit être fixée à la valeur *Forçante(P)*. Dans ce cas, la sortie de P aura la valeur *Forcée(P)*. La suppression de i doit générer $\overline{Forcée(P)}$ à la sortie de P, et donc les autres entrées de P doivent être fixées à la valeur $\overline{\text{Forçante(P)}}$. $\diamond$

Dans le prototype que nous avons réalisé, nous commençons par générer des vecteurs de test capables de détecter les fautes de connexion excédentaire sur les portes situées près des entrées primaires. Ces vecteurs peuvent aussi détecter d'autres fautes de connexion excédentaire tout au long du chemin sensibilisé. Toute porte P sur ce chemin aura une entrée i ayant la valeur sensibilisée, tandis que toutes ses autres entrées auront la valeur $\overline{\text{Forçante(P)}}$. Si la valeur sensibilisée

est égale à *Forçante(P)*, le vecteur de test généré pourra donc déterminer si i est une entrée excédentaire de P.

L'algorithme de diagnostic est exécuté sous l'application de ces vecteurs de test et l'espace de recherche est progressivement réduit. Des vecteurs de test supplémentaires sont générés pour les portes restant dans l'espace de recherche, en commençant par celles qui sont les plus proches des entrées primaires. La même démarche est répétée jusqu'à ce que l'erreur soit trouvée.

L'algorithme de diagnostic complet est donc donné comme suit:

---

algorithm x-cnct-diag;
begin
  Espace de Recherche = Toutes les portes ayant plus d'une entrée;
  Testées $\leftarrow \phi$;
  while (| *Espace de recherche* | > 1) **and** (*Espace de Recherche* $\not\subseteq$ *Testées*) **do**
    Soit $P \in$ *Espace de Recherche* plus proche des entrées primaires, et $P \notin$ *Testées*
    Testées $\leftarrow Testées \cup \{P\}$;
    for toute entrée suspecte i de P **do**
      VT = Vecteur détectant si i est excédentaire à P; % proposition 3.4
      diagnostiquer-x-cnct(VT);
    endfor
  enddo
  for toute $P \in$ *Espace de Recherche* **do**
    if $P_x(P) \neq \phi$ **or** $S_x(P) \neq \phi$ **then**
      write(P,$P_x(P)$,$S_x(P)$,$I_x(P)$);
    endif
  endfor
end.

3.2.5 Résultats expérimentaux

Pour valider les algorithmes décrits dans les sections précédentes, nous avons réalisé un logiciel prototype en PROLOG, comprenant le générateur de vecteurs de test, le simulateur et les algorithmes de diagnostic. Ce logiciel a été testé sur des circuits de tailles différentes. Ces circuits sont ceux du jeu d'essais ISCAS'85 [27].

Dans chaque test effectué une, deux ou trois connexions sont choisies aléatoirement et insérées à l'entrée d'une porte choisie aléatoirement, puis l'algorithme de diagnostic est exécuté.

Les résultats obtenus sur une station de travail SUN SPARC-10 à 128 Megaoctets de mémoire sont donnés dans le tableau 3.3.

La première colonne de ce tableau donne le nom du circuit; la seconde colonne donne le nombre d'expériences effectuées (Exp.). Les trois colonnes suivantes donnent le nombre moyen de vecteurs de test (Vec.) utilisés dans chaque expérience, le temps CPU moyen en secondes (ce dernier inclut le temps de génération des vecteurs de test, le temps de simulation et le temps de diagnostic), et le nombre moyen de candidats d'erreur proposés par l'algorithme. Ce nombre est le nombre de portes aux entrées desquelles des connexions excédentaires peuvent exister. L'algorithme de diagnostic précise, pour chacune de ces portes l'ensemble d'entrées excédentaires. Les écarts types de ces valeurs sont montrés dans les trois dernières colonnes. Pour montrer les extrêmes, nous donnons aussi ces valeurs dans le meilleur et le pire des cas, dans le tableau 3.4 sous les colonnes intitulées "Max de" et "Min de".

Circuit	Exp.	Moyenne de			Ecart type de		
		Vec.	CPU(sec)	Cand.	Vec.	CPU(sec)	Cand.
C432	109	14.91	37.35	1.06	6.30	21.23	0.28
C499	158	9.85	69.72	1.09	5.64	48.91	0.39
C880	271	8.70	21.66	1.03	4.38	12.46	0.18
C1355	482	13.59	110.51	1.14	9.89	86.37	0.58
C1908	160	14.44	342.27	1.14	11.69	467.48	0.51
C2670	183	17.18	448.56	1.17	13.09	371.13	0.79
C3540	105	12.51	584.92	1.30	6.82	577.79	0.89
C5315	199	8.17	442.91	1.09	3.16	179.27	0.28
C6288	29	26.24	2004.13	1.10	11.58	1014.78	0.41
C7552	67	9.49	908.68	1.12	7.42	908.70	0.66

TAB. 3.3 - *Résultats des tests effectués sur les jeux d'essai ISCAS'85*

Il est à noter que dans la plupart des cas, la porte erronée est localisée précisément avec son ensemble d'entrées excédentaires. Dans d'autres cas plusieurs candidats sont proposés mais la porte erronée est troujours trouvée parmis ces candidats. La valeur moyenne du nombre de candidats est presque '1' dans tous les cas, et l'écart-type est toujours inférieur à 1 (voir la $5^{ème}$ et la $8^{ème}$ colonne du tableau 3.3). L'écart-type du temps CPU est relativement grand. Ceci est justifié par le grand écart-type du nombre de vecteurs de test, qui dépend de la nature de l'erreur et de la structure du circuit.

Circuit	Max. de			Min. de		
	Vec.	CPU(sec)	Cand.	Vec.	CPU(sec)	Cand.
C432	36	117.90	3	4	4.88	1
C499	25	215.34	3	3	12.33	1
C880	27	78.76	2	2	4.34	1
C1355	54	456.02	7	3	14.82	1
C1908	71	4021.12	5	5	57.48	1
C2670	65	1957.60	6	3	57.51	1
C3540	36	2371.99	8	3	88.20	1
C5315	22	1254.19	2	3	135.97	1
C6288	62	3967.09	3	9	594.94	1
C7552	39	4215.89	6	3	322.80	1

TAB. 3.4 - *Les extrêmes des résultats des tests effectués sur les jeux d'essai ISCAS'85*

Le temps d'exécution augmente d'une façon linéaire avec le produit de la taille du circuit par le nombre de vecteurs de test utilisés (voir la Figure 3.2). Le nombre de vecteurs de test utilisés dépend de la topologie du circuit en question.

Cette relation linéaire est justifiée comme suit: le temps nécessaire pour simuler un circuit sous l'application d'un vecteur de test est proportionnel au nombre de portes du circuit. L'algorithme *diagnostiquer-x-cnct* parcourt le circuit de ses sorties vers ses entrées en analysant chaque porte rencontrée pendant ce parcours. Dans le pire des cas, l'algorithme parcourra tous les chemins du circuit en analy-

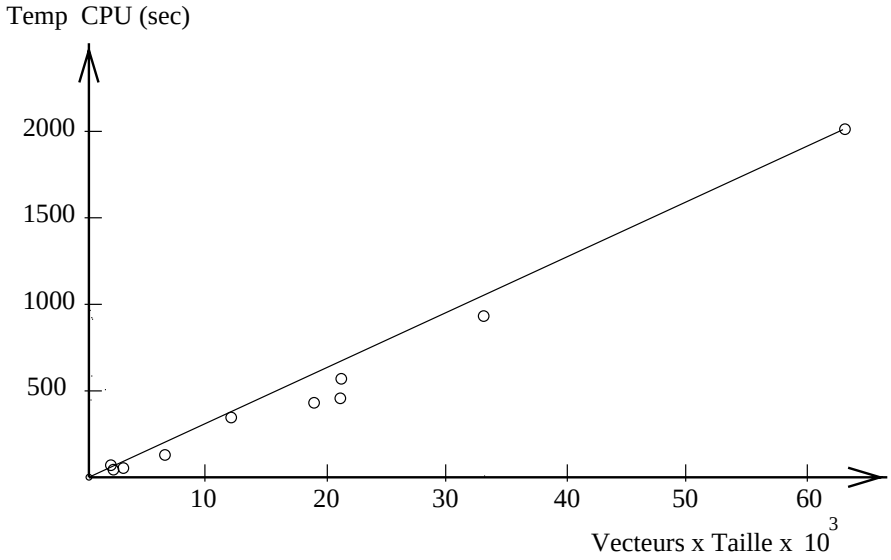

FIG. 3.2 - *Temps de diagnostic en fonction du produit de la taille du circuit par le nombre de vecteurs de test*

sant toutes les portes. Le temps de ce parcours est donc proportionnel au nombre de portes du circuit. Puisque l'algorithme *diagnostiquer-x-cnct* simule le circuit une fois et le parcourt une fois sous l'application d'un vecteur de test donné, le temps pour exécuter cet algorithme est donc proportionnel au nombre de portes du circuit. Si le diagnostic est fait en utilisant v vecteur de test, le temps de diagnostic sera proportionnel à ($v \times$ nombre de portes). Cela explique la relation linéaire dans la figure 3.2.

Il est à noter que le nombre de vecteurs de test utilisés dans le diagnostic a tendance à décroître quand le nombre de sorties primaires augmente (voir les résultats des circuits c5315, et c7552 par exemple). Ceci s'explique par le fait que l'observabilité d'un nœud dans le circuit a tendance à croître avec le nombre de sorties primaires, et donc quand il y a une valeur erronée sur ce nœud, il y a une grande probabilité que l'erreur soit détéctée sur plusieurs sorties simultanément. L'erreur doit se trouver dans le cône d'influence commun à toutes ces sorties, et cela réduit largement le nombre de connexions suspectes.

Borne supérieure du temps de diagnostic :

Soit une implémentation contenant n portes, chacune ayant k entrées en moyenne. Le temps d'exécution de l'algorithme *diagnostiquer-x-cnct* est propor-

tionnel à n. Dans le pire des cas, chaque vecteur de test généré selon la proposition 3.4 est capable de vérifier si une *seule* connexion est excédentaire ou non. [1] S'il en est ainsi, l'algorithme *x-cnct-diag* générera $(n \times k)$ vecteurs de test, et *diagnostiquer-x-cnct* sera exécuté $(n \times k)$ fois. Le temps de diagnostic est donc proportionnel à $(n \times n \times k) = (n^2 \times k)$. La valeur k est constante (< 10 en pratique), et donc nous pouvons dire que dans le pire des cas, le temps de diagnostic est proportionnel à n^2.

3.3 Diagnostic des fautes de connexions manquantes

3.3.1 Analyse avec des vecteurs de test détectant l'erreur

Proposition 3.5 :
Une porte P dans IMPL sous l'application d'un vecteur de test détectant l'erreur VT est suspecte d'avoir une connexion manquante à son entrée si et seulement si les conditions suivantes sont maintenues:

1. *P ∈ Espace de Recherche.*
2. *Sus(P,VT) = 0.*

Preuve :

La première condition est évidente; nous allons seulement discuter la deuxième.

- Si *Type(P)* est AND ou NOR:
  - $Sus(P, VT) = 0$ seulement dans trois cas (Voir la définition 3.1):
    1. $VRequise(P, VT) = 0$ et $VCourante(P, VT) = 1$
       La valeur requise $VRequise(P, VT) = 0$ peut être obtenue à la sortie de P si la valeur d'une de ses entrées est égale à *Forçante(P)*. Si la valeur de la sortie d'une autre porte P_1 est égale à *Forçante(P)*,

1. En pratique, un vecteur de test peut éliminer plusieurs connexions suspectées à la fois.

alors en ajoutant la connexion $C(P_1, P)$, $VRequise(P, VT)$ sera générée à la sortie de P. Donc, P pourrait être suspecte d'avoir une connexion manquante.

2. $VRequise(P, VT) = 0$ et $VCourante(P, VT) = X$
   Le même raisonnement que celui utilisé dans le cas précédent s'applique ici aussi.

3. $VRequise(P, VT) = X$ et $VCourante(P, VT) = 1$
   Ici, la valeur courante $CV(P, VT) = 1$, ce qui signifie que toutes les entrées de la porte P sont fixées à la valeur $\overline{\text{Forçante(P)}}$. Si la valeur de la sortie d'une porte P_1 est égale à 'X', alors en ajoutant la connexion $C(P_1, P)$ la valeur requise $VRequise(P, VT)$ sera obtenue. Donc, P pourrait être suspecte d'avoir une connexion manquante.

- $Sus(P, VT) = X$ seulement si $VRequise(P, VT) = VCourante(P, VT)$.
  Cela signifie que pour corriger l'implémentation il n'y a pas besoin de changer la valeur de la sortie de P. Par conséquent, P n'est pas suspecte.

- $Sus(P, VT) = 1$ dans trois cas:

  1. VRequise(P,VT) = 1 et VCourante(P,VT) = 0
     Dans ce cas, la valeur courante de P est égale à *Forcée(P)*, et donc une ou plusieurs entrées de P sont fixées à la valeur *Forçante(P)*. L'addition de n'importe quelle autre connexion à l'entrée de P ne changera jamais la valeur de sa sortie. Donc, P n'est pas suspecte d'avoir des connexions manquantes.

  2. VRequise(P,VT) = X et VCourante(P,VT) = 0
     Le même raisonnement que celui utilisé dans le cas précédent s'applique ici aussi.

  3. VRequise(P,VT) = 1 et VCourante(P,VT) = X
     Dans ce cas, l'addition d'une autre entrée à la porte P va générer soit '0', si sa valeur est égale à *Forçante(P)*, soit 'X' si sa valeur

n'est pas égale à *Forçante(P)*. Donc, la valeur requise ne pourra pas être obtenue, et P ne sera pas suspecte.

Un raisonnement semblable peut être utilisé si Type(P) est NAND ou OR. $\diamond$

Si une porte $P \in IMPL$ est suspecte d'avoir une connexion manquante sous l'application d'un vecteur de test détectant l'erreur VT, il lui sera associé un ensemble de nœuds desquels la connexion manquante pourra provenir $P_m(P, VT)$. Un nœud est une sortie d'une porte ou une entrée primaire.

Proposition 3.6 *:*
Si une porte $P \in IMPL$ est suspecte d'avoir une connexion manquante sous l'application d'un vecteur de test détectant l'erreur VT, alors:

$$P_m(P, VT) = \{i \mid (i \in n\text{œ}uds\ of\ IMPL)\ \wedge valeur(i) = V_{comp}(P, VRequise(P, VT))\}$$

Preuve :

Pour générer la valeur requise $VRequise(P, VT)$ à la sortie de la porte P, un nœud ayant une valeur compatible avec $VRequise(P, VT)$ doit être connecté à l'entrée de P. Donc, tout nœud ayant une valeur $V_{comp}(P, VRequise(P, VT))$ peut être la connexion manquante. $\diamond$

3.3.2 Analyse avec des vecteurs de test ne détectant pas l'erreur

Dans plusieurs cas, les vecteurs de test utilisés ne détectent pas l'erreur, ou détectent l'erreur sur quelques sorties tandis que les autres sont correctes. L'analyse du circuit en commençant par ces sorties correctes peut aussi diminuer l'*espace de recherche* en excluant quelques connexions correctes.

Proposition 3.7 :
Si la valeur courante de la sortie d'une porte P, $VCourante(P, VT)$, doit être fixée

sous l'application d'un vecteur de test VT pour que la valeur d'une sortie primaire reste invariable, alors toute connexion C(Z,P) est correcte où $Z \in I_m(P,VT)$, et $I_m(P,VT)$ est donné par:

$$I_m(P,VT) = \{i \mid VCourante(P,VT) * (valeur(i) \oplus Forçante(P) \oplus Forcée(P)) = \overline{Forcée(P)}\}$$

Preuve :

Nous apportons une preuve pour chaque type de porte:

- Si *Type(P)* est AND:
  $Forcée(P) = 0$, et *Forçante(P) = 0.*
  Donc, $I_m(P,VT) = \{i \mid (VCourante(P,VT) * valeur(i)) = 1\}$
  - Si $VCourante(P,VT) = 1$, alors $I_m(P,VT)$ contiendra tous les nœuds qui ont la valeur '0' ou 'X' (voir la définition 3.1). Ceci est vrai car l'ajout d'une entrée ayant la valeur '0' ou 'X' à la porte P changera la valeur de sa sortie à '0' ou 'X'.
  - Si $VCourante(P,VT) = 0$, alors $I_m(P,VT)$ sera un ensemble vide. L'ajout de n'importe quelle entrée supplémentaire à P ne changera pas la valeur de sa sortie.
  - Si $VCourante(P,VT) = X$, alors $I_m(P,VT)$ contiendra tous les nœuds ayant la valeur '0'. Ce sont en fait les nœuds qui peuvent changer la valeur de la sortie de P s'ils y sont connectés.

- Si *Type(P)* est NAND:
  $Forcée(P) = 1$, et *Forçante(P) = 0.*
  Donc, $I_m(P,VT) = \{i \mid (VCourante(P,VT) * \overline{valeur(i)}) = 0\}$
  - Si $VCourante(P,VT) = 1$, alors $I_m(P,VT)$ sera un ensemble vide.
  - Si $VCourante(P,VT) = 0$, alors $I_m(P,VT)$ contiendra tous les nœuds ayant la valeur '0' ou 'X'.
  - Si $VCourante(P,VT) = X$, alors $I_m(P,VT)$ contiendra tous les nœuds ayant la valeur '0'.

- Si *Type(P)* est OR:
  $Forcée(P) = 1$, et *Forçante(P) = 1.*
  Donc, $I_m(P,VT) = \{i \mid (VCourante(P,VT) * valeur(i)) = 0\}$
  - Si $VCourante(P,VT) = 1$, alors $I_m(P,VT)$ sera un ensemble vide.
  - Si $VCourante(P,VT) = 0$, alors $I_m(P,VT)$ contiendra tous les nœuds ayant la valeur '1' ou 'X'.
  - Si $VCourante(P,VT) = \mathrm{X}$, alors $I_m(P,VT)$ contiendra tous les nœuds ayant la valeur '1'.

- Si *Type(P)* est NOR:
  $Forcée(P) = 0$, et *Forçante(P) = 1.*
  Donc, $I_m(P,VT) = \{i \mid (VCourante(P,VT) * \overline{valeur(i)}) = 1\}$
  - Si $VCourante(P,VT) = 1$, alors $I_m(P,VT)$ contiendra tous les nœuds ayant la valeur '1' ou 'X'.
  - Si $VCourante(P,VT) = 0$, alors $I_m(P,VT)$ sera un ensemble vide.
  - Si $VCourante(P,VT) = \mathrm{X}$, alors $I_m(P,VT)$ contiendra tous les nœuds ayant la valeur '1'. $\diamond$

Si une porte P est analysée sous l'application de plusieurs vecteurs de test $VT_1, VT_2, \cdots VT_n$, alors les ensembles de nœuds $P_m(P)$ desquels la connexion manquante pourra provenir, et $I_m(P)$ desquels la connexion manquante ne pourra pas provenir sont donnés par:

$$P_m(P) = P_m(P,VT_1) \cap P_m(P,VT_2) \cap \cdots P_m(P,VT_n)$$

$$I_m(P) = I_m(P,VT_1) \cup I_m(P,VT_2) \cup \cdots I_m(P,VT_n)$$

3.3.3 L'algorithme de diagnostic

Nous présentons ici un algorithme de diagnostic de fautes de connexions manquantes basé sur les trois propositions précédentes. Etant donné un vecteur de test VT, l'algorithme parcourt le circuit à partir des sorties primaires et progresse à reculons vers les entrées primaires. Ceci est fait par l'appel récursif à la procédure

analyser-m-cnct décrite ci-après. Au départ, l'*espace de recherche* contient toutes les portes du circuit, sauf les inverseurs et les passeurs (NOT et BUF), car ceux-ci ne peuvent pas avoir de connexions manquantes. L'algorithme *diagnostiquer-m-cnct* est ensuite exécuté à plusieurs reprises jusqu'à ce que l'erreur soit trouvée ou jusqu'à ce qu'il n'y ait plus de vecteurs de test à générer. Nous parlerons de la génération des vecteurs de test dans la section suivante.

---

algorithm diagnostiquer-m-cnct(VT);
begin
  Simuler SPEC et IMPL sous l'application de VT;
  for toute porte P_{y_i} dont la sortie est la sortie primaire $y_i \neq X$ **do**
    Nouvel Espace de Recherche $= \phi$;
    if $y_i \neq w_i$ **then**
      Type-Vct:= VDE; % *vecteur détectant l'erreur*
    else
      Type-Vct:= VNE; % *vecteur ne détectant pas l'erreur*
    endif analyser-m-cnct(*Type-Vct*,P_{y_i});
    if *Type-Vct* = VDE **then**
      Espace de Recherche = Nouvel Espace de Recherche;
    endif;
  endfor;
end.

---

procedure analyser-m-cnct(*Type-Vct*,*P*);
begin
  if *P* est une entrée primaire **then**
    exit
  endif;
  if *Type-Vct* = VDE **then**
    if ($P \in$ *Espace de Recherche*) $\wedge$ (*Sus(P,VT)* $= 0$) **then** % proposition 3.5
      Nouvel Espace de Recherche = Nouvel Espace de Recherche $\cup \{P\}$;
      Nœuds $= P_m(P, VT)$;
      mettre-à-jour-m-cnct(*Type-Vct*,*P*,*Nœuds*);
    endif;
    for toute entrée $i \in$ *Nœuds-modifiables(P)* **do**
      analyser-m-cnct(*Type-Vct*,*i*);
    endfor
  else % *Type-Vct*:= VNE
    Nœuds $= I_m(P, VT)$;

```
        mettre-à-jour-m-cnct(Type-Vct,P,Nœuds);
        for toute entrée i ∈ Entrées-fixes(P) do
            analyser-m-cnct(Type-Vct,i);
        endfor;
    endif;
end.
```

```
procedure mettre-à-jour-m-cnct(Type-Vct,P,Nœuds);
begin
    if Type-Vct = VDE then
        % nœuds est l'ensemble des connexions manquantes possibles
        if P est déjà examinée sous un VDE then
            P_m(P) = P_m(P) ∩ Nœuds;
        elseif P est déjà examinée sous un VNE then
            P_m(P) = Nœuds - I_m(P);
        else                    % la première fois d'examiner P
            P_m(P) = Nœuds;
        endif
    else        % Noeuds est l'ensemble des connexions manquantes impossibles
        if P est déjà examinée sous un VDE then
            P_m(P) = P_m(P) - Nœuds;
        elseif  P est déjà examinée sous un VNE then
            I_m(P) = I_m(P) + Nœuds;
        else                    % la première fois d'examiner P
            I_m(P) = Nœuds;
        endif
    endif;
end.
```

3.3.4 Génération des vecteurs de test

L'algorithme de diagnostic présenté dans la section précédente permet d'analyser le circuit sous l'application de n'importe quel vecteur de test. Cependant, pour accélérer le processus de diagnostic, il est préférable d'employer les vecteurs détectant l'erreur pour les mêmes raisons que celle mentionnées dans la section 3.2.4

Proposition 3.8 *:*

Un vecteur de test VT est capable de détecter si une connexion d'un nœud i à l'entrée d'une porte P est manquante ou non, si en appliquant VT aux entrées d'IMPL, i est fixée à la valeur Forçante(P), si toutes les entrées de P sont fixées à la valeur $\overline{Forçante(P)}$*, et si un chemin est sensibilisé de la sortie de P jusqu'au moins une des sorties primaires.* $\diamond$

Preuve :

Si une entrée de P est fixée à la valeur *Forçante(P)*, alors la mauvaise implémentation et celle correcte généreront la même valeur à la sortie de P et l'erreur ne sera pas détectée. Donc, toutes les entrées de P doivent être fixées à la valeur $\overline{\text{Forçante(P)}}$. Si la valeur de la connexion manquante est *Forçante(P)*, la mauvaise implémentation générera une valeur opposée à celle que génère l'implémentation correcte à la sortie de la porte P. Cette valeur sera ensuite propagée à travers le chemin sensibilisé, et donc l'erreur sera détectée. $\diamond$

Dans le prototype que nous avons réalisé, nous commençons par appliquer un contre-exemple, qui est un vecteur détectant l'erreur, généré par le vérificateur ou par n'importe quel autre moyen. Cela limite le nombre de portes suspectes et leurs connexions manquantes possibles. Pour toute porte suspecte P ayant des connexions manquantes possibles provenant de $n_1, n_2, \cdots n_m$, des vecteurs sont générés pour détecter si $C(n_i, P)$ $(i \in [1, m])$ est une connexion manquante ou non, et l'algorithme de diagnostic et exécuté. Les résultats expérimentaux que nous présentons dans la section suivante montrent qu'après l'emploi d'un petit nombre de vecteurs de test, ainsi générés, la connexion manquante est identifiée.

L'algorithme complet est présenté ci-dessous:

---

algorithm m-cnct-diag;
begin
Espace de Recherche = Toutes les portes de IMPL - $\{p \mid Type(p) =$ NOT ou BUF$\}$;
$Test \leftarrow \phi$;
while ($\mid$ *Espace de Recherche* $\mid > 1$) **and** (*Espace de Recherche* $\not\subseteq$ *Test*) **do**
Soit $P \in$ *Espace de Recherche* plus proche des entrées primaires, et $P \notin Testées$
$Testées \leftarrow Testées \cup \{P\}$;
for toute connexion suspecte $C(i, P)$ **do**

```
        VT = Vecteur détectant si C(i,P) est manquante;  % proposition 3.8
        diagnostiquer-m-cnct(VT);
      endfor
    enddo
    for toute P ∈ Espace de Recherche do
      if P_m(P) ≠ φ then
        write(P,P_m(P));
      endif
    endfor
end.
```

3.3.5 Résultats expérimentaux

L'algorithme donné ci-dessus a aussi été réalisé en PROLOG. Le prototype est également testé sur les jeux d'essai ISCAS'85 [27]. Dans chaque expérience effectuée une porte ayant plus d'une entrée est sélectionnée aléatoirement, et une de ses entrées est déconnectée. L'algorithme de diagnostic est ensuite exécuté pour trouver l'erreur.

Les résultats obtenus sur une machine SPARC-10 à 128 Mega-octets de mémoire sont donnés dans le tableau 3.5 et le tableau 3.6 . Ces deux tableaux se lisent exactement comme les tableau 3.3 et le tableau 3.4.

Discussion :

Sous cette hypothèse d'une connexion manquante, nous trouvons aussi que le temps d'exécution augmente d'une façon linéaire avec le produit du nombre de portes par le nombre de vecteurs de test utilisés. Pour les mêmes raisons que celles décrites dans la section 3.2.5, nous trouvons que le nombre de vecteurs de test utilisés est relativement petit pour les circuits c2670, c5315, et c7552.

Dans presque tous les cas, nous avons pu localiser précisément la porte à laquelle la connexion manquante existe, et nous avons pu aussi identifier le nœud duquel cette connexion doit provenir. Dans le pire des cas, nous avons obtenu 7 candidats (dans un circuit de 1193 portes). Dans la plupart des cas, les différents candidats suggérés par le prototype, quand il y en a plusieurs, sont tous corrects; ils peuvent tous corriger la mauvaise implémentation.

Par exemple, la faute d'une connexion manquante à l'entrée d'une porte de type AND dans une suite de n portes de type AND peut être corrigée en connec-

tant la connexion manquante à l'entrée de n'importe laquelle de ces n portes. Ceci est illustré dans la figure 3.3. Dans ce cas, le système suggére n portes à l'entrée desquelles nous pouvons connecter la connexion manquante.

Dans d'autres cas, le système propose plusieurs connexions possibles à une seule porte. Ces cas se produisent quand plusieurs nœuds dans le circuit représentent la même fonction (par exemple, l'entrée et la sortie d'une porte de type BUF).

Circuit	Exp.	Moyenne de			Ecart type de		
		Vec.	CPU(sec)	Cand.	Vec.	CPU(sec)	Cand.
C432	119	12.34	61.31	1.45	5.94	58.34	0.63
C499	120	24.87	188.43	1.13	16.21	225.86	0.34
C880	290	12.24	69.17	1.52	6.08	52.38	0.95
C1355	237	23.80	513.90	1.20	13.81	525.39	0.49
C1908	139	23.91	736.50	1.65	13.22	662.09	0.68
C2670	215	11.60	330.98	1.23	4.99	210.99	0.51
C3540	25	17.92	2476.25	1.20	8.20	2437.78	0.50
C5315	98	13.26	738.26	1.06	12.97	1026.14	0.35
C6288	14	20.50	4982.62	1.00	8.96	4219.42	0.00
C7552	68	18.87	3629.89	1.25	13.98	3218.45	0.72

TAB. 3.5 - *Résultats des tests effectués sur les jeux d'essai ISCAS'85*

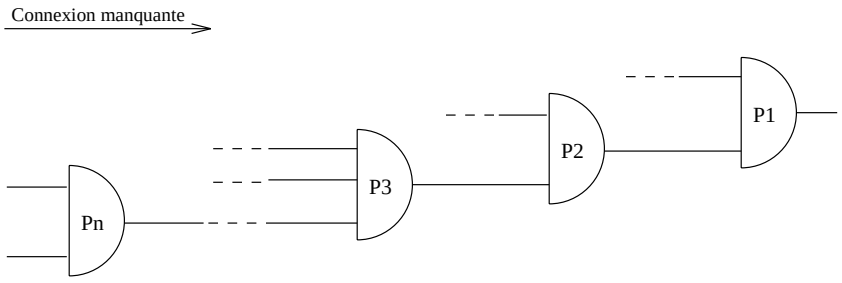

FIG. 3.3 - *En connectant la connexion manquante à P_1, P_2, ... ou P_n, nous obtenons la même fonction*

Circuit	Max. de			Min. de		
	Vec.	CPU(sec)	Cand.	Vec.	CPU(sec)	Cand.
C432	48	430.82	3	6	10.58	1
C499	66	1047.78	2	6	10.74	1
C880	48	371.40	4	3	8.83	1
C1355	69	3250.72	3	6	59.95	1
C1908	67	3371.46	4	6	71.90	1
C2670	35	1394.96	7	2	36.21	1
C3540	44	8656.94	3	7	609.43	1
C5315	72	6058.07	3	4	105.22	1
C6288	32	13280.99	1	3	220.52	1
C7552	83	15312.65	4	6	774.21	1

TAB. 3.6 - *Les extrêmes des résultats des tests effectués sur les jeux d'essai ISCAS'85*

Borne supérieure du temps de diagnostic :

Soit une implémentation contenant n portes. Chacune de ces portes peut avoir une connexion manquante provenant de la sortie de n'importe quelle autre porte. Dans le pire des cas, un vecteur de test généré selon la proposition 3.8 peut vérifier le manque d'une *seule* connexion parmi les n connexions possibles. Donc nous avons besoin de générer n vecteurs de test pour chaque porte. [2] Le nombre de vecteurs de test necéssaires pour le diagnostic est donc n^2, et l'algorithme *diagnostiquer-m-cnct* sera exécuté n^2 fois. Le temps de diagnostic est donc proportionnel à n^3 dans le pire des cas.

2. Ici nous surestimons le nombre de vecteurs de test, car il y a des portes d'où la connexion manquante ne peut pas provenir, faute de quoi l'implémentation corrigée ne sera plus combinatiore.

3.4 Diagnostic des fautes de connexions déplacées

Le diagnostic des fautes de connexions déplacées est plus compliqué que celui des fautes de connexions manquantes ou excédentaires. Supposons qu'une implémentation *IMPL* contient n portes, et que le nombre moyen d'entrées de chaque porte est k. Sous l'hypothèse d'une faute de connexion excédentaire, chacune de ces n portes pourrait avoir k connexions excédentaires différentes. Le nombre de fautes possibles est donc $O(k \times n)$. En pratique, k est un petit nombre (inférieur à 10). Dans le cas de fautes de connexions manquantes, chacune des n portes pourrait avoir une connexion manquante provenant de n'importe laquelle de ces n portes.[2] Donc, le nombre de fautes possibles est $O(n^2)$. Dans le cas de connexions déplacées, chaque entrée de chacune des n portes pourrait être remplacée par une connexion provenant de n'importe laquelle de ces n portes. Le nombre de fautes possibles est donc $O(k \times n^2)$.

3.4.1 Analyse avec des vecteurs de test détectant l'erreur

Proposition 3.9 :

Une porte P dans IMPL sous l'application d'un vecteur de test détectant l'erreur VT est suspecte d'avoir une connexion déplacée si et seulement si P est dans l'espace de recherche et si l'une des conditions suivantes est satisfaite:

1. *Sus(P,VT) = 0.*

2. *Sus(P,VT) = 1 et VRequise(P,VT)* $\neq X$ *et*
   $\exists i, i \in entrées(P), valeur(i) = V_{comp}(P, VCourante(P,VT)) \wedge$
   $\forall j \neq i, j \in entrées(P), valeur(j) = \overline{Forçante(P)}$

3. *Sus(P,VT) = 1 et VRequise(P,VT)* $= X$ *et*
   $\exists i, i \in entrées(P), valeur(i) = V_{comp}(P, VCourante(P,VT)) \wedge$
   $\forall j \neq i, j \in entrées(P), valeur(j) \neq$ Forçante(P)

Preuve :

- Si $Type(P)$ est AND ou NOR:
  - $Sus(P, VT) = 0$ seulement dans trois cas (voir la définition 3.1):
    1. $VRequise(P, VT) = 0$ et $VCourante(P, VT) = 1$
       Pour ces deux types de portes (AND et NOR), $Forcée(P) = VRequise(P, VT) = 0$. Le remplacement de n'importe quelle entrée de P par une connexion $C(P_1, P)$ où $valeur(P_1) = Forçante(P)$ produira $VRequise(P, VT)$ à la sortie de P. Donc, P est suspecte d'avoir une connexion déplacée à ses entrées.
    2. $VRequise(P, VT) = 0$ et $VCourante(P, VT) = X$
       Un raisonnement semblable à celui du cas précédent s'applique aussi dans ce cas.
    3. $VRequise(P, VT) = X$ et $VCourante(P, VT) = 1$
       Puisque $VCourante(P, VT) = 1$, toutes les entrées de P ont une valeur égale à 1 (resp. 0) si $Type(P)$ = AND (resp. NOR). Le remplacement de n'importe quelle entrée de P par une connexion $C(P_1, P)$ où $valeur(P_1) = X$ produira $VRequise(P, VT) = X$. Donc, P est suspecte d'avoir une faute de connexion déplacée.
  - $Sus(P, VT) = X$ seulement si $VRequise(P, VT) = VCourante(P, VT)$. Ceci signifie qu'il n'y a pas besoin de changer la valeur de sortie de P pour corriger l'implémentation. Par conséquent, P n'est pas suspecte.
  - $Sus(P, VT) = 1$ dans trois cas:
    1. $VRequise(P, VT) = 1$ et $VCourante(P, VT) = 0$ (c'est-à-dire $VRequise(P, VT) \neq X$)
       Pour générer $VRequise(P, VT)$ (qui est en fait $\overline{Forcée(P)}$ dans ce cas), toutes les entrées de P doivent être fixées à la valeur $\overline{\text{Forçante(P)}}$. Sous l'hypothèse d'une seule erreur, P est suspecte s'il est possible de générer $VRequise(P, VT)$ à sa sortie en remplaçant une seule de ses entrées par une autre connexion. P est donc suspecte si toutes ses entrées, exceptée une, sont égales à $\overline{\text{Forçante(P)}}$, tandis que l'entrée restante est égale à $V_{comp}(P, 0)$ (puisque $VCourante(P, VT) = 0$).

2. $VRequise(P, VT) = 1$ et $VCourante(P, VT) = X$ (c'est-à-dire $VRequise(P, VT) \neq X$)
   En utilisant le même raisonnement que dans le cas précédent, nous concluons que P est suspecte seulement si toutes ses entrées, exceptée une, sont égales à $\overline{\text{Forçante(P)}}$, tandis que l'entrée restante est égale à $V_{comp}(P, X)$ (car $VCourante(P, VT) = X$).
3. $VRequise(P, VT) = X$ et $VCourante(P, VT) = 0$
   Pour générer $VRequise(P, VT)$, toutes les entrées de P doivent être fixées à une valeur différente de $Forçante(P)$. En utilisant le même raisonnement que auparavant, nous concluons que P est suspecte seulement si toutes ses entrées, exceptée une, ont des valeurs différentes de *Forçante(P)* tandis que l'entrée restante est égale à $V_{comp}(P, 0)$ (car $VCourante(P, VT) = 0$).

Un raisonnement semblable peut être utilisé si $Type(P)$ est NAND ou OR. Si $Type(P)$ est NOT (resp. BUF), $VRequise(P, VT)$ peut toujours être obtenue en remplaçant l'entrée de P par une connexion provenant d'un nœud ayant la valeur $\overline{VRequise(P, VT)}$ (resp. $VRequise(P, VT)$), et donc P est toujours suspecte. Ce cas est couvert par la première condition de la proposition. $\diamond$

Si une porte $P \in IMPL$ est suspecte sous l'application d'un vecteur de test détectant l'erreur VT, il lui sera associés les deux ensembles $P_{mauvais}$ et P_{bon}.

1. $P_{mauvais}(P, VT)$ est un sous-ensemble d'entrées de P contenant l'entrée qui doit être remplacée pour corriger la valeur de sortie de l'implémentation.
2. $P_{bon}(P, VT)$ est un sous-ensemble de nœuds de l'implémentation qui pourraient être connectés à l'entrée de P, après la suppression de la connexion déplacée, pour corriger la valeur de sortie de l'implémentation.

Proposition 3.10 :
Si une porte $P \in$ IMPL *est suspecte d'avoir une connexion déplacée sous l'application d'un vecteur de test détectant l'erreur* VT*, alors:*

$$P_{mauvais}(P,VT) = \begin{cases} \{i \mid i \in entrées(P)\} & si\ \ Sus(P,VT) = 0 \\ \{i \mid i \in entrées(P) \wedge \\ \quad valeur(i) = V_{comp}(P, VCourante(P,VT))\} & sinon. \end{cases} \diamond$$

Preuve :

- $\underline{Sus(P,VT) = 0}$:
  $Sus(P,VT)$ est égale à 0 dans les trois cas suivants:
  1. $VRequise(P,VT) = Forcée(P)$, $VCourante(P,VT) = \overline{Forcée(P)}$:
     Dans ce cas, toutes les entrées de P auront une valeur égale à $\overline{Forçante(P)}$. Le remplacement de n'importe laquelle de ces entrées par une connexion ayant la valeur $Forçante(P)$ produira la valeur requise $RV(P,VT)$ à la sortie de P.
  2. $VRequise(P,VT) = Forcée(P)$, $VCourante(P,VT) = X$:
     Dans ce cas, toutes les entrées de P auront une valeur égale soit à $\overline{Forçante(P)}$ soit à X.Le remplacement de n'importe laquelle de ces entrées par une connexion ayant la valeur $Forçante(P)$ générera la valeur $VRequise(P,VT)$ à la sortie de P.
  3. $VRequise(P,VT) = X$, $VCourante(P,VT) = \overline{Forcée(P)}$:
     Dans ce cas, toutes les entrées de P auront une valeur égale à $\overline{Forçante(P)}$. Le remplacement de n'importe laquelle de ces entrées par une connexion ayant la valeur X produira $VRequise(P,VT)$ à la sortie de P.
- $\underline{Sus(P,VT) = X}$:
  Dans ce cas, la porte P n'est pas suspecte et l'ensemble $P_{mauvais}$ ne sera pas calculé.
- $\underline{Sus(P,VT) = 1}$:
  $Sus(P,VT)$ est égale à 1 dans les trois cas suivants:
  1. $VRequise(P,VT) = \overline{Forcée(P)}$, $VCourante(P,VT) = Forcée(P)$:
     Dans ce cas, une des entrées de P aura la valeur $Forçante(P)$. Pour générer $VRequise(P,VT)$ à la sortie de P, cette entrée doit

être enlevée. Il est à noter que *Forçante*(P) est compatible avec $VCourante(P, VT)$.

2. $VRequise(P, VT) = \overline{Forcée(P)}$, $VCourante(P, VT) = X$:
   Dans ce cas, une des entrées de P aura une valeur égale à X. Pour générer $VRequise(P, VT)$ à la sortie de P il faut enlever cette entrée. Il est à noter que X est compatible avec $VCourante(P, VT)$.
3. $VRequise(P, VT) = \overline{Forcée(P)}$, $VCourante(P, VT) = Forcée(P)$:
   Dans ce cas, une des entrées de P aura une valeur égale à *Forçante*(P). Pour générer $VRequise(P, VT)$ cette entrée doit être enlevée. Il est à noter que *Forçante*(P) est compatible avec $VCourante(P, VT)$. $\diamond$

Proposition 3.11 :

Soit P une porte dans l'implémentation IMPL, et soient Nœuds $= \{i \mid i \in IMPL \wedge i \notin$ successeur(P)$\}$. Si P est suspecte d'avoir une connexion déplacée sous l'application d'un vecteur de test détectant l'erreur VT, alors:

---

if $VRequise(P, VT) \neq X$ ***then***

$P_{bon} = \{j \mid j \in Nœuds \wedge valeur(j) = V_{comp}(P, VRequise(P, VT))\}$

elseif $\exists\ i \in$ entrées(P), valeur$(i) = X$ ***then***

$P_{bon} = \{j \mid j \in Nœuds \wedge valeur(j) \neq$ *Forçante(P)*$\}$

else

$P_{bon} = \{j \mid j \in Nœuds \wedge valeur(j) = X\}$. $\diamond$

Preuve :

- $VRequise(P, VT) \neq X$:
  Après la suppression de la connexion déplacée de l'entrée de la porte P, il existe deux possibilités: soit $VRequise(P, VT)$ est générée à la sortie de P et dans ce cas la nouvelle connexion ne doit pas changer la valeur de sortie de P, soit une autre valeur est générée à la sortie de P et dans ce cas la nouvelle connexion doit générer la valeur requise. Dans les deux cas, la valeur de la nouvelle connexion doit être compatible avec $VRequise(P, VT)$.

- $\underline{VRequise(P,VT) = X}$, et $\exists i \in entrées(P)$, $\underline{valeur(i) = X}$:
  Puisque P est suspecte, alors après la suppression de la connexion déplacée les entrées de P auront soit la valeur X soit la valeur $\overline{\text{Forçante(P)}}$. La nouvelle connexion doit avoir une valeur différente de $Forçante(P)$, sinon $Forcée(P)$ sera générée à la sortie de P.
- $\underline{VRequise(P,VT) = X}$, et $\forall i \in entrées(P)$, $\underline{valeur(i) \neq X}$:
  Si la valeur X doit être générée à la sortie de P, alors P doit avoir au moins une entrée ayant la valeur X. La nouvelle connexion doit donc avoir la valeur X. $\diamond$

Si une porte P est examinée sous l'application de plusieurs vecteurs de test détectant l'erreur $VT_1, VT_2, \cdots VT_n$, alors:

$$P_{bon}(P) = P_{bon}(P,VT_1) \cap P_{bon}(P,VT_2) \cap \cdots P_{bon}(P,VT_n)$$

$$P_{mauvais}(P) = P_{mauvais}(P,VT_1) \cap P_{mauvais}(P,VT_2) \cap \cdots P_{mauvais}(P,VT_n)$$

3.4.2 Analyse avec des vecteurs de test ne détectant pas l'erreur

Proposition 3.12 :

Si la valeur courante de la sortie d'une porte P, VCourante(P,VT)*, ne doit pas changer sous l'application d'un vecteur de test VT pour que la valeur correcte d'une sortie primaire reste correcte, alors l'ensemble des nœuds desquels la connexion déplacée ne peut pas provenir, $I_{bon}(P,VT)$, est calculé comme suit:*

---

if $VCourante(P,VT) = \overline{Forcée(P)}$ ***then***
  $I_{bon}(P,VT) = \{j \mid j \in IMPL \wedge valeur(j) \neq \overline{Forçante(P)}\}$
elseif $VCourante(P,VT) = Forcée(P)$ ***then***
  if $(\mid P_{mauvais}(P) \mid = 1) \wedge$
    $(\{i \mid i \in entrées(P) \wedge valeur(i) =$ Forçante(P) $\} = P_{mauvais}(P))$ ***then***
    $I_{bon}(P,VT) = \{j \mid j \in IMPL \wedge valeur(j) \neq Forçante(P)\}$
  else
    $I_{bon}(P,VT) = \phi$

endif

else

$I_{bon}(P, VT) = \phi$

endif. $\diamond$

Preuve :

- $\underline{VCourante(P, VT) = \overline{Forcée(P)}}$:
  Dans ce cas, toutes les entrées de P auront une valeur égale à $\overline{\text{Forçante(P)}}$. Si nous voulons garder la sortie de P inchangée après le remplacement d'une de ses entrées par la nouvelle connexion, alors la valeur de la nouvelle connexion doit être égale à $\overline{\text{Forçante(P)}}$. Les connexions qui ont d'autres valeurs que celle-ci ne peuvent pas être les bonnes connexions.

- $\underline{VCourante(P, VT) = Forcée(P)}$:
  Dans ce cas, une ou plusieurs entrées de P auront la valeur *Forçante(P)*. Si plus d'une entrée est égale à *Forçante(P)*, nous ne pouvons pas définir l'ensemble $I_{bon}(P, VT)$. Le remplacement d'une seule entrée de P par une autre connexion ne changera pas la sortie de P, parce que les autres entrées qui ont la valeur *Forçante(P)* entretiendront la sortie de P inchangée.

  D'ailleurs, si une seule entrée de P a la valeur *Forçante(P)*, et si cette entrée est la seule entrée suspecte d'être la mauvaise entrée (c'est-à-dire la seule entrée dans $P_{mauvais}(P)$), alors si cette entrée est à remplacer, il faut qu'elle soit remplacée par une connexion ayant la valeur *Forçante(P)*, sinon la valeur $Forcée(P)$ ne sera pas entretenue à la sortie de P. Donc, dans ce cas, tous les nœuds ayant une valeur autre que *Forçante(P)* ne peuvent pas être les sources des bonnes connexions.

3.4.3 L'Algorithme de diagnostic

L'algorithme de diagnostic des connexions déplacées, que nous présentons ici, est basé sur les propositions présentées ci-dessus. Etant donné un vecteur de test VT, l'algorithme parcourt le circuit de ses sorties primaires vers les entrées primaires. Ceci est fait par l'appel récursif à la procédure *analyser-b-cnct* décrite ci-dessous. Au départ, l'*espace de recherche* contient toutes les portes du circuit. L'algorithme *diagnostiquer-b-cnct* est ensuite exécuté à plusieurs reprises, sous l'application de vecteurs de test différents, jusqu'à ce que l'erreur soit trouvée, ou jusqu'à ce qu'il ne reste plus de vecteurs de test à générer.

---

algorithm diagnostiquer-b-cnct(VT);
begin
  Simuler *SPEC* et *IMPL* sous l'application de VT;
  for toute porte P_{y_i} dont la sortie est la sortie primaire $y_i \neq X$ **do**
    Nouvel Espace de Recherche $= \phi$;
    if $y_i \neq w_i$ **then**
      Type-Vct:= VDE; % *vecteur détectant l'erreur*
    else
      Type-Vct:= VNE; % *vecteur ne détectant pas l'erreur*
    endif
    analyser-b-cnct(*Type-Vct*,P_{y_i});
    if *Type-Vct* = VDE **then**
      Espace de Recherche = Nouvel Espace de Recherche;
    endif;
  endfor;
end.

---

procedure analyser-b-cnct(*Type-Vct*,P);
begin
  if P est une entrée primaire **then**
    exit
  endif;
  if *Type-Vct* = VDE **then**
    if P est suspecte **then** % proposition 3.9
      Nouvel Espace de Recherche = Nouvel Espace de Recherche $\cup \{P\}$;
      $Mauvais = P_{mauvais}(P, VT)$;
      $Bon = P_{bon}(P, VT)$;

```
            mettre-à-jour-b-cnct(Type-Vct,P,Mauvais,Bon,–);
            % '–' signifie un paramètre non utilisé.
        endif;
        for toute entrée i ∈ Nœuds-modifiables(P) do
            analyser-b-cnct(Type-Vct,i);
        endfor
    else
        Nœuds = I_bon(P, VT);
        % '–' signifie un paramètre non utilisé.
        mettre-à-jour-b-cnct(Type-Vct,P,–,–,Nœuds);
        for toute entrée i ∈ Entrées-fixes(P) do
            analyser-b-cnct(Type-Vct,i);
        endfor;
    endif;
end.
```

```
procedure mettre-à-jour-b-cnct(Type-Vct,P,Mauvais,Bon,Impossible);
begin
    if Type-Vct = VDE then
        if P est déjà examinée sous un VDE then
            P_mauvais(P) = P_mauvais(P) ∩ Mauvais;
            P_bon(P) = P_bon(P) ∩ Bon;
        elseif P est déjà examinée sous un VNE then
            P_mauvais(P) = Mauvais;
            P_bon(P) = Bon - I_bon(P);
        else                                  % la première fois que l'on examine P
            P_mauvais(P) = Mauvais;
            P_bon(P) = Bon;
        endif
    else        % Type-Vct = VNE
        if P est déjà examinée sous un VDE then
            P_bon(P) = P_bon(P) - Impossible;
        elseif P est déjà examinée sous un VNE then
            I_bon(P) = I_bon(P) + Impossible;
        else                                  % la première fois que l'on examine P
            I_bon(P) = Impossible;
        endif
    endif;
end.
```

3.4.4 Génération des vecteurs de test

Proposition 3.13 *:*

Un vecteur de test VT est capable de détecter si un nœud i est mal connecté à l'entrée d'une porte P à la place d'un autre nœud j si, quand VT est appliquée sur les entrées primaires de IMPL, valeur(i) est égale à $\overline{valeur(j)}$*, si toutes les autres entrées de P ont une valeur égale à* $\overline{Forçante(P)}$*, et si un chemin est sensibilisé de la sortie de P jusqu'au moins une des sorties primaires.* $\diamond$

Preuve :

Si toutes les entrées de P, exceptée une entrée k, sont fixées à la valeur $\overline{\text{Forçante(P)}}$, la sortie de P dépendra de *valeur(k)*. Si *valeur(k)* est égale à $\overline{\text{Forçante(P)}}$, la sortie de P aura la valeur $\overline{Forcée(P)}$. Si *valeur(k)* est égale à *Forçante(P)*, la sortie de P aura la valeur *Forcée(P)*. Donc si i et j sont fixées à deux valeurs binaires différentes, la sortie de P aura une valeur erronée, égale au complément de la valeur correcte, si i est connectée à P à la place de j. Si un chemin est sensibilisé de la sortie de P jusqu'à au moins une des sorties primaires, la valeur erronée sera propagée à travers ce chemin et l'erreur sera donc détectée. $\diamond$

Dans le prototype que nous avons réalisé, nous commençons par appliquer les vecteurs de test détectant l'erreur fournis par le vérificateur ou par n'importe quel autre moyen. Ceci diminue le nombre de portes suspectes et leurs ensembles P_{bon} et $P_{mauvais}$ associés. Pour chaque porte restante P, des vecteurs de test, capables de vérifier si P a de connexions déplacées, sont générés et l'algorithme de diagnostic est exécuté. Ces vecteurs de test sont aussi capables de détecter d'autres fautes de connexions déplacées tout au long du chemin sensibilisé car toute porte P se trouvant sur ce chemin a toutes ses entrées, exceptée une, fixées à la valeur $\overline{\text{Forçante(P)}}$. Cette démarche est répétée jusqu'à ce que l'erreur soit trouvée.

L'algorithme complet de diagnostic est donné dans ce qui suit:

```
algorithm b-cnct-diag;
begin
    Espace de Recherche = Toutes les portes du circuit;
    Testées ← φ;
    while (| Espace de Recherche | > 1) and (Espace de Recherche ⊈ Testées) do
        Soit P ∈ Espace de Recherche et P est plus proche des entrées primaires,
        et P ∉ Testées
        Testées ← Testées ∪ {P};
        for toute entrée suspecte i de P do
            VT = Vecteur détectant si i est mal connectée à P;      % proposition 3.13
            diagnostiquer-b-cnct(VT);
        endfor
    enddo
    for toute P ∈ Espace de Recherche do
        if P_mauvais(P) ≠ φ et P_bon(P) ≠ φ then
            write(P,P_mauvais(P),P_bon(P));
        endif
    endfor
end.
```

3.5 Résultats expérimentaux

L'algorithme précédent a été réalisé en PROLOG et testé sur les jeux d'essai ISCAS'85 [27]. Dans chaque expérience, une porte est sélectionnée aléatoirement et une de ses entrées est aussi sélectionnée aléatoirement et remplacée par une connexion provenant d'un autre nœud sélectionné également aléatoirement. L'algorithme de diagnostic est appliqué pour trouver l'erreur.

Les résultats obtenus sur une station de travail SPARC-10 à 128 Mega-octets de mémoire sont donnés dans le tableau 3.7 et le tableau 3.8 .

Discussion :

Comme dans les deux cas précédents (les erreurs de connexions excédentaires et les erreurs de connexions manquantes), le temps de diagnostic dans ce cas est également proportionnel aux produit du nombre de portes par le nombre de vecteurs de test utilisés. Dans presque tous les cas le nombre de candidats d'erreur proposés par l'algorithme est de 1. Dans le pire des cas nous avons eu 8 candidats dans un circuit de 1669 portes (c3540).

Circuit	Exp.	Moyenne de			Ecart type de		
		Vec.	CPU(sec)	Cand.	Vec.	CPU(sec)	Cand.
C432	119	27.29	114.84	1.01	38.57	241.62	0.09
C499	87	23.76	149.14	1.09	18.67	179.82	0.29
C880	306	17.26	72.98	1.00	15.25	93.09	0.06
C1355	278	46.18	491.20	1.20	40.93	476.52	0.46
C1908	75	23.31	793.50	1.44	19.41	1014.07	1.00
C2670	62	20.74	536.97	1.05	43.13	2497.94	0.38
C3540	42	25.00	2710.32	1.40	18.34	2504.96	1.33
C5315	36	26.81	2041.75	1.06	31.49	2956.36	0.33
C6288	24	31.38	6594.70	1.08	16.09	5855.37	0.28
C7552	55	10.84	928.57	1.00	5.60	645.26	0.00

TAB. 3.7 - *Résultats des tests effectués sur les jeux d'essai ISCAS'85*

Circuit	Max. de			Min. de		
	Vec.	CPU(sec)	Cand.	Vec.	CPU(sec)	Cand.
C432	314	2357.42	2	4	5.03	1
C499	124	1525.05	2	7	29.16	1
C880	211	1403.58	2	4	11.19	1
C1355	310	3311.99	4	6	74.70	1
C1908	117	5801.21	6	7	124.74	1
C2670	350	19852.17	4	4	33.08	1
C3540	110	12497.22	8	5	195.80	1
C5315	166	15198.60	3	8	354.39	1
C6288	67	23640.92	2	8	841.50	1
C7552	36	4251.38	4	3	216.57	1

TAB. 3.8 - *Les extrêmes des résultats des tests effectués sur les jeux d'essai ISCAS'85*

Le fait de trouver plusieurs candidats est normal car dans certaines situations de nombreuses corrections différentes peuvent exister. Prenons par exemple le circuit montré dans la figure 3.4. Si l'implémentation correcte doit réaliser la fonction $(a \cdot b \cdot c \cdot d)$ à la sortie y, alors 7 corrections sont possibles:

1. Remplacer la connexion $C(h, y)$ par $C(g, y)$.
2. Remplacer la connexion $C(h, y)$ par $C(i, y)$.
3. Remplacer la connexion $C(h, y)$ par $C(d, y)$.
4. Remplacer la connexion $C(b, y)$ par $C(g, y)$.
5. Remplacer la connexion $C(b, y)$ par $C(i, y)$.
6. Remplacer la connexion $C(b, y)$ par $C(d, y)$.
7. Remplacer la connexion $C(a, f)$ par $C(d, f)$.

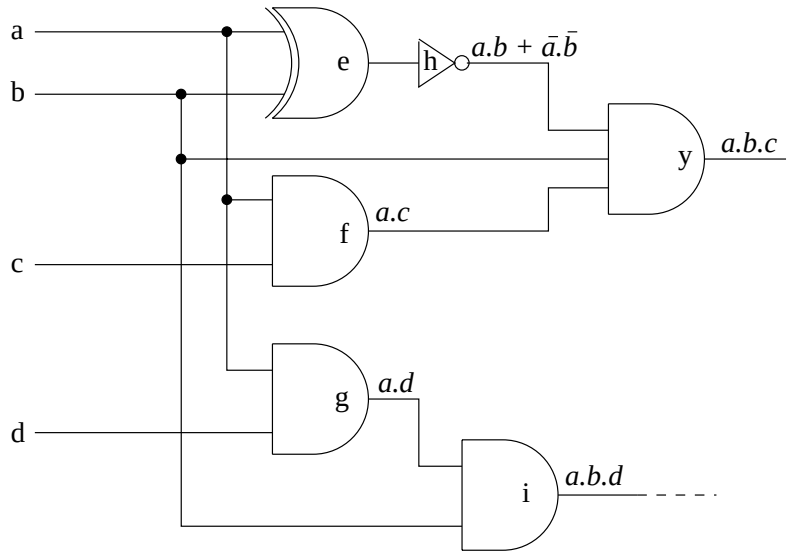

FIG. 3.4 - *7 corrections sont possibles pour obtenir la fonction* $(a \cdot b \cdot c \cdot d)$ *à la sortie* y.

Borne supérieure du temps de diagnostic :

Soit une implémentation contenant n portes, chacune ayant k entrées en moyenne. Chacune de ces k entrées peut provenir de la sortie de n'importe quelle

autre porte. [3] Dans le pire des cas, un vecteur de test généré selon la proposition 3.13 peut vérifier la correction d'une *seule* connexion. Donc, dans le pire des cas, l'algorithme *b-cnct-diag* générera $k \times n$ vecteurs de test pour chaque porte. Le nombre de vecteurs de test nécessaires pour le diagnostic est donc $k \times n^2$, et l'algorithme *diagnostiquer-b-cnct* sera exécuté $k \times n^2$ fois. Le temps de simulation d'un vecteur de test étant proportionnel à n, le temps de diagnostic est donc proportionnel à $k \times n^3$ dans le pire des cas.

3. Idem note précédente.

Chapitre 4

Diagnostic des circuits séquentiels

Nous présentons dans ce chapitre une méthode pour le diagnostic des erreurs de composants dans les circuits séquentiels. Le diagnostic de circuits séquentiels est plus difficile que le diagnostic de circuits combinatoires: la spécification peut avoir un nombre, et un codage, des variables d'états différents de ceux de l'implémentation. Nous ne connaissons donc pas les valeurs correctes auxquelles les valeurs des variables mémorisantes de l'implémentation peuvent être comparées. La spécification est donnée comme une boîte noire dont nous pouvons observer seulement les entrées et les sorties.

Pour surmonter cette difficulté nous avons introduit le nouveau concept d'*états suivants possibles*. Ce sont les états atteignables à partir d'un certain état, sous l'application d'un vecteur de test aux entrées primaires, par l'insertion d'une erreur aux différents emplacements possibles dans le circuit.

4.1 Préliminaires

Dans ce chapitre, nous considérerons des circuits séquentiels décrits au niveau logique et synchronisés par une horloge principale unique. Ces circuits sont modélisés par une machine d'états finis (voir Figure 4.1) de type Mealy, $M = (I,O,R,E_0,\delta,\lambda)$, où:

- I est l'ensemble des entrées primaires, $I = (I_1, I_2, ...I_n)$.

Un vecteur de test VT est une affectation de valeurs aux entrées primaires, c'est-à-dire un élément de $\mathcal{T}^n$.

- O est l'ensemble des sorties primaires, $O = (O_1, O_2, ...O_p)$.
- R est un ensemble d'éléments mémorisants, $R = (R_1, R_2, ...R_m)$.
  Un état E est un vecteur des valeurs de R, c'est-à-dire un élément de $\mathcal{T}^m$.
- $E_0 \in \mathcal{T}^m$ est l'état initial.
- δ: $\mathcal{T}^m \times \mathcal{T}^n \rightarrow \mathcal{T}^m$ est la fonction de transition qui calcule l'état suivant, $\delta = (\delta_1, \delta_2, ...\delta_m)$.
- λ: $\mathcal{T}^m \times \mathcal{T}^n \rightarrow \mathcal{T}^p$ est la fonction de sortie qui calcule la valeur courante des sorties primaires, $\lambda = (\lambda_1, \lambda_2, ...\lambda_p)$.

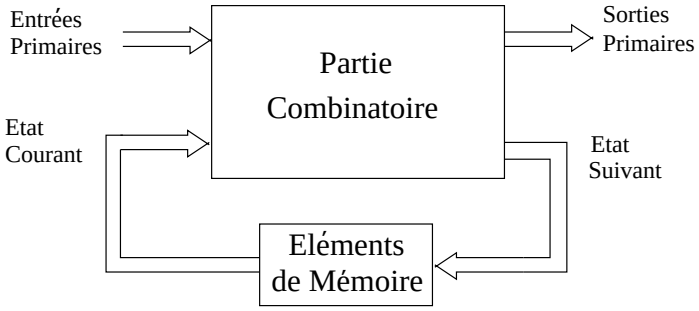

FIG. 4.1 - *Machine d'états finis*

Dans la suite, pour simplifier la description, nous parlerons de composants de la machine M pour signifier des composants du circuit modélisé par la machine M.

Le *modèle déplié* (*"iterative logic array model"* en anglais) des circuits logiques séquentiels a été défini dans [26]. Ce modèle a été défini pour les générateurs de vecteurs de test séquentiels destinés à détecter les fautes de fabrication. Ce modèle est donné dans la figure 4.2.

Dans ce modèle le fonctionnement séquentiel est représenté par duplication de la partie logique combinatoire du circuit. Les entrées VT^1, VT^2, ... VT^k représentent les vecteurs de test mis aux entrées primaires dans chaque intervalle de temps. EC^i $(1 \leq i \leq k)$ représentent les lignes d'état courant (c'est-à-dire les sorties d'éléments mémorisants). SP^i $(1 \leq i \leq k)$ sont les valeurs des sorties primaires à chaque intervalle de temps, et ES^i $(1 \leq i \leq k)$ sont les lignes

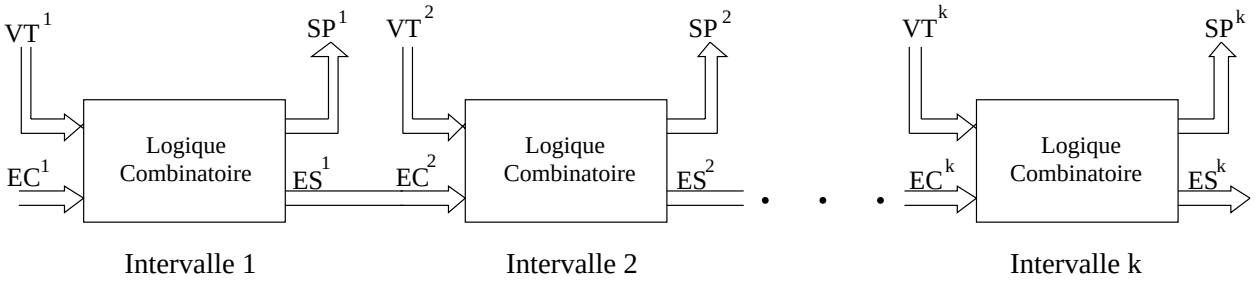

FIG. 4.2 - *Modèle déplié*

d'état suivant (c'est-à-dire les entrées d'éléments mémorisants). Les lignes d'état courant dans le premier intervalle de temps sont associées aux valeurs de l'état initial. Toutes les autres EC^i sont connectées aux ES^{i-1}.

La procédure de génération des vecteurs de test consiste en la génération d'une séquence $VT^1, VT^2, ... VT^k$ telle que l'erreur est détectée à SP^k, si cette séquence est appliquée au circuit en commençant par son état initial.

4.1.1 Les difficultés de diagnostic des circuits séquentiels

Dans le diagnostic des circuits combinatoires, le circuit est analysé sous l'application de vecteurs de test spécialement conçus pour le diagnostic. Sous l'hypothèse d'une seule erreur, il y aura dans le circuit un seul nœud duquel proviendra la valeur erronée. La plupart des algorithmes de diagnostic combinatoire sont basés sur ce fait.

Dans le cas des circuits séquentiels, au lieu d'employer des vecteurs de test, nous utilisons des *séquences de vecteurs de test*. Le circuit séquentiel est déployé dans le temps en répétant la partie combinatoire du circuit autant de fois qu'il y a de vecteurs dans la séquence de test appliquée, et en mettant ces parties en juxtaposition. Chaque partie représente le circuit dans un intervalle de temps différent (voir Figure 4.2).

La complexité supplémentaire dans ce cas est due à l'existence de plusieurs sources pour la valeur erronée: si chaque copie de la partie combinatoire est analysée séparément il y aura l'emplacement de l'erreur en plus des lignes d'état-présent auxquelles la valeur erronée a pu être propagée lors des intervalles de temps précédents. Si le circuit déployé est analysé en entier comme un circuit

combinatoire, l'erreur sera répétée dans chaque copie de la partie combinatoire et nous aurons un problème d'erreurs multiples. D'ailleurs, le circuit déployé peut être très grand, si la séquence de test appliquée est longue, et il est impossible de le stocker dans la mémoire de l'ordinateur.

4.1.2 Principes de la méthode

Nous introduisons ici le nouveau concept d'*etats suivants possibles*. Ce sont l'ensemble des états atteignables d'un certain état initial, ou d'un ensemble d'états initiaux, sous l'application d'un certain vecteur d'entrées, si une erreur est insérée aux différents emplacements possibles dans le circuit. Chaque emplacement possible engendre un *état suivant possible*.

L'implémentation du circuit séquentiel est représentée par son modèle de *déplié*. Le circuit est ensuite simulé séparément dans chaque intervalle de temps ("time frame"), et diagnostiqué en appliquant les règles et les algorithmes de diagnostic combinatoire. Les lignes d'état courant sont traitées comme des entrées primaires, et les lignes d'état suivant sont traitées comme des sorties primaires. Avant de poursuivre l'analyse du circuit dans l'intervalle de temps suivant, l'ensemble des états suivants possibles est calculé, puis l'analyse est faite dans l'intervalle de temps suivant sous l'application de chacun de ces états possibles. La même démarche est répétée dans tous les intervalles de temps suivants.

Dans notre travail sur les circuits séquentiels, les registres et les bascules sont des éléments de base. L'erreur peut se trouver seulement dans la partie combinatoire du circuit, tandis que les éléments mémorisants sont exempts d'erreur (pas d'erreur sur le nombre de ces éléments ni dans leurs structures internes).

Comme nous l'avons déjà précisé dans le deuxième chapitre de cette thèse, l'implémentation est décrite par un réseau de composants. La description de ces composants est donnée dans une bibliothèque de conception. L'erreur est due au remplacement d'un composant par un autre composant de type différent ayant le même nombre d'entrées.

4.2 Définitions et terminologie

Dans toutes les définitions données ci-après, E est l'état courant d'une machine M, et VT est un vecteur de test appliqué aux entrées primaires.

Définition 4.1 *: Fonction de transition modifiée.*
Soit C_1 un composant dans Espace-de-Recherche ayant le type T_1. La fonction $\delta_{C_1 \leftrightarrow C_2}$ est la fonction de transition de M modifiée par le remplacement de C_1 par un autre composant C_2 ayant un type T_2, où T_1 et T_2 appartiennent à la même classe de remplacement. $\diamond$

Définition 4.2 *: Etats suivants possibles esp(E,VT).*
L'ensemble des états suivants possibles d'un état E sous l'application d'un vecteur de test VT est l'ensemble de tous les états suivants dans lesquels M peut se trouver en remplaçant au plus un des composants C ∈ Espace-de-Recherche par un autre composant ayant le type T ∈ Ensemble-P(C). $\diamond$

$$esp(E,VT) = \delta(E,VT) \cup \{\delta_{C_1 \leftrightarrow C_2}(E,VT) \mid (C_1 \in \textit{Espace-de-Recherche}) \wedge (Type(C_2) \in \textit{Ensemble-P}(C_1))\}$$

Proposition 4.1 *:*
Si M a un Espace-de-Recherche contenant c composants, le nombre maximum d'états suivants possibles distincts est:

$$\min\{2c+1, 2^m\}$$

Preuve :

Le nombre total d'états dans lesquels la machine peut se trouver est 2^m (m est le nombre de variables mémorisantes). Sous l'application d'un vecteur de test ternaire, la sortie d'un composant peut assumer une des trois valeurs: 1, 0 ou X. Le remplacement d'un composant par un autre peut changer la valeur de sa sortie à l'une des deux autres valeurs possibles. Si cette nouvelle valeur peut se propager et affecter au moins une ligne d'état suivant, un nouvel état suivant sera obtenu. Donc, les différents remplacements possibles d'un certain composant peuvent générer, au pire, 2 états suivants différents.

La borne $2c+1$ représente donc les $2c$ états suivants possibles dus à tous les remplacements possibles des c composants de l'*Espace-de-Recherche* plus l'état suivant obtenu sans aucun remplacement. $\diamond$

Cette borne est une borne théorique. En pratique, plusieurs remplacements peuvent aboutir au même état suivant; cela dépend de la topologie du circuit. Dans les circuits arborescents, par exemple, cette borne est grandement réduite.

Proposition 4.2 *:*
Soit M une machine d'états finis à m variables d'état à un Espace-de-Recherche contenant c composants. Si l'implémentation de sa fonction de transition est arborescente, la borne supérieure du nombre d'états suivants possibles est :

$$\min\{2c+1, m+1\}$$

Preuve :

Dans les circuits arborescents, tous les composants situés dans le cône d'influence d'une ligne d'état suivant affectent seulement cette ligne. Donc, les états suivants possibles sont les m états correspondant, chacun, au changement de la valeur d'une seule ligne d'état suivant, plus l'état obtenu sans changer aucune valeur. La borne $2c+1$ est la même que celle de la proposition précédente. $\diamond$

Définition 4.3 *: Composants inversant une ligne d'état suivant.*
Soit ES_i une ligne d'état suivant d'une machine M, $1 \leq i \leq m$, et soit C_1 un composant de M. Un type T de composant est mis dans l'ensemble $Inversant(C_1, ES_i, E, VT)$ si $C_1 \in$ Espace-de-Recherche et le remplacement de C_1 par un autre composant C_2 de type $T \in$ Ensemble-P(C_1) inverse la valeur de ES_i sous l'application d'un vecteur de test VT quand la machine est dans l'état E.

$$Inversant(C_1, ES_i, E, VT) = \{T \mid (T \in \textit{Ensemble-P}(C_1)) \wedge (\delta_i(E, VT) = \overline{\delta_{i_{C_1 \leftrightarrow C_2}}}(E, VT)) \wedge (Type(C_2) = T\}$$

Définition 4.4 *: Lignes d'états suivants sensibles.*
Une ligne d'état suivant ES_i $(1 \leq i \leq m)$ est sensible au remplacement d'un composant C par un autre composant de type T si $T \in Inversant(C, ES_i, E, VT)$.

L'ensemble des lignes d'états suivants sensibles au remplacement d'un composant C par un autre composant de type T est noté Sensibles(C,T,E,VT).

$$Sensibles(C,T,E,VT) = \{ES_i \mid T \in Inversant(C, ES_i, E, VT) \wedge (1 \leq i \leq m)\}$$

Définition 4.5 *: Etat correspondant à un ensemble Sensibles(C,T,E,VT).*
$Etat(Sensibles(C,T,E,VT))$ est dit l'état correspondant à l'ensemble $Sensibles(C,T,E,VT)$ et il est calculé comme suit:

$$Etat(Sensibles(C,T,E,VT) = \begin{cases} \overline{\delta_i(E,VT)} & si\ ES_i \in Sensibles(C,T,E,VT) \\ \delta_i(E,VT) & sinon \end{cases}$$

où $1 \leq i \leq m$, et ES_i est la $i^{ème}$ ligne d'état suivant. $\diamond$

Exemple 4.1 *:*

Soit une machine M ayant 5 lignes d'état suivant (c'est-à-dire 5 variables d'état). Soient les valeurs de ces lignes, quand M est dans l'état E sous l'applicatoin d'un vecteur de test VT, sont $(es_1, es_2, es_3, es_4, es_5)$ = (1,0,0,1,0). C'est-à-dire $\delta(E,VT)$ = (1,0,0,1,0). Si Sensibles(C,T,E,VT) = (es_2, es_4, es_5), alors Etat(Sensibles(C,T,E,VT)) = (1,1,0,0,1).

4.3 Calcul d'états suivants possibles

Nos algorithmes de diagnostic sont basés sur le fait qu'une seule erreur existe dans le circuit. Quand le circuit séquentiel est déployé dans le temps et examiné dans chaque intervalle de temps, il peut y avoir des cas où il existe simultanément plusieurs sources pour la valeur erronée . Par exemple si, dans l'intervalle de temps i, la valeur erronée arrive à se propager jusqu'aux lignes d'états suivants,

il y aura plusieurs sources de la valeur erronée dans l'intervalle de temp $i+1$: le composant erroné et les lignes d'état courant. Donc, le diagnostic fait dans l'intervalle de temps $i+1$ pour corriger les sorties primaires peut être faux.

C'est pour cette raison que le diagnostic doit être effectué dans tous les *états suivants possibles*. Un de ces états est correct, et donc le diagnostic sera fait correctement.

4.3.1 États suivants possibles atteignables d'un seul état

Nous présentons ici une méthode pour trouver l'ensemble des états suivants possibles d'un circuit séquentiel à partir d'un état E et sous l'application d'un vecteur de test VT.

Une méthode directe consiste à changer successivement chaque composant $C \in$ *Espace-de-Recherche* par un autre composant de type $T \in$ *Ensemble-P(C)*, et à simuler le circuit après chaque changement pour obtenir l'état suivant. L'ensemble des états distincts obtenu est l'ensemble d'états suivants possibles *esp(E,VT)*. Cette méthode est très coûteuse par rapport au temps de calcul: si le circuit a NC composants, si c d'entre eux sont dans l'*Espace-de-Recherche*, et si chacun de ces c composants a un *Ensemble-P* contenant t types, le temps de calcul sera proportionnel à $NC \times c \times t$.

Une méthode plus efficace permet d'exploiter les algorithmes de diagnostic des circuits combinatoires présentés dans le Chapitre 2. Le circuit est traité comme un circuit combinatoire (voir figure 4.2), et le calcul est fait selon les trois étapes suivantes:

Étape 1: Calcul des ensembles $Inversant(C, ES_i, E, VT)$, $1 \leq i \leq m$.

Le circuit est mis dans l'état E et simulé une seule fois, sous l'application du vecteur de test VT. Chaque ligne d'état suivant ES_i est ensuite traitée séparément comme étant une sortie erronée, et les algorithmes de diagnostic combinatoire sont appliqués pour obtenir, pour chaque composant C affectant une ligne ES_i, l'ensemble des types qui peuvent inverser la valeur de ES_i en remplaçant le type de C par un autre type de cet ensemble. Cette étape génère donc les ensembles $Inversant(C, ES_i, E, VT)$.

Le temps nécessaire pour trouver $Inversant(C, ES_i, E, VT)$ pour toutes

les ES_i est proportionnel au nombre de composants du circuit. Donc, cette étape est exécutée dans un temps proportionnel à $2 \times NC$.

Étape 2: Calcul des lignes d'états suivants sensibles.

Le but de cette étape est de trouver les ensembles de lignes d'état suivant sensibles (voir la définition 4.4) en utilisant le résultat de l'étape précédente. Ceci est fait en utilisant la formule donnée dans la définition 4.4.

Étape 3: Calcul des états suivants possibles.

Le premier état suivant possible est $\delta(E, VT)$. Les autres états suivants possibles sont ceux qui correspondent aux ensembles *Sensibles* calculés dans l'étape précédente. Ces états sont calculés en utilisant la formule donnée dans la définition 4.5.

Exemple 4.2 *:*

Le circuit donné par la Figure 4.3 est un des circuits des jeux d'essai ISCAS'89 [28]. Il a quatre entrées (x_0, x_1, x_2, x_3)*, une sortie primaire* z*, trois lignes d'état suivant* (y_1, y_2, y_3)*, et trois lignes d'état courant* (Y_1, Y_2, Y_3)*.*

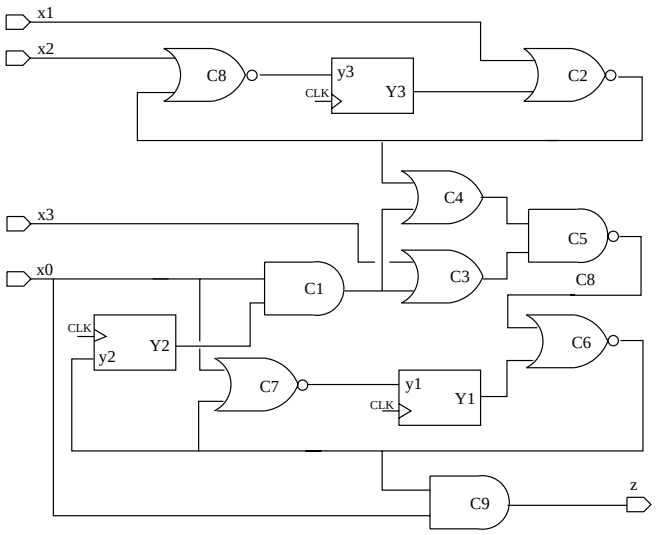

FIG. 4.3 - *Circuit s27 des jeux d'essai ISCAS'89*

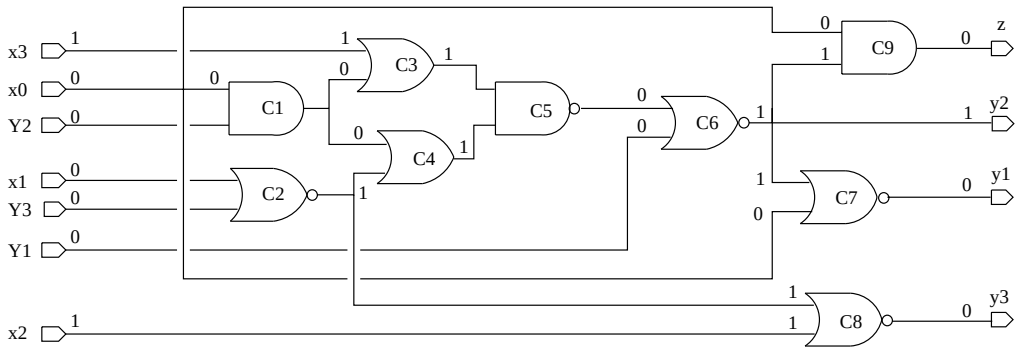

FIG. 4.4 - *Vue combinatoire du circuit s27*

Supposons que l'état courant E soit $(Y_1, Y_2, Y_3) = (0, 0, 0)$*, et que le vecteur d'entrées VT soit* $(x_0, x_1, x_2, x_3) = (0, 0, 1, 1)$*. Pour faciliter la présentation, nous allons considérer une bibliothèque de conception contenant des composants de types simples: AND, OR, NAND et NOR, tous à deux entrées. Toutefois, cela ne change rien dans la méthode car ce qui s'applique ici s'applique aussi quand il y a des composants plus complexes. Supposons qu'au départ l'espace de recherche contienne tous les composants du circuit et que l'Ensemble-P de chacun de ces composants contienne tous les types de la bibliothèque de conception.*

La vue combinatoire du circuit est donnée par la Figure 4.4.

Résultat de l'étape 1 *:*

Chaque case de le tableau suivant est un des ensembles "Inversant". Par exemple, la case de la première colonne et de la deuxième ligne est l'ensemble $Inversant(C_6, y_1, E, VT)$*.*

Pour inverser y_1 *remplacer*	*Pour inverser* y_2 *remplacer*	*Pour inverser* y_3 *remplacer*
C_7 *par* {*OR,NAND*}	C_6 *par* {*AND,OR*}	C_8 *par* {*AND,OR*}
C_6 *par* {*AND,OR*}	C_5 *par* {*AND,OR*}	
C_5 *par* {*AND,OR*}	C_4 *par* {*AND,NOR*}	
C_4 *par* {*AND,NOR*}	C_3 *par* {*AND,NOR*}	
C_3 *par* {*AND,NOR*}	C_2 *par* {*AND,OR*}	
C_2 *par* {*AND,OR*}		

Résultat de l'étape 2 *:*

Dans cette étape, nous extrayons les ensembles Sensibles *à partir des en-*

sembles calculés dans l'étape précédente. Les résultats suivants sont obtenus:

$Sensibles(C_7, OR, E, VT) = \{y_1\}$ $Sensibles(C_7, NAND, E, VT) = \{y_1\}$
$Sensibles(C_6, AND, E, VT) = \{y_1, y_2\}$ $Sensibles(C_6, OR, E, VT) = \{y_1, y_2\}$
$Sensibles(C_5, AND, E, VT) = \{y_1, y_2\}$ $Sensibles(C_5, OR, E, VT) = \{y_1, y_2\}$
$Sensibles(C_4, AND, E, VT) = \{y_1, y_2\}$ $Sensibles(C_4, NOR, E, VT) = \{y_1, y_2\}$
$Sensibles(C_3, AND, E, VT) = \{y_1, y_2\}$ $Sensibles(C_3, NOR, E, VT) = \{y_1, y_2\}$
$Sensibles(C_2, AND, E, VT) = \{y_1, y_2\}$ $Sensibles(C_2, OR, E, VT) = \{y_1, y_2\}$
$Sensibles(C_8, AND, E, VT) = \{y_3\}$ $Sensibles(C_8, OR, E, VT) = \{y_3\}$

Parmi ces ensembles, trois seulement sont distingués: $\{y_1\}$, $\{y_1, y_2\}$ *et* $\{y_3\}$.

Résultat de l'étape 3 *: États suivants possibles.*

Les états suivants possibles sont l'état $\delta((0,0,0),(0,0,1,1))$*, plus trois autres états correspondant aux trois ensembles obtenus dans l'étape précédente. Chacun de ces trois états est obtenu en inversant dans* $\delta((0,0,0),(0,0,1,1))$ *les bits qui existent dans les ensembles Sensibles calclulés ci-dessus.*

$\delta((0,0,0),(0,0,1,1)) = (0,1,0)$.
En inversant y_1 *nous obtenons l'état* $(1,1,0)$.
En inversant (y_1, y_2) *nous obtenons l'état* $(1,0,0)$.
En inversant (y_3) *nous obtenons l'état* $(0,1,1)$.

Alors, l'ensemble détats-suivants possibles $esp((0,0,0),(0,0,1,1))$ *est donné par:*

$$esp((0,0,0),(0,0,1,1)) = \{(0,1,0),(1,1,0),(1,0,0),(0,1,1)\}$$

4.3.2 Analyse dans un état suivant possible

Comme nous l'avons déjà dit, après le calcul des états suivants possibles, un autre vecteur de test est appliqué sur les entrées primaires et les algorithmes de diagnostic sont appliqués dans chacun de ces états possibles. Quand le diagnostic est fait dans un état E, l'espace de recherche se restreint aux composants qui mènent à cet état en les remplaçant par d'autres composants d'autres types. Cet espace de recherche restreint est associé à cet état. Par exemple, l'espace

de recherche restreint de l'état $(1,1,0)$, obtenu en inversant y_1 dans l'exemple précédent, contient un seul composant $\{C_7\}$. Si un autre composant est remplacé, la machine ne peut pas être dans l'état $(1,1,0)$. Chaque composant C dans cet espace de recherche restreint aura aussi un *Ensemple-P-Restreint* qui est un sous-ensemble de l'*Ensemble-P(C)* et qui contient les types de composants qui peuvent mener à létat E si C est remplacé par un autre composant ayant un de ces types. Par exemple, le composant C_7 dans l'exemple précédent a un *Ensemble-P*(C_7) = {AND,OR,NAND,NOR}, mais dans l'état suivant possible $(1,1,0)$ il aura un *Ensemble-P-Restreint* contenant les types {OR,NAND}.

Définition 4.6 *: L'Ensemble-P-Restreint.*
Soit E_2 un état suivant possible atteignable d'un état E_1 sous l'application d'un vecteur de test VT, et soit S l'espace de recherche restreint quand la machine est dans l'état E_1. Chaque composant $C_1 \in S$ a un Ensemble-P-Restreint$(C_1, E_2) \subseteq$ Ensemble-P-Restreint(C_1, E_1) contenant les types de composants qui peuvent mener à l'état E_2 en remplaçant C_1 par un autre composant C_2 ayant un de ces types.

$$\begin{aligned} \textit{Ensemble-P-Restreint}(C_1, E_2) = \{T \mid\ & (T \in \textit{Ensemble-P-Restreint}(C_1, E_1)) \wedge \\ & (\delta_{C_1 \leftrightarrow C_2}(E_1, VT) = E_2) \wedge \\ & (Type(C_2) \in \textit{Ensemple-P-Restreint}(C_1, E_1)\} \end{aligned}$$

Définition 4.7 *: L'espace de recherche restreint, EDR-Restreint.*
Commençant dans un état courant E_1, et sous l'application d'un vecteur de test VT*, l'espace de recherche restreint, EDR-Restreint(E_2), d'un état $E_2 \in esp(E_1, VT)$ est un sous-ensemble de EDR-Restreint(E_1) tel que le remplacement d'un de ses composants par un autre composant de type différent mène à l'état E_2.*

$$\begin{aligned} \textit{EDR-Restreint}(E_2) = \{C_1 \mid\ & (C_1 \in \textit{EDR-Restreint}(E_1)) \wedge \\ & (\delta_{C_1 \leftrightarrow C_2}(E_1, VT) = E_2) \wedge \\ & (Type(C_2) \in \textit{Ensemple-P-Restreint}(C_1, E_1)\} \end{aligned}$$

Proposition 4.3 *:*
L'Ensemble-P-Restreint d'un composant C dans un état suivant possible E_2 attei-

gnable d'un état E_1 sous l'application d'un vecteur de test VT, où $E_2 \neq \delta(E_1, VT)$, est calculé comme suit:

$$\textit{Ensemble-P-Restreint}(C, E_2) = \{T \mid (Etat(Sensibles(C, T, E_1, VT)) = E_2) \wedge (T \in \textit{Ensemble-P-Restreint}(C, E_1))\}$$

et

$$\textit{Ensemble-P-Restreint}(C, \delta(E_1, VT)) = \textit{Ensemble-P-Restreint}(C, E_1) - \bigcup_{E_2 \in esp(E_1, VT)} \textit{Ensemble-P-Restreint}(C, E_2)$$

Preuve :

$Etat(Sensibles(C, T, E_1, VT)$ est l'état atteignable à partir de E_1 sous l'application de VT en remplaçant C par un autre composant de type T. Donc T est un membre de l'*Ensemble-P-Restreint*(C, E_2). Cela explique la première formule de la proposition.

Si le remplacement d'un composant C par un autre ayant le type T ne change aucune ligne d'état suivant, alors l'état suivant est $\delta(E_1, VT)$ et ce type doit être mis dans l'*Ensemble-P-Restreint*$(C, \delta(E_1, VT))$. Un tel type n'existe dans aucun *Ensemble-P-Restreint*(C, E_2) où $E_2 \neq \delta(E_1, VT)$. Cela explique la deuxième formule de la proposition. $\diamond$

Proposition 4.4 *:*
L'espace de recherche restreint, EDR-Restreint(E_2), d'un état suivant possible E_2 atteignable d'un état E_1 sous l'application d'un vecteur de test VT est calculé comme suit:

$$\textit{EDR-Restreint}(E_2) = \{C \mid \textit{Ensemble-P-Restreint}(C, E_2) \neq \phi\}$$

Preuve :

Si l'*Ensemble-P-Restreint*(C, E_2) est vide, alors cela signifie que E_2 ne peut

pas être atteint en remplaçant le composant C. Donc, le composant C n'est pas le composant erroné si la machine est dans l'état E_2, et par conséquent *EDR-Restreint*(E_2) ne doit pas contenir le composant C. $\diamond$

Exemple 4.3 *:*
Dans l'exemple précédent, le calcul de l'Espace-de-Recherche-Restreint pour chaque état suivant possible donne le résultat suivant:

- *Dans l'état* $(1,1,0)$*:*
  Espace-de-Recherche-Restreint$((1,1,0)) = \{C_7\}$
  Ensemble-P-Restreint$(C_7,(1,1,0)) = \{OR, NAND\}$

- *Dans l'état* $(1,0,0)$*:*
  Espace-de-Recherche-Restreint$((1,0,0)) = \{C_6, C_5, C_4, C_3, C_2\}$
  Ensemble-P-Restreint$(C_6,(1,0,0)) = \{OR, AND\}$
  Ensemble-P-Restreint$(C_5,(1,0,0)) = \{OR, AND\}$
  Ensemble-P-Restreint$(C_4,(1,0,0)) = \{NOR, AND\}$
  Ensemble-P-Restreint$(C_3,(1,0,0)) = \{NOR, AND\}$
  Ensemble-P-Restreint$(C_2,(1,0,0)) = \{OR, AND\}$

- *Dans l'état* $(0,1,1)$*:*
  Espace-de-Recherche-Restreint$((0,1,1)) = \{C_8\}$
  Ensemble-P-Restreint$(C_8,(0,1,1)) = \{OR, AND\}$

4.3.3 États suivants possibles atteignables d'un ensemble d'états courants

Les définitions et les propositions présentées dans la section précédente s'appliquent quand il s'agit d'un seul état à partir duquel l'ensemble d'états suivants possibles sont calculés. Cela convient très bien dans le premier intervalle de temps, où les états suivants possibles (ESP^1) sont calculés à partir de l'état initial. Dans le deuxième intervalle de temps, nous allons aussi calculer l'ensemble des états

suivants possibles (ESP^2) atteignables à partir de ESP^1. De manière plus générale, dans le $i^{ème}$ intervalle de temps, nous aurons besoin de calculer l'ensemble ESP^i à partir de l'ensemble ESP^{i-1}.

Définition 4.8 *:*
Soit $ESP^{i-1} = \{E_1, E_2, \ldots E_q\}$ l'ensemble des états suivants possibles dans l'intervalle de temps $i-1$, et soit VT un vecteur de test appliqué à l'intervalle de temps i. L'ensemble des états suivants possibles dans l'intervalle de temps i est donné par:

$$ESP^i = \bigcup_{j=1}^{q} esp(E_j, VT)$$

Dans ce qui suit, nous appelons un état $E_j \in ESP^{i-1}$ un ***prédécesseur possible*** *d'un autre état $D \in ESP^i$ si $D \in ESP(E_j, VT)$. Le calcul de l'EDR-Restreint et de l'Ensemble-P-Restreint est étendu comme suit:*

$$\textit{Ensemble-P-Restreint}(C_1, D) = \bigcup_{j=1}^{q} \{T \mid (T \in \textit{Ensemble-P-Restreint}(C_1, E_j)) \wedge (\delta_{C_1 \leftrightarrow C_2}(E_j, VT) = D) \wedge (Type(C_2) = T)\}$$

$$\textit{EDR-Restreint}(D) = \bigcup_{j=1}^{q} \{C_1 \mid (C_1 \in \textit{EDR-Restreint}(E_j)) \wedge (\delta_{C_1 \leftrightarrow C_2}(E_j, VT) = D) \wedge (Type(C_2) \in \textit{Ensemble-P-Restreint}(C_1, E_j)\}$$

Proposition 4.5 *:*
Dans n'importe quel intervalle de temps i, l'état suivant correct, dans lequel la machine doit se trouver pour se comporter correctement, est un membre de l'ensemble des états suivants possibles ESP^i. $\diamond$

Preuve par induction sur les intervalles de temps :

ESP^1 est calculé sous l'application de l'état initial qui est sûr d'être correct. ESP^1 contient les états qui peuvent être atteints en remplaçant *un seul composant au plus*. Il existe deux possibilités:

1. L'erreur n'affecte pas les lignes d'états suivants, et dans ce cas l'état suivant correct est celui obtenu sans remplacer aucun composant.

2. L'erreur affecte les lignes d'états suivants, et donc l'état suivant correct sera obtenu en remplaçant l'un des composants, à savoir le composant erroné.

Supposons que l'état correct existe dans ESP^{i-1}; le même raisonnement conclut que ESP^i contient l'état correct. $\diamond$

4.4 L'algorithme de diagnostic séquentiel

Le diagnostic commence par appliquer une séquence de test qui détecte l'erreur. Cette séquence peut être le contre-exemple généré par un vérificateur ou par n'importe quel autre moyen.

Sous l'application d'un vecteur de test VT et avec un état courant E, l'algorithme effectue deux fonctions principales:

1. Simuler l'implémentation et la spécification, comparer les valeurs de leurs sorties et appliquer les algorithmes de diagnostic combinatoire pour corriger les valeurs des sorties erronées ou garder les valeurs correctes des sorties correctes. Cette étape permet de diminuer l'espace de recherche *EDR-Restreint(E)* et les ensembles *Ensemble-P-Restreint(C,E)* de ses composants.

2. Trouver les états suivants possibles atteignables de E et calculer pour chacun de ces états l'espace de recherche restreint et les *Ensemble-P-Restreint*.

Avant d'appliquer l'algorithme de diagnostic, l'*Espace-de-Recherche* contient tous les composants du circuit et l'*Ensemble-P(C)* de chacun de ses composants, C, contient tous les types de composant qui sont dans la même classe de remplacement, *Classe-R*, que le type de C.

Si la première séquence de test contient k vecteurs, alors après l'analyse du circuit dans l'intervalle de temps k un nouvel *Espace-de-Recherche* est calculé et un nouvel *Ensemble-P(C)* est calculé pour chacun de ses composants:

$$\textit{Espace-de-Recherche} = \bigcup_{E \in ESP^{k-1}} \textit{EDR-Restreint(E)}$$

$$\textit{Ensemble-P(C)} = \bigcup_{E \in ESP^{k-1}} \textit{Ensemble-P-Restreint(C,E)}$$

Si l'*Espace-de-Recherche* contient plusieurs composants ou un composant ayant plusieurs éléments dans son *Ensemble-P*, des séquences de test supplémentaires sont exigées. Un composant C est sélectionné dans l'*Espace-de-Recherche*, un type T est sélectionné dans son *Ensemble-P*, et une séquence de test capable de détecter le remplacement de C par un autre composant de type T est générée. La spécification et l'implémentation sont ensuite simulées sous l'application de cette séquence, et les valeurs des sorties primaires sont comparées. Si l'erreur n'est pas détectée, alors T n'est pas le type correct de C parce que la séquence de test utilisée est capable de détecter le remplacement de C par un autre composant de type T. T est donc enlevé de l'*Ensemble-P*(C). Si l'erreur est détectée, nous ne pouvons rien décider en ce qui concerne le type T. Dans les deux cas, l'algorithme de diagnostic est ensuite exécuté exactement comme dans la première séquence de test.

La même procédure est répétée jusqu'à ce qu'il ne reste aucun composant dans l'espace de recherche, ce qui signifie que l'erreur n'est pas due au remplacement de composants, ou jusqu'à ce que des séquences de test soient générés pour tous les composants restants.

Nous donnons ici le pseudo-code de notre algorithme de diagnostic que nous appelons *diagnostic-composant-séquentiel*. Cet algorithme prend comme entrée la séquence de test S fournie par le vérificateur comme contre-exemple.

---

algorithm diagnostic-composant-séquentiel(S);
begin
  Testé $\leftarrow \phi$;
  Espace-de-Recherche $\leftarrow$ tous les composants dans *IMPL*;
  EDR-Restreint(Etat_initial) $\leftarrow$ *Espace-de-Recherche*;
  for tout composant $C \in$ *Espace-de-Recherche* **do**
    $T = type(C)$;
    Ensemble-P(C) $= \{T_1 \mid (T_1 \in$ classe de remplacement de $T) \wedge (T \neq T_1)\ \}$;
    Ensemble-P-Restreint(C,Etat_initial) = *Ensemble-P(C)*;

endfor;
Etats := {*Etat_initial*};
diagnostiquer_séquentiel(*S*,*Etats*);
while (|*Espace-de-Recherche*| > 1) **and** (*Espace-de-Recherche* $\not\subseteq$ *Testé*) **do**
Soit C_e un composant dans *Espace-de-Recherche* à la plus grande sortance, et $C_e \notin$ *Testé*
Testé $\leftarrow$ *Testé* $\cup \{C_e\}$;
for tout $T \in$ *Ensemble-P*(C_e) **do**
générer une séquence détectant le remplacement de C_e par un autre
composant de type T, S_d;
Simuler *SPEC* et *IMPL* sous l'application de S_d;
if $\forall i, 1 \leq i \leq m, y_i = w_i$ **then**;
Ensemble-P(C_e) = *Ensemble-P*(C_e) $- T$;
endif;
EDR-Restreint(Etat_initial) := *Espace-de-Recherche*;
for tout composant $C \in$ *EDR-Restreint(Etat_initial)* **do**
Ensemble-P-Restreint(C,Etat_initial) = *Ensemble-P(C)*;
endfor;
Etats := {*Etat_initial*};
diagnostiquer_séquentiel(S_d,*Etats*)
endfor
endwhile
% Ici l'*Espace-de-Recherche* contient un ou plusieurs composants, parmi lesquels se trouve
% le composant erroné.
for tout composant $C \in$ *Espace-de-Recherche* **do**
print(*C*,*Ensemble-P(C)*);
endfor
end.

---

procedure diagnostic-séquentiel(*Séquence,Etats*);
begin
longueur := Nombre de vecteurs de test dans *Séquence*;
Simuler SPEC sous l'application de *Séquence*;
for $i := 1$ **to** *longueur* **do**
W^i := Valeurs des sorties de SPEC dans l'intervalle de temps i;
endfor;
for $i := 1$ **to** *longueur* **do**
for tout $E \in$ *Etats* **do**
Espace := *EDR-Restreint(E)*;
Tableau := Tableau d'*Ensemble-P-Restreint(C,E)*, $\forall C \in$ *Espace*;
Appliquer VT^i sur les entrées primaires d'IMPL;

Appliquer E sur les lignes d'état courant d'IMPL;
Simuler IMPL;
$Y^i :=$ Valeurs des sorties d'IMPL;
diagnostic-combinatoire(VT^i,*Espace*, *Tableau*,Y^i,W^i);
Calculer les états suivants possibles: $esp(E, VT^i)$;
Calculer pour chaque composant C, l'*Ensemble-P-Restreint*(C, E_p), $\forall E_p \in esp(E, VT^i)$;
Calculer *EDR-Restreint*(E_p), $\forall E_p \in esp(E, VT^i)$;
endfor;
Calculer ESP^i; % définition 4.8
Calculer pour chaque composant C, l'*Ensemble-P-Restreint*(C, E_p), $\forall E_p \in ESP^i$; % définition 4.8
Calculer *EDR-Restreint*(E_p), $\forall E_p \in ESP^i$; % définition 4.8
Etats := ESP^i;
endfor;
Espace-de-Recherche = $\bigcup_{E \in ESP^{longueur}}$ *EDR-Restreint(E)*;
for tout composant $C \in$ *Espace-de-Recherche* **do**
Ensemble-P(C) = $\bigcup_{E \in ESP^{longueur}}$ *Ensemble-P-Restreint(C,E)*
endfor;
end.

Exemple 4.4 *:*

Supposons que dans le circuit de l'exemple 4.2, le composant C7 soit erroné et qu'il faille le remplacer par un autre composant de type NAND. Une séquence de test qui détecte l'erreur est $(x0, x1, x2, x3) = ((0, 0, 1, 1), (1, 0, 0, 1))$. *En commençant par l'état initial* $(Y0, Y1, Y2)$ = (0,0,0), *et sous l'application de cette séquence de test, l'erreur est détectée dans le deuxième intervalle de temps. Après la simulation dans le premier intervalle de temps, sous l'application du vecteur* $(0, 0, 1, 1)$, *la valeur de la sortie z est égale à* 0 *dans l'implémentation et dans la spécification. Après la simulation dans le deuxième intervalle de temps, sous l'application du vecteur* $(1, 0, 0, 1)$, *la valeur de sortie de la spécification est* 0 *tandis que celle de l'implémentation est* 1. *Nous montrons ici comment l'algorithme de diagnostic est appliqué.*

Premier intervalle de temps :

Au départ, nous ne connaissons rien sur l'erreur et donc l'Espace-de-Recherche contient tous les composants du circuit, et l'Ensemple-P de

chacun de ces composants contient tous les types possibles. Nous commençons par appliquer l'état initial $(0,0,0)$ *sur les lignes d'état courant, et le vecteur* $(0,0,1,1)$ *sur les entrées primaires. EDR-Restreint*$((0,0,0))$ = Espace-de-Recherche = $\{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9\}$ *et l'Ensemble-P-Restreint*$(C,(0,0,0))$ = *Ensemble-P(C),* $\forall C \in \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9\}$ *(voir la figure 4.5).*

Les algorithmes de diagnostic combinatoire sont appliqués pour trouver les conditions sous lesquelles la valeur correcte de la sortie z restera correcte. Le résultat de cette étape est que C_9 *ne peut pas être remplacé par un autre composant de type OR ni de type NAND. L'Ensemble-P-Restreint*$(C_9,(0,0,0))$ *est donc mis à jour pour qu'il soit égal à* $\{NOR\}$ *(voir la figure 4.5).*

L'ensemble des états suivants possibles ESP^1 *est calculé comme montré dans l'exemple 4.2. Cet ensemble, ainsi que les EDR-Restreint et les Ensemble-P-Restreint correspondants, sont montrés dans la figure 4.5.*

Deuxième intervalle de temps :

Dans le deuxième intervalle de temps, les algorithmes de diagnostic combinatoire sont exécutés, sous l'application du deuxième vecteur $(1,0,0,1)$*, dans chaque état suivant possible séparément. L'EDR-Restreint et les Ensemble-P-Restreint dans chacun de ces états sont mis à jour, comme il l'est montré dans la figure 4.5.*

Puisque la séquence de test donnée contient deux vecteurs seulement, l'analyse du circuit s'arrête à ce point et une nouvelle séquence de test est générée; la même procédure recommence sous l'application de cette nouvelle séquence. Avant l'application de la nouvelle séquence, un nouvel espace de recherche Espace-de-Recherche est calculé, et pour chaque composant C dans cet espace de recherche un Ensemble-P(C) est calculé comme suit:

$$\textit{Espace-de-Recherche} = \bigcup_{E \in ESP^1} \textit{EDR-Restreint(E)}$$

$$\textit{Ensemble-P(C)} = \bigcup_{E \in ESP^1} \textit{Ensemble-P-Restreint(C,E)}$$

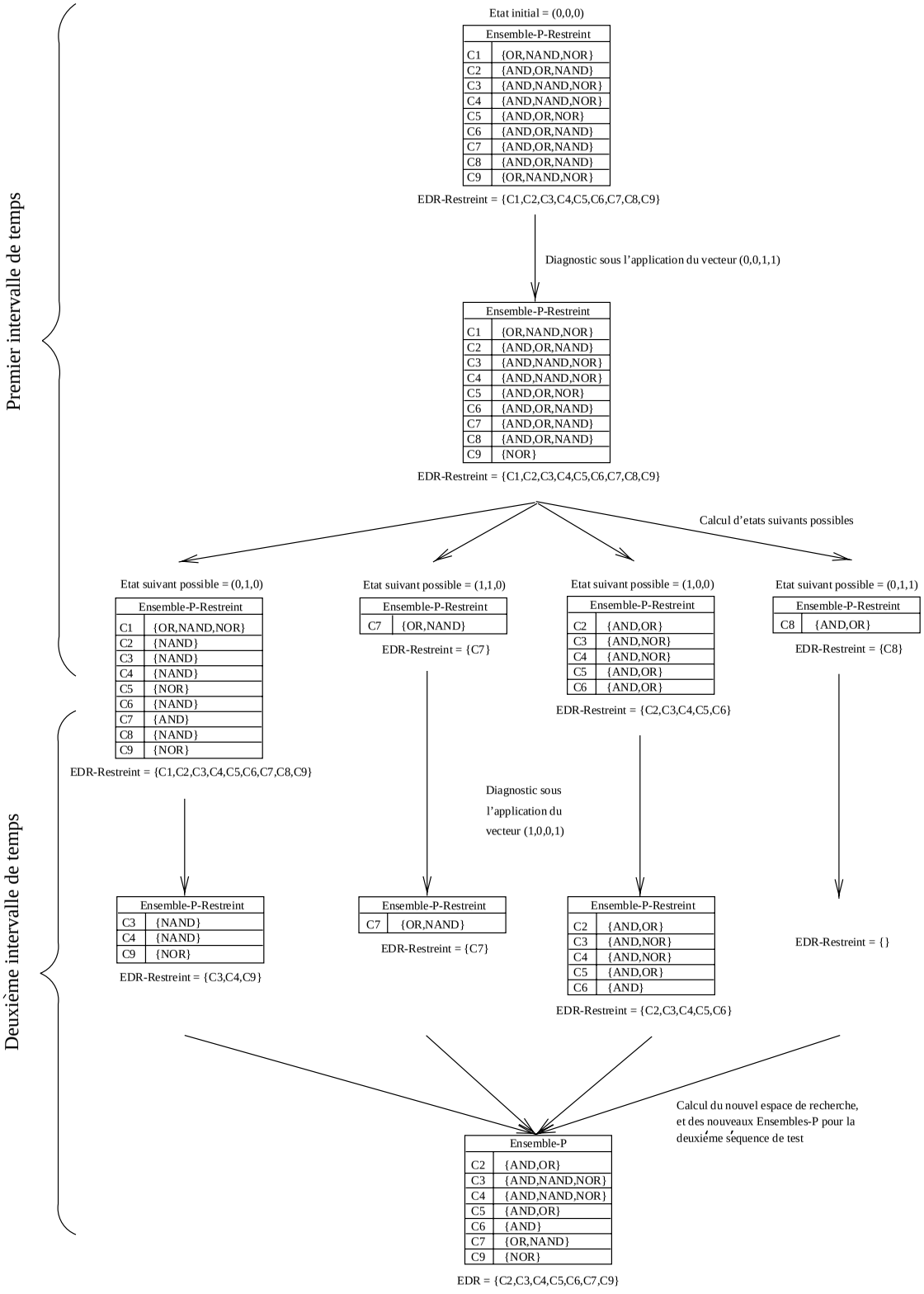

FIG. 4.5 - *Application de l'algorithme de diagnostic*

Ce nouvel Espace-de-Recherche sera l'EDR-Restreint de l'état initial quand le circuit sera analysé avec la deuxième séquence de test, et de même pour les Ensemble-P.

Il est à noter qu'avant le diagnostic, en utilisant la séquence $\{(0,0,1,1),(1,0,0,1)\}$*, il y avait 9 composants dans l'espace de recherche, et chacun de ces composants avait 3 types dans son Ensemble-P. Cela fait 27 remplacements possibles. Après le diagnostic avec cette séquence, il reste 14 remplacements possibles seulement (voir la figure 4.5). Ce nombre continue à diminuer à chaque fois qu'une séquence de plus est appliquée jusqu'à ce que l'emplacement exact de l'erreur soit trouvé.*

4.5 Résultats expérimentaux

Nous avons réalisé un système de diagnostic prototype. Ce système comprend l'algorithme de diagnostic proposé dans ce chapitre, un générateur de séquences de test, et un simulateur séquentiel. Nous avons utilisé le langage de programmation PROLOG pour son adéquation au type de problème. Un autre langage de programmation, comme le C par exemple, améliorerait la performance.

Nous avons testé ce prototype sur les jeux d'essai ISCAS89 [28]. Pour chaque circuit dans ces jeux d'essai nous avons utilisé deux versions: la version originale qui est décrite par des portes simples de type AND, OR, NOR, NAND, XOR, XNOR ou NOT, et une autre version décrite par des composants plus complexes de type MUX, AO, OA, etc. Cette dernière version est générée automatiquement par un programme qui reconnaît certaines structures de portes simples, et les remplace par des composants plus complexes. Dans nos tests la version originale a servi comme spécification et la deuxième version comme implémentation.

Dans chaque expérience, un composant de l'implémentation est choisi aléatoirement et il est remplacé aléatoirement par un autre composant de la même classe de remplacement. L'algorithme de diagnostic est ensuite appliqué. Le diagnostic commence par appliquer une séquence de test qui détecte l'erreur. Dans nos expériences, cette séquence est générée dans une phase préalable.

Les caractéristiques des circuits testés sont données dans la Table 4.1. Les

Nom du circuit	Bascules	Portes	Comp.	Boucles par bascule	Entrées primaires		Sorties primaires
					Nombre	Aff. mém.	
s27	3	10	8	1.3	4	4	1
s208	8	96	76	1.8	11	2	1
s298	14	119	92	1.8	3	3	6
s344	15	160	109	3.5	9	9	11
s349	15	150	109	3.1	9	9	11
s382	21	158	106	2.1	3	3	6
s386	6	159	94	8.0	7	6	7
s400	21	148	109	2.2	3	3	6
s420	16	196	161	1.8	19	2	1
s510	6	211	165	11.8	19	19	7
s641	19	379	218	2.9	35	15	24
s820	5	289	217	9.2	18	18	19
s838	32	390	331	1.8	35	2	1
s953	29	395	315	1.8	16	16	23
s1196	18	529	382	0.0	14	14	14
s1423	74	657	406	3.3	17	17	5
s5378	179	2779	1723	2.9	35	35	49

TAB. 4.1 - *Caractéristiques des circuits des jeux d'essai*

quatre premières colonnes donnent le nom du circuit, le nombre de bascules, le nombre de portes dans la spécification, et le nombre de composants dans l'implémentation.

La colonne intitulée *boucles par bascule* donne le nombre moyen de boucles dans lesquelles chaque bascule prend part. Cette valeur exprime la force avec laquelle les différentes boucles dans le circuit sont reliées entre elles. Il a été montré dans [104] que plus cette valeur est grande, plus la génération des vecteurs de test est difficile. La septième colonne donne le nombre d'entrées primaires qui affectent les valeurs des variables d'états. Plus ce nombre est petit, plus la longueur des séquences de test générées est grande.

La Table 4.2 donne les résultats du diagnostic obtenus. La deuxième colonne donne le nombre d'expériences effectuées sur chaque circuit: dans chaque expérience, une erreur différente est insérée. La troisième colonne donne le nombre

Circuit	Exp.	Moyenne de				Ecart type de			
		Séq.	Vec.	CPU(sec)	Cand.	Séq.	Vec.	CPU(sec)	Cand.
s27	10	2.20	3.7	0.50	1.10	0.42	1.16	0.52	0.32
s208	49	3.29	14.69	57.13	1.63	1.41	18.85	66.31	0.67
s298	125	4.58	22.58	248.50	2.54	3.53	16.99	188.09	2.62
s344	44	4.07	12.70	165.80	2.25	1.48	5.41	74.06	1.48
s349	43	3.77	11.58	157.05	1.98	1.41	4.30	62.64	1.01
s382	45	5.22	44.62	662.55	2.69	3.10	71.97	1474.52	2.09
s386	127	4.28	10.53	49.61	1.71	2.36	6.76	29.05	1.08
s400	43	4.70	30.28	467.57	2.74	2.08	30.28	851.97	1.43
s420	41	5.32	93.73	1918.94	3.27	2.30	161.13	2674.46	1.76
s510	48	3.06	11.85	199.69	1.53	1.65	7.15	90.48	0.95
s641	41	2.24	4.37	257.13	1.20	0.54	2.15	231.10	0.40
s820	59	5.10	16.10	285.58	1.76	2.53	12.71	167.69	1.06
s838	23	4.65	12.91	1093.08	3.22	2.08	19.30	1453.25	2.45
s953	68	4.34	15.29	2285.62	2.54	2.59	8.55	786.54	2.16
s1196	116	3.61	6.02	307.96	1.48	2.10	4.22	251.30	0.76
s1423	11	5.40	27.90	34193.24	2.10	1.43	11.37	10189.37	1.29
s5378	10	10.10	27.40	57174.51	1.30	3.00	8.38	23754.34	0.48

TAB. 4.2 - *Résultats des tests effectués sur les jeux d'essai ISCAS'89*

moyen de séquences de test utilisées. Chacune de ces séquences se compose d'un ou de plusieurs vecteurs. Le nombre moyen de vecteurs utilisés par expérience est donné dans la quatrième colonne. Cette valeur représente le nombre d'intervalles de temps dans lesquels le circuit est analysé avant de rapporter le résultat de diagnostic. Le temps CPU moyen, en secondes, sur une station SUN SPARC 10 à 128 Mega-octets de mémoire, est donné dans la cinquième colonne. Ce temps comprend le temps nécessaire pour simuler le circuit, générer les vecteurs de test, calculer les états suivants possibles, et appliquer les règles de diagnostic. La sixième colonne donne le nombre moyen de candidats d'erreur proposés par le système de diagnostic. L'erreur se trouve toujours parmi ces candidats. Les quatre dernières colonnes donnent les écarts types de ces valeurs. L'écart type de temps CPU est relativement grand. Ceci est justifié par les grands écarts types de nombre de vecteurs de test utilisés. Ce nombre depend de l'architecture et de la nature de circuit. Ce qui nous intéresse le plus c'est l'écart type du nombre de

candidats d'erreur. Cet écart type se montre petit pour tous les circuits: moins que 1.8 pour tous les circuits excepté 4 circuits, où il est inférieur à 2.7. Cela signifie que dans la majorité des cas étudiés, le nombre de candidats d'erreur est de l'ordre de 1 à 3.

Circuit	Max. de				Min. de			
	Séq.	Vec.	CPU	Cand.	Séq.	Vec.	CPU	Cand.
s27	3	6	1.21	2	2	2	0.21	1
s208	7	104	363	4	2	2	7	1
s298	16	80	862	11	2	4	14	1
s344	7	31	330	8	2	4	48	1
s349	6	22	277	5	2	5	48	1
s382	14	360	8116	11	2	4	69	1
s386	13	35	159	7	2	2	7	1
s400	10	159	4032	8	2	4	79	1
s420	12	819	10068	9	2	2	29	1
s510	8	35	415	6	2	2	42	1
s641	4	9	762	2	2	2	62	1
s820	13	62	845	5	2	2	42	1
s838	9	55	4344	9	2	2	109	1
s953	13	57	4361	9	2	4	902	1
s1196	15	28	1195	5	2	2	73	1
s1423	8	47	61703	5	3	14	27021	1
s5378	14	38	104612	2	6	16	27926	1

TAB. 4.3 - *Les extrêmes des résultats des tests effectués sur les jeux d'essai ISCAS'89*

Pour montrer les extrêmes, nous donnons aussi les meilleurs et les pires des cas, dans le tableau 4.3 sous les colonnes intitulées "Max de" et "Min de". Dans quelques expériences le nombre de candidats d'erreurs a atteint 11 (dans les circuits s298 et s382). Ceci est expliqué comme suit: notre algorithme de diagnostic génère des séquences de test spéciales pour les composants de l'espace de recherche. Pour des raisons de performance temporelle, le générateur de séquences de test, comme tous les générateurs existants, met une limite sur le nombre de retour-en-arrière ("backtracking") effectué pendant la génération. Si cette limite est dépassée la génération de séquence de test pour le composant choisi est aban-

donée, et un autre composant est choisi. Les composants qui restent dans l'espace de recherche après l'exécution de l'algorithme de diagnostic sont ceux dont la génération de séquences de test est abandonée, et qui ne sont pas éliminés par les autres séquences de test utilisées pendant le diagnostic.

Dans d'autres cas, le circuit peut être corrigé par plusieurs remplacements différents, et il y aura donc plusieurs candidats d'erreurs qui sont tous valides. Par exemple, dans le circuit montré dans la figure 4.6, la sortie correcte $y = a\overline{b}+\overline{a}b$ peut être obtenue par 4 remplacements différents:

1. Remplacer le composant C de type BUF par un composant de type NOT.
2. Remplacer le composant D de type BUF par un composant de type NOT.
3. Remplacer le composant Y de type BUF par un composant de type NOT.
4. Remplacer le composant I de type OR par un composant de type NOR.

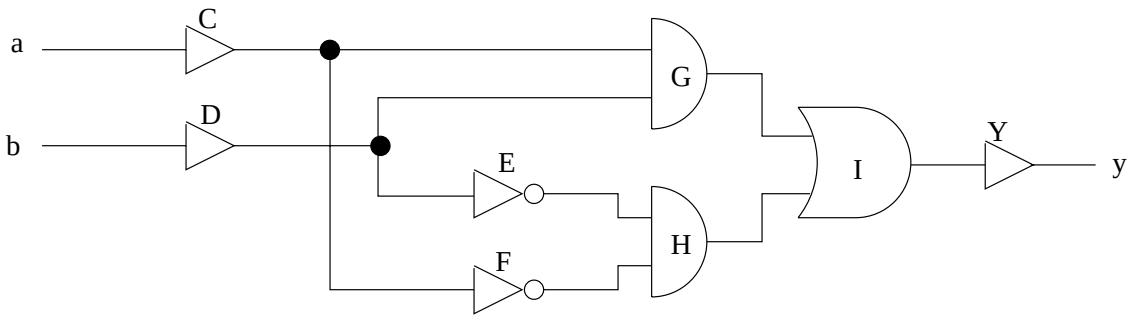

FIG. 4.6 - *La fonction correcte* $y = a\overline{b} + \overline{a}b$ *peut être obtenue par 4 remplacements possibles*

Il est à noter que le temps de diagnostic est en proportion avec le produit du *nombre de vecteurs utilisés* par le *nombre de composants dans le circuit*. Ce nombre reflète le nombre de composants traversés et analysés au cours du processus de diagnostic. Le nombre de vecteurs utilisés dépend de la longueur des séquences de test qui dépend, à son tour, du nombre de *boucles par bascule*, et du nombre d'*entrées affectant les variables mémorisantes*.

Considérons par exemple les circuits *s420* et *s641*. *s641* a un nombre de composants plus grand que celui de *s420* (218 contre 161), et un nombre de bascules plus grand aussi (19 contre 16). Pourtant, le temps de diagnostic moyen du circuit

s641 est très inférieur au temps de diagnostic du circuit *s420* (257 secondes contre 1919 secondes). Ceci est dû au fait que le circuit *s641* est analysé au moyen de 4 intervalles de temps dans chaque expérience, tandis que *s420* est analysé à travers 93 intervalles de temps.

Borne supérieure du temps de diagnostic :

Soit une implémentation utilisant une bibliothèque de composants contenant $k \times l$ types de composants qui peuvent être classés en k classes de remplacement. Chaque classe contient l types différents de composants. Pour un composant donné, l'algorithme de diagnostic a besoin de générer $l-1$ séquences de test dans le pire des cas.

Si l'implémentation est composée de c composants, alors dans le pire des cas, où chaque séquence de test générée pour un composant C élimine un seul type de l'*Ensemble-P(C)*, le nombre de séquences de test est $c \times (l-1)$. Supposons que le nombre moyen de vecteurs dans une séquence de test soit n.

Si sous l'application d'un vecteur de test l'analyse du circuit, pour faire le diagnostic et pour calculer l'ensemble des états suivants possibles, examine tous les composants du circuit (ce qui n'arrive jamais en pratique), le temps d'exécution sera proportionnel à $c^2 \times (l-1) \times n$. Donc, dans le pire des cas, le temps d'exécution augmente d'une manière quadratique avec la taille du circuit. Le facteur n dépend de la topologie du circuit. Il est à noter que cette limite représente le pire des cas, et qu'elle est une limite théorique qui n'a jamais été atteinte dans nos expérimentations.

Chapitre 5

Conclusion et perspectives

5.1 Conclusion

Dans cette thèse nous avons attaqué un problème important dans le domaine de la CAO : le diagnostic des erreurs de conception dans les circuits logiques. Le terme *diagnostic* comporte trois aspects: la *détection* de l'existence d'une erreur, la *localisation* et la *correction* de cette erreur. Ici nous nous sommes consacré à ces deux derniers aspects.

Bien que des outils automatisés de synthèse soient employés pour obtenir des produits *corrects-par-construction*, des exemples industriels montrent que, souvent, les conceptions obtenues par synthèse automatique sont ensuite modifiées manuellement pour améliorer certaines caractéristiques critiques telle que la performance ou la taille du circuit. Dans d'autres cas, les concepteurs effectuent de petites modifications sur la spécification et modifient l'implémentation en conséquence. Pendant cette phase de modification, qui est souvent faite manuellement, des erreurs peuvent être introduites involontairement.

Encore que les programmes de vérification montrent qu'il y a une erreur dans la conception et fournissent des contre-exemples mettant en évidence l'erreur, les concepteurs passent beaucoup de temps à essayer de comprendre ce qui a causé l'erreur et à identifier les points erronés dans leurs conceptions. L'expérience montre que ce travail de localisation et de correction de l'erreur peut demander au concepteur plusieurs heures de travail, auxquelles il faut ajouter le temps nécessaire pour vérifier de nouveau le circuit pour s'assurer que la correction

apportée est bonne. En général, lors de la conception de circuits logiques, le temps nécessaire pour la mise au point des circuits est comparable, voire supérieur, au temps de la conception elle-même. La correction automatique des erreurs de conception aiderait les concepteurs à trouver les erreurs plus rapidement.

Nous croyons qu'un outil de diagnostic automatique est un complément nécessaire aux outils de preuves automatiques, et il doit être intégré avec ces outils dans un cadre commun prenant en entrée des descriptions écrites dans un langage convenable de description de matériel.

Nous avons présenté ici de nouvelles méthodes permettant le diagnostic automatique des erreurs simples de conception dans les circuits logiques. Un outil prototype a été bâti en se fondant sur ces méthodes. Cet outil est supérieur aux autres outils comparables sur 4 points:

1. Il n'est pas limité aux implémentations décrites par des portes simples. Des composants plus complexes peuvent également être utilisés.

2. Il est fiable et précis: il est *fiable* dans le sens que si l'erreur suit une de nos hypothèses, alors elle est toujours trouvée. Nos hypothèses sont basées sur le modèle classique d'erreurs de conception présenté dans [3], et sur d'autres informations obtenues grâce à nos communications avec les ingénieurs de Thomson-TCS. L'outil est précis dans le sens que l'erreur est trouvée parmi un petit nombre de candidats (un seul candidat dans presque tous les cas).

3. Grâce aux vecteurs de test spécialement générés pour le diagnostic, l'erreur est localisée après l'application d'un petit nombre de ces vecteurs (10-20 vecteurs pour un circuit de 2500 composants). L'outil exploite non seulement les vecteurs de test qui détectent l'erreur, mais aussi les vecteurs qui ne la détectent pas. Ces vecteurs ne détectant pas l'erreur se montrent particulièrement efficaces, quand ils sont utilisés avec les vecteurs détectant l'erreur, pour réduire rapidement la zone suspecte du circuit. Le fait que ces vecteurs sont des vecteurs ternaires, et non pas seulement binaires, aide aussi à réduire cette zone rapidement. Les vecteurs sont générés dynamiquement pendant la phase de diagnostic, et non dans une phase préalable comme le proposent les autres méthodes; cette méthode accélère le procédé, car seuls les vecteurs nécessaires sont employés.

4. Il n'est pas limité au diagnostic des circuits combinatoires. Les circuits séquentiels peuvent aussi être traités même si l'implémentation et la spécification ont un nombre et un codage différent des variables d'états. Ceci est dû à l'emploi du nouveau concept d'*états-suivant-possibles*.

Ce prototype est actuellement intégré dans l'environnement PREVAIL et il a été utilisé avec succès pour localiser et corriger des erreurs dans plusieurs versions d'un circuit fourni par Thomson-TCS, et dans un ensemble de jeux d'essai classiquement utilisés par les chercheurs dans ce domaine (ISCAS'85, ISCAS'89). Le prototype prend en entrées une spécification et une implémentation décrites en VHDL (c'est le langage d'entrée de PREVAIL), et génère en sortie un rapport contenant les candidats d'erreur, leurs emplacements dans le fichier VHDL et les corrections proposées. Le style de description de la spécification n'est pas restreint tandis que l'implémentation doit être décrite de manière structurelle par un réseau de composants inter-connectés.

Dans ce prototype nous employons le concept d'*hypothèses d'erreurs*: le problème du diagnostic, posé dans sa forme générale, est extrêmement complexe: il a été prouvé NP-complet [73]. Nous avons suivi une approche pragmatique, en identifiant des types d'erreurs et en proposant des méthodes traitant chaque type séparément. Parmi ces types d'erreurs, certains sont rencontrés plus fréquemment que les autres [1]. Dans notre prototype, ils sont examinés successivement par ordre de probabilité décroissante, et dans chaque cas la méthode de diagnostic appropriée est appliquée. Si le diagnostic échoue sous l'hypothèse d'un certain type d'erreur, un autre type est présumé et le diagnostic est refait sous cette nouvelle hypothèse. Si l'erreur n'est trouvée sous aucune de ces hypothèses, nous aurons au moins l'information que le type d'erreur n'est pas un des types examinés.

Les types d'erreurs que nous avons considérés dans cette thèse sont les erreurs de remplacement de composants, dans les circuits combinatoires et séquentiels et les erreurs de connexions dans les circuits combinatoires. Nous avons considéré ces types en se basant sur le modèle classique d'erreurs de conception présenté dans [3], et sur d'autres informations obtenues grâce à nos communications avec les ingénieurs de Thomson-TCS dans le cadre du projet JESSI-AC3.

Une hypothèse souvent faite dans la littérature, et que nous avons aussi adop-

tée, est qu'il n'y a qu'une seule erreur dans l'implémentation. Cette hypothèse peut sembler irréaliste car d'autres erreurs plus catastrophiques peuvent aisément être imaginées. Cependant, ce modèle est pratiquement valable si la vérification est faite fréquemment pendant la phase de changement. Pour des raison d'efficacité humaine, nous croyons que chaque série de modification dans une implémentation doit être suivie par une phase de vérification formelle. Une fois que l'équivalence entre l'implémentation initiale et l'implémentation modifiée (pour une optimisation manuelle), ou entre la spécification modifiée et l'implémentation modifiée (pour un changement fonctionnel dans la spécification) est niée, le système de diagnostic est invoqué. Si peu de changements ont été effectués, la probabilité d'insérer des erreurs multiples n'est pas très élevée, et la correction peut être obtenue automatiquement.

5.2 Perspectives

Nous envisageons l'extension de ce travail selon deux axes principaux:

1. **L'extension à d'autres types d'erreurs:**
   Une extension directe est le traitement des erreurs de connexions dans les circuits séquentiels. Une autre extension majeure consiste à traiter des erreurs plus complexes. Par exemple, les erreurs de *fils échangés*: dans ce type d'erreurs, il y a plus d'une source du signal erroné, contrairement à notre modèle d'erreur actuel où il n'y a qu'une source pour le signal erroné. Les erreurs de fils échangés représentent une forme particulière d'un autre problème plus géneral: le problème du diagnostic des erreurs multiples. La localisation et la correction des erreurs multiples reste toujours un problème pour lequel aucune méthode satisfaisante n'existe actuellement.

2. **L'amélioration de la performance de notre outil:**
   Les trois modules de notre prototype (le simulateur, le générateur de vecteurs de test et le module de diagnostic) sont écrits en PROLOG. Nous avons utilisé PROLOG parce qu'il permet un prototypage rapide; de nombreuses expériences antérieures ont montré son adéquation pour mettre en œuvre des algorithmes de simulation [78], de génération de vecteurs de

test [65, 79] et de diagnostic [17, 66, 119, 153, 113]. Nous démarrons avec des stagiaires une réécriture de notre prototype en langage C^{++}. Cela requiert beaucoup plus de travail de programmation, mais nous espérons gagner au moins deux ordres de grandeur en temps de calcul.

Une autre façon d'améliorer la performance est de remplacer certains modules de notre prototype par d'autres modules plus efficaces: dans le cas des circuits séquentiels, par exemple, un vérificateur de machines d'états-finis basé sur l'emploi des graphes de décisions binaires (BDD), qui effectue un parcours symbolique de l'espace d'états afin de vérifier l'équivalence entre une spécification et son implémentation, peut générer les séquences de test les plus courtes qui détectent l'erreur [38]. Nous ne possédons pas un tel outil; le générateur prototype que nous avons développé ne génére pas toujours les séquences de test les plus courtes. Le fait que la longueur d'une séquence de test détectant l'erreur est minimale peut être utilisé pour optimiser davantage l'algorithme de diagnostic: les états-suivant-possibles à partir desquels l'erreur est détectée, avant que le dernier vecteur de la séquence de test soit appliqué, peuvent être écartés.

Bibliographie

Références concernant la vérification:
[11, 15, 16, 19, 20, 21, 24, 32, 30, 35, 36, 40, 56, 58, 57, 59, 61, 62, 63, 64, 67, 68, 69, 80, 84, 88, 97, 99, 101, 107, 109, 131, 139, 143, 155]

Références concernant la génération de vecteurs de test:
[4, 5, 6, 10, 12, 13, 18, 25, 27, 28, 37, 38, 41, 42, 43, 45, 44, 48, 46, 47, 49, 51, 53, 52, 60, 65, 70, 74, 75, 76, 77, 79, 81, 83, 82, 89, 93, 98, 102, 104, 105, 106, 110, 111, 112, 114, 118, 120, 121, 122, 126, 125, 127, 130, 132, 136, 138, 148]

Références concernant les graphes de décision binaires:
[8, 23, 29, 31, 33, 34, 39, 84, 85, 90, 115, 116, 117, 134, 135]

Références concernant le diagnostic:
[1, 2, 3, 7, 9, 14, 17, 26, 50, 54, 55, 66, 71, 72, 78, 86, 87, 91, 92, 94, 96, 100, 103, 108, 113, 119, 123, 124, 128, 129, 133, 137, 140, 141, 142, 144, 145, 146, 147, 150, 151, 149, 152, 153, 154, 155, 156]

Références diverses:
[22, 73, 78, 87, 95]

[1] E. J. Aas, K. A. Klingsheim, and T. Steen, "Quantifying Design Quality: A Model and Design Experiments," *Proc. EURO-ASIC'92*, pp. 172-177, 1992.

[2] E. J. Aas, T. Steen and K. Klingsheim, "Quantifying Design Quality Through Design Experiments," *IEEE Design & Test of Computers*, pp. 27-38, Spring 1994.

[3] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic Design Verification via Test Generation," *IEEE Transactions on Computer-Aided Design*, Vol. 7, No. 1, pp. 138-148, Jan. 1988.

[4] M. Abramovici and M. A. Breuer, "Multiple Fault Diagnosis in Combinational Circuits Based on an Effect-Cause Analysis," *IEEE Transactions on Computers*, Vol. C-29, No. 6, pp. 451-460, June 1980.

[5] M. Abramovici, J. J. Kulikowski, P. R. Menon, and D. T. Miller, "Test Generation in LAMP2: Concepts and algorithms," *International Test Conference (ITC'85)*, pp. 49-56, 1985.

[6] V. D. Agrawal, K.-T. Cheng, and P. Agrawal, "CONTEST: A concurrent test generator for sequential circuits," *Proceedings of 25th Design Automation Conference*, pp. 84-89, June 1988.

[7] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek, "Test Program Generation for Functional Verification of PowerPC Processors in IBM," *Proceedings of the 32nd Design Automation Conference DAC'95*, pp. 279-285, 1995.

[8] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, Vol. C-26, No. 6, pp. 509-516, June 1978.

[9] T. W. Albrecht, "Concurrent Design Methodology and Configuration Management of the SIEMENS EWSD-CCS7E Processor System Simulation," *Proceedings of the 32nd Design Automation Conference DAC'95*, pp. 222-227, 1995.

[10] P. Altenbernd and J. Strathaus, "Solving the Path Sensitization Problem in Linear Time," *Proceedings of EDAC'92*, pp. 378-382, 1992.

[11] H. Asaad and J. P. Hayes, "Design Verification via Simulation and Automatic Test Pattern Generation," *Proceedings of ICCAD'95*, 1995.

[12] E. Auth, and M. H. Schulz, "A Test Pattern Generation Algorithm for Sequential Circuits," *IEEE Design and Test of Computers*, Vol. 8, pp. 72-85, June 1991.

[13] B. Ayari, and B. Kaminska, "A New Dynamic Test Vector Compaction for Automatic Test Pattern Generation", *IEEE Trans. On CAD*, Vol. 13, No. 3, pp. 353-358, March 1994.

[14] F. Barsi, F. Grandoni and P. Maestrini, "A Theory of Diagnosability of Digital Systems," *IEEE Transactions on Computers*, Vol. C-25, No. 6, pp. 585-593, June 1976.

[15] A. Bartsch, H. Eveking, H. Faerber, M. Kelelatchew, J. Pinder and U. Schellin, "LOVERT - A Logic Verifier of Register Transfer Descriptions," *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Houthalen (Belgium), Nov. 1989.

[16] A. Bartsch, H. Eveking, H. J. Faerber, M. Kelelatchew, J. Pinder, and U. Schellin, "LOVERT - A Logic Verifier of Register-Transfer Descriptions," *Proc. Formal VLSI Correctness Verification*, L. Claesen, ed., pp. 247-256, North-Holland, 1990

[17] J. S. Bennet and C. R. Hollander, "DART: An Expert System for Computer Fault Diagnosis," *Proc. International Joint Conference on Artificial Intelligence*, 1981.

[18] M. J. Bending, "Hitest: A Knowledge-Based Test Generation System," *IEEE Design and Test of Computers*, Vol. 1, pp-83-92, May 1984.

[19] D. Borrione, L. Pierre, and A. Salem, "Formal Verification of VHDL Descriptions in the Prevail environment," *IEEE Design & Test of Computers*, Vol. 9, No.2, pp. 42-56, June 1992.

[20] D. Borrione, H. Bouamama, R. Suescun, "Validation of the Numeric_Bit Package using NQTHM Theorem Prover," *Proc. Asia Pacific Conference on Hardware Description Languages (APCHDL'96)*, pp. 168-172, Jan. 1996.

[21] D. Borrione, H. Bouamama, D. Déharbe, C. Le Faou, and A. Wahba "HDL-Based Integration of Formal Methods and CAD Tools in the PREVAIL Environment," *Proc. Int. Conf. on Formal Methods in Computer Aided Design, Lecture Notes in Computer Science*, No. 1166, Springer Verlag, pp. 450-467, Nov. 1996.

[22] R. S. Boyer, and J. S. Moore, *A Computational Logic Handbook*, Academic Press, New York, 1988.

[23] K.S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," *Proceedings of 27th ACM/IEEE Design Automation Conference DAC'90*, pp. 40-45, 1990.

[24] D. Brand, "Exhaustive Simulation Need Not Require an Exponential Number of Tests", *IEEE Trans. On CAD*, Vol. 12, No. 11, pp. 1635-1641, Nov. 1993.

[25] M. A. Breuer, "A Random and an Algorithmic Technique for Fault Detection Test Generation for Sequential Circuits," *IEEE Transactions on Computers*, Vol. C-20, No. 11, pp. 1364-1370, Nov. 1971.

[26] M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, New York: Computer Science Press, 1976.

[27] F. Brglez, and H. Fujiwara, "A neutral netlist of 10 combinatorial benchmark circuits and a target translator in FORTRAN," in *Proc. IEEE Int. Symp. Circuits and Systems*, pp. 663-698, June 1985.

[28] F. Brglez, D. Bryan and K. Koźmiński, "Combinational Profiles of Sequential Circuits," *Proceedings of International Symposium of Circuits and Systems (ISCAS'89)*, Portland, OR, May 1989.

[29] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, Vol. C-35, No.8, pp. 677-691, August 1986.

[30] R. E. Bryant, "A Methodology for Hardware Verification Based on Logic Simulation," *Journal of ACM*, Vol. 38, No. 2, pp. 299-329, April 1991.

[31] R. E. Bryant, "On the Complexity of VLSI Implementations and Graph Representation of Boolean Functoins with Application to Integer Multiplication," *IEEE Trans. on Computers*, Vol. 40, No. 2, pp. 205-213, Feb. 1991.

[32] R. E. Bryant, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," *Proceedings of 28th Design Automation Conference (DAC'91)*, pp. 397-402, 1991.

[33] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Survey*, Vol. 24, No. 3, pp. 293-318, Sep. 1992.

[34] R. E. Bryant, "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification," *Proceedings of ICCAD'95*, 1995.

[35] J. R. Burch, E. M. Clarke, K. L. McMillan and D. L. Dill,"Sequential Circuit Verification Using Symbolic Model Checking," *Proceedings of 27th Design Automation Conference (DAC'90)*, pp. 46-51, 1990.

[36] J. R. Burch, "Using BDDs to Verify Multipliers," *Proceedings of 28th Design Automation Conference (DAC'91')*, pp. 408-412, 1991.

[37] G. Cabodi, P. Camurati, F. Corno, S. Gai, P. Prinetto and M. S. Reorda, "A New Model for Improving Symbolic Product Machine Traversal," *Proceedings of 29th Design Automation Conference (DAC'92)*, pp. 614-619, 1992.

[38] G. Cabodi, P.Camurati, and S. Quer, "Symbolic Exploration of Large Circuits with Enhanced Forward/Backward Traversals," *Proceedings of European Design Automation Conference (EURO-DAC'94)*, pp. 22-27, Sep. 1994.

[39] N. Calazans, Q. Zhang, R. Jacobi, B. Yernaux and A. M. Trullemans, "Advanced Ordering and Manipulation Techniques for Binary Decision Diagrams," *Proceedings of EDAC'92*, pp. 452-457, 1992.

[40] P. Camurati and P. Prinetto, "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research," *IEEE Computer*, pp. 8-19, July 1988.

[41] S. J. Chandra and J. H. Patel, "A Hierarchical Approach to Test Vector Generation," *Proceedings of 24rd Design Automation Conference (DAC'87)*, pp. 495-501, 1987.

[42] S. T. Chakradhar, V. D. Agrawal, S. G. Rothweiler, "A Transitive Closure Algorithm for Test Generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 7, pp. 1015-1028, July 1993.

[43] K. T. Cheng, and V. D. Agrawal, "A Simulation-Based Directed-Search Method for Test Generation," *Proceeding of International Conference on Computers Design (ICCD'87)*, Port Chester, NY, pp. 48-51, Oct. 1987.

[44] W.-T. Cheng, "The Back Algorithm for Sequential Test Generation," *Proceedings of International Conference of Computers Design (ICCD'88)*, pp. 66-69, Oct. 1988.

[45] W. -T. Cheng, "Split Circuit Model for Test Generation," *Proceedings of 25th Design Automation Conference (DAC'88')*, pp. 96-101, June 1988.

[46] K. T. Cheng and V. D. Agrawal, "Concurrent Test Generation and Design for Testability," *Proc. ISCAS'89*, pp. 1935-1938, 1989.

[47] W. -T. Cheng, and J. Chakraborty, "GENTEST: An Automatic Test-Generation System for sequential circuits," *IEEE Computer*, Vol. 22, No. 4, pp. 43-48, April 1989.

[48] W. T. Cheng and S. Davidson, "Sequential Circuit Test Generator (STG) Benchmark Results," *Proc. ISCAS'89*, pp. 1939-1941, 1989.

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

[50] W. T. Cheng, J. L. Lewandowski and E. Wu, "Optimal Diagnostic Method for wiring Interconnects", IEEE Trans. On CAD, Vol. 11, No. 9, Sept. 1992.

[51] H. Cho, G. D. Hachtel, S.-W. Jeong, B. Pleisser, E. Schwarz, and F. Somenzi, "ATPG aspects of FSM verification," *Proceedings of IEEE International Conference of Computer-Aided Design*, pp. 134-137, Nov. 1990.

[52] H. Cho, G. D. Hachtel, and F. Somenzi, "Redundancy Identification/Removal and Test Generation for Sequential Circuits Using Implicit State Enumeration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 7, pp. 935-945, July 1993.

[53] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi, "Algorithms for Approximate FSM Traversal," Proceedings of 30th Design Automation Conference, pp. 25-30, TX, USA, June 1993.

[54] P. Y. Chung, I. N. Hajj, "ACCORD: Automatic Catching and CORrection of Logic Design Errors in Combinational Circuits," *Proceedings of International Test Conference ITC'92*, pp. 742-751, 1992.

[55] P. Y. Chung, Y. M. Wang, I. N. Hajj, "Diagnosis and Correction of Logic Design Errors in Digital Circuits," *Proceedings of 30th Design Automation Conference DAC'93*, pp. 503-508, 1993.

[56] E. M. Clarke, E. A. Emerson and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions On Programming Languages and Systems*, Vol. 8, No. 2, pp. 244-263, April 1986.

[57] O. Coudert, C. Berthet and J. C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, No. 407, Springer Verlag, pp. 365-373, Grenoble, France, June 1989.

[58] O. Coudert, C. Berthet, and J. C. Madre, "Verification of Sequential Machines Using Boolean Functional Vectors," *Proceedings of IFIP International Workshop on Applied*

Formal Methods for Correct VLSI Design, L. Claesen, ed., Leuven, Belgium, pp. 111-128, Nov. 1989.

[59] O. Coudert and J. C. Madre, "Symbolic Computation of the Valid States of a Sequential Machine: Algorithms and Discussion," *Proc. ACM/IFIP International Workshop on Formal Methods for Correct VLSI Design*, Participants' Edition, Miami, Jan. 1991.

[60] H. Cox and J. Rajski, "A Method of Fault Analysis for Test Generation and Fault Diagnosis," *IEEE Transactions on Computer-Aided Design*, Vol. 7, No. 7, pp. 813-833, July 1988.

[61] D. Déharbe, D. Borrione, "Symbolic Model Checking with Past and Future Temporal Modalities: Fundamentals and Algorithms," Current Issues in Electronic Modeling, Vol. 1, on *Model Generation in Electronic Design*, Kluwer, March 1995.

[62] D. Déharbe, "Vérification Formelle de Propriétés Temporelles: Étude et Application au Langage VHDL," *Thèse de Doctorat*, Université Joseph Fourier, Grenoble, France, 1996.

[63] P. Déverchére, C. Madre, J. B. Guignet and M. Currat, "Functional Abstraction and Formal Proof of Digital Circuits," *Proceedings of EDAC'92*, pp. 458-462, 1992.

[64] D. L. Dill and J. Rushby, "Acceptance of Formal Methods: Lessons from Hardware Design," *IEEE Computer*, Vol. 29, No, 4, pp. 23-24, April 1996.

[65] R. O. Duda, "Application of Expert Systems," *Automatic test generation workshop*, California, USA, March 1983.

[66] K. Eshghi, "Application of Metalevel Programming to Fault Finding in Logic Circuits," *Proc. 1st International Logic Programming Conference*, Marseille, France, Sep. 1982.

[67] H. Eveking, "Axiomatizing Hardware Description Languages," *International Journal of VLSI Design*, Vol. 2, No. 3, pp. 263-280, 1990.

[68] C. L. Faou, L. Pierre, and A. Salem, "A User-Oriented Presentation of Prevail: A Proof Environment for VHDL Descriptions," *Technical Report*, No. 71, IMAG/ARTEMIS, Grenoble, Sep. 1991.

[69] T. Filkorn, "A Method for Symbolic Verification of Synchronous Circuits," *Proc. 10^{th} International Symposium on Computer Hardware Description Languages and their Applications*, D. Borrione and R. Waxman ed., pp. 249-259, Elsevier Science Publishers, April 1991.

[70] H. Fujiwara, and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers*, Vol. C-32, No. 12, pp.1137-1144, Dec. 1983.

[71] M. Fujita, T. Kakuda and Y. Matsunga, "Redesign and Automatic Error Correction of Combinational Circuits," *Proc. of IFIP TC10/WG10.5 Workshop on Logic and Architecture Synthesis*, pp. 253-262, North-Holland, Elsevier Science Publishers, 1990.

[72] M. Fujita, "Methods for Automatic Design Error Correction in Sequential Circuits," *Proceedings of European Conference on Design Automation with The European Event in ASIC Design*, 1993, pp. 76-80, 1993.

[73] M. R. Garey, and D. S. Johnson, "Computers and Interactibility: A Guide to the Theory of NP-Completeness," Freeman, New York, 1979.

[74] A. Ghosh, S. Devadas, and R. Newton, "Test Generation and Verification for Highly Sequential Circuits," *IEEE Transactions on Computer-Aided Design*, Vol. 10, No. 5, May 1991.

[75] U. Gläser and H. T. Vierhaus, "FOGBUSTER: An Efficient Algorithm for Sequential Test Generation," *Proceedings of EURO-DAC'95*, pp. 230-241, 1995.

[76] Prabhakar Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, Vol. C-30, No. 3, pp. 215-222, March 1981.

[77] L. H. Goldstein, "Controllability/Observability Analysis of Digital Circuits," *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, No. 9, pp. 685-693, Sep. 1979.

[78] E. Gullichsen, "Heuristic Circuit Simulation using PROLOG," *INTEGRATION, the VLSI Journal*, North-Holland, 1985.

[79] R. Gupta, "Test Pattern Generation for VLSI Circuits in a PROLOG Environment," *3rd. International Conference on Logic Programming*, E. Shapiro Ed., Springer Verlag, England, July 1986.

[80] A. Gupta, "Formal Hardware Verification Methods: A Survey," *Formal Methods in System Design*, pp. 151-238, Kluwer Academic Publishers, 1992.

[81] P. Hansen, B. Jaumard and M. Minoux, "A Linear Expected-Time Algorithm for Deriving All Logical Conclusions Implied by a Set of Boolean Inequalities," *Mathematical Programming*, Vol. 34, pp. 223-231, March 1986.

[82] M. Henftling, H. Wittmann and K. J. Antreich, "A Single-Path-Oriented Fault-Effect Propagation in Digital Circuits Considering Multiple-Path Sensitization," *Proceedings of ICCAD'95*, 1995.

[83] M. Henftling, H. Wittmann and K. J. Antreich, "A Formal Non-Heuristic ATPG Approach," *Proceedings of EURO-DAC'95*, pp. 248-253, 1995.

[84] S. Höreth, "Improving the Performance of a BDD-based tautology checker," *Proc. Advanced Research Workshop on Correct Hardware Design Methodology*, Turin, North Holland, 1991.

[85] S. Höreth, "Compilation of Optimized OBDD-Algorithms," *Proceedings of EURO-DAC'96*, pp. 152-157, 1996.

[86] D. Humblot, "Naissance d'un Ordinateur: Bull DPS 7000, à livre ouvert," *Oeuvre collective coordonnée par Dan Humblot, Réseaux et Système d'information BULL S.A.*, BULL, 1992.

[87] R. L. Hummel, "Cyrix 6x86 Bug Puts Brakes on NT 4.0," *Byte Magazine*, pp.30-32, Nov. 1996.

[88] W. A. Hunt, "FM8501: A Verified Microprocessor," *Technical Report 47*, University of Texas at Austin, 1986.

[89] K. S. Hwang and M. R. Mercer, "Derivation and Refinement of Fan-Out Constraints to Generate Tests in Combinational Logic Circuits," *IEEE Transactions on CAD*, Vol. CAD-5, No. 4, pp. 564-572, Oct. 1986.

[90] S. W. Jeong, B. Plessier, G. D. Hachtel and F. Somenzi, "Variable Ordering for Binary Decision Diagrams," *Proceedings of EDAC'92*, pp. 447-451, 1992.

[91] S. Kang and S. A. Szygenda, "New Design Error Modeling and Metrics for Design Validation," *Proceedings of EURO-DAC'92*, pp. 472-477, 1992.

[92] S. Kang and S. A. Szygenda, "Modeling and Simulation of Design Errors," *Proceedings of International Conference of Computers Design (ICCD'92)*, 1992.

[93] T. Kirkland and M. R. Mercer, "A Topological Search Algorithm for ATPG," *Proceedings of 24rd Design Automation Conference (DAC'87)*, pp. 502-508, 1987.

[94] J. Kleer and B. C. Williams, "Diagnosing Multiple Faults," *Artificial Intelligence*, No. 32, pp. 97-130, Elsevier Science Publishers, 1987.

[95] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill Book Edition, 1978.

[96] A. Kuehlmann, D. I. Cheng, A. Srinivasan, and D. P. LaPotin, "Error Diagnosis for Transistor-Level Verification," *Proceedings of 31st Design Automation Conference (DAC'94)*, pp. 218-224, 1994.

[97] A. Khehlmann, A. Srinivasan and D. P. LaPotin, "Verity - A Formal Verification Program for Custom CMOS Circuits," *IBM J. RES. DEVELOP.*, Vol. 39, No. 1/2, pp. 149-165, Jan/March 1995.

[98] W. Kunz and D. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," *Proc. International Test Conference ITC'92*, pp. 816-825, 1992.

[99] W. Kunz and D. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems - Test, Verification and Optimization," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 9, pp. 1143-1158, Sept. 1994.

[100] S. Y. Kuo, "Locating Logic Design Errors via Test Generation and Don't Care Propagation," *Proceedings of EURO-DAC'92*, pp. 466-471, 1992.

[101] Y. T. Lai and S. Sastry, "Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification," *Proceedings of 29th Design Automation Conference (DAC'92)*, pp. 608-613, 1992.

[102] J. Lee and J. H. Patel, "Architecture Level Test Generation for Microprocessors", *IEEE Trans. On CAD*, Vol. 13, No. 10, pp. 1288-1300, Oct. 1994.

[103] H.-T. Liaw, J.-H. Tsaih, C.-S. Lin, "Efficient Automatic Diagnosis of Digital Circuits," *Proceedings of ICCAD'90*, pp. 464-467, 1990.

[104] A. Lioy, P. L. Montessoro, and S. Gai, "A Complexity Analysis of Sequential ATPG," *Proceedings of IEEE International Symposium of Circuits and Systems (ISCAS'89)*, pp. 1946-1949, May 1989.

[105] H.-K. T. MA, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli, "Test Generation for Sequential Circuits," *IEEE Transactions on Computer-Aided Design*, Vol. 7, pp. 1081-1093, Oct. 1988.

[106] E. Macii and A. R. Meo, "A Test Generation Program for Sequential Circuits," *Journal of Electronic Testing: Theory and Applications*, No. 5, pp. 115-119, Kluwer Academic Publishers, 1994.

[107] J. C. Madre and J. P. Billon, "Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behaviour," *Proceedings of 25th Design Automation Conference (DAC'88)*, pp. 205-210, 1988.

[108] J. C. Madre, O. Coudert, and J. P. Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM", *Proc. ICCAD'89*, pp. 30-33, 1989.

[109] J. C. Madre, "PRIAM: Un Outil de Vérification Formelle de Circuits Intégrés Digitaux," Thèse de Doctorat, École Nationale Supérieure des Télécommunications, Paris, Juin 1990.

[110] S. Mallela, and S. Wu, "A Sequential Test Generation System," *Proceedings of International Test Conference*, Philadelphia, PA, pp. 57-61, Oct. 1985.

[111] R. A. Marlett, "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits," *Proceedings of 15th Design Automation Conference (DAC'78)*, pp. 335-339, 1978.

[112] R. Marlett, "An Effective Test Generation System for Sequential Circuits," *Proceedings of 23rd Design Automation Conference (DAC'86)*, pp. 250-256, 1986.

[113] M. Marzouki and B. Courtois, "Debugging Integrated Circuits: A.I. can Help !," *Proceedings of 1st European Test Conference ETC'89*, pp. 184-191, 1989.

[114] E. J. McCluskey, "Verification testing - a pseudo exhaustive test technique," *IEEE Transactions on Computers*, Vol. C-33, No. 6, June 1984.

[115] M. R. Mercer, R. Kapur and D. E. Ross, "Functional Approaches to Generating Orderings for Efficient Symbolic Representations," *Proceedings of 29th Design Automation Conference (DAC'92)*, pp. 624-627, 1992.

[116] S. I. Minato, N. Ishiura and S. Yajima, "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation," *Proceedings of 27th Design Automation Conference (DAC'90)*, pp. 52-57, 1990.

[117] S. I. Minato, "Fast Generation of Irredundant Sum-of-Products Forms from Binary Decision Diagrams," *Proceedings of SASIMI'92*, pp. 64-73, 1992.

[118] H. B. Min, H. A. Luh, and W. A. Rogers, "Hierarchical Test Pattern Generation: A Cost Model and Implementation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 7, pp. 1029-1039, July 1993.

[119] R. Mullis, "An Expert System for VLSI Tester Diagnosis," *Proc. International Test Conference ITC'84*, Philadelphia, USA, 1984.

[120] S. Muroga, Y. Kambayashi, H. C. LAI and J. N. Culliney, "The Transduction Method - Design of Logic Networks Based on Permissible Functions," *IEEE Transactions on Computers*, Vol. 38, No. 10, pp. 1404-1424, Oct. 1989.

[121] P. Muth, "A Nine-Value Circuit Model for Test Generation," *IEEE Transactions on Computers*, Vol. C-25, pp. 630-636, June 1976.

[122] S. Nitta, M. Kawamura, and K. Hirabayash, "Test Generation by Activation and Defect-Drive (TEGAD)," *INTEGRATION*, Vol. 3, pp. 2-12, 1985.

[123] G. Odawara, M. Tomita, O. Okuzawa, T. Ohta, and Z. Q. Zhuang, "A Logic Verifier Based on Boolean Comparison," *Proceedings of 23rd Design Automation Conference (DAC'86)*, pp. 208-214, 1986.

[124] V. Pitchumani, P. Mayor and N. Radia, "A System for Fault Diagnosis and Simulation of VHDL descriptions," *Proceedings of 28th Design Automation Conference (DAC'91)*, pp. 144-150, 1991.

[125] I. Pomeranz, S. M. Reddy, "3-Weight Pseudo-Random Test Generation Based on a Deterministic Test Set for Combinational and Sequential Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 7, pp. 1050-1058, July 1993.

[126] I. Pomeranz, L. N. Reddy, S. M. Reddy, "COMPACTEST: A Method to Generate Compact Test Sets for Combinational Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 7, pp. 1040-1049, July 1993.

[127] I. Pomeranz and S. M. Reddy, "On Generating Compact Test Sequences for Synchronous Sequential Circuits," *Proceedings of EURO-DAC'95*, pp. 105-110, 1995.

[128] C. V. Ramamoorthy and W. Mayeda, "Computer Diagnosis Using the Blocking Gate Approach," *IEEE Transactions on Computers*, Vol. C-20, No. 11, pp. 1294-1299, Nov. 1971.

[129] R. Reiter, "A Theory of Diagnosis from First Principles," *Artificial Intelligence*, No. 32, pp. 57-95, Elsevier Science Publishers, 1987.

[130] J. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Transactions on Electronic Computers*, Vol. EC-16, No. 5, pp. 567-580, October 1967.

[131] A. Salem, "Vérification Formelle des Circuits Digitaux Décrits en VHDL," *Thèse de Doctorat*, Université Joseph Fourier, Grenoble, France, 1992.

[132] M. H. Schulz, E. Trischler and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Trans. On CAD*, Vol. 7, No. 1, pp. 126-137, Jan. 1988.

[133] F. F. Sellers, M. Y. Hsiao and L. W. Bearnson, "Analyzing Errors with the Boolean Difference," *IEEE Transactions on Computers*, Vol. C-17, No. 7, pp. 676-683, July 1968.

[134] E. M. Sentovich, "A Brief Study of BDD Package Performance," *Proc. Int. Conf. on Formal Methods in Computer Aided Design, Lecture Notes in Computer Science*, No. 1166, Springer Verlag, pp. 389-403, Nov. 1996.

[135] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli and R. K. Brayton, "Heuristic Minimization of BDDs Using Don't Cares," *Proceedings of 31st Design Automation Conference (DAC'94)*, 1994.

[136] S. Shteingart, A. W. Nagle, and J. Garson, "RTG: Automatic Register Level Test Generator," *Proceedings of 22nd Design Automation Conference*, pp. 803-807, June 1985.

[137] W. R. Simpson and J. W. Sheppard, *System Test and Diagnosis*, Kluwer Academic Publisher, 1994.

[138] N. Singh, *An Artificial Intelligence Approach to Test Generation*, Kluwer Academic Publishers, Boston, MA, 1987.

[139] R. Suescun, "Preuve d'une Bibliothèque VHDL pour la Synthèse Logique: Spécification des Mécanismes de Traduction Automatique dans la Logique de Boyer et Moore," *Mémoire de DEA*, Université Joseph Fourier, Grenoble, France, June 1995.

[140] K. A. Tamura, "Locating Functional Errors in Logic Circuits," *Proceedings of 26th Design Automation Conference (DAC'89)*, pp. 185-191, 1989.

[141] M. Tomita, and H. H. Jiang, "An Algorithm for Locating Logic Design Errors," Proceedings of ICCAD'90, pp. 468-471, 1990.

[142] M. Tomita, T. Yamamoto, F. Sumikawa, and K. Hirano, "Rectification of Multiple Logic Design Errors in Multiple Output Circuits," *Proceedings of the 31st Design Automation Conference DAC'94*, pp. 212-217, 1994.

[143] "VHDL Tool Integration Platform (VTIP)," CAD Language Systems, INc., Rockville, Md., April 1990.

[144] A. M. Wahba, "Vérification et Diagnostic des Circuits Logiques Combinatoire par Utilisation de la Simulation à 3 Valeurs," *Mémoire de DEA*, ENSIMAG, Université Joseph Fourier, Grenoble, France, June 1993.

[145] A. Wahba, and D. Déharbe, "Design Error Diagnosis in Logic Circuits using Ternary Test Sets," *Research Report RR-928-M*, ARTEMIS-IMAG, Grenoble, France, Dec. 1993.

[146] A. M. Wahba, E. J. Aas, "Verification and Diagnosis of Digital Systems by Ternary Reasoning," *Proc. Correct Hardware Design and Verification Methods, CHARME'93, Lecture Notes in Computer Science* No. 683, Springer Verlag, pp. 55-67, May 1993.

[147] A. Wahba, and D. Borrione, "Design Error Diagnosis in Sequential Circuits," *Proc. Correct Hardware Design and Verification Methods, CHARME'95, Lecture Notes in Computer Science* No. 987, pp. 171-188, Springer Verlag, Oct. 1995.

[148] A. Wahba, "General Survey on Sequential Test Generation Methods," *Research Report RR-948-I*, ARTEMIS-IMAG, Grenoble, France, July 1995.

[149] A. Wahba, and D. Borrione, "Connection Errors Location and Correction in Combinational Circuits," *Research Report TIMA 96.1.I*, INPG, Grenoble, France, Feb. 1996.

[150] A. Wahba, and D. Borrione, "A Method for Automatic Design Error Location and Correction in Combinational Logic Circuits," *Journal of Electronic Testing: Theory and Applications*, Kluwer, Vol. 8, No. 2, April 1996.

[151] A. Wahba, and D. Borrione, "Automatic Diagnosis may Replace Simulation for Correcting Simple Design Errors," *Proceedings of EURO-DAC'96*, pp. 476-481, 1996.

[152] A. Wahba, and D. Borrione, "Connection Error Location and Correction in Combinational Circuits," *Proceedings of ED&TC'97*, pp. 235-241, 1997.

[153] A. J. Wilkinson, "A Method for Test System Diagnosis Based on the Principles of Artificial Intelligence," *Proc. International Test Conference ITC'84*, Philadelphia, USA, 1984.

[154] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein, "System Design Methodology of UltraSPARC^{TM}-I," *Proceedings of the 32nd Design Automation Conference DAC'95*, pp. 7-12, 1995.

[155] Q. H. Zhang, C. Trullemans, "Logic Verification of Incomplete Functions and Design Error Location," *Proc. Correct Hardware Design and Verification Methods, CHARME'93, Lecture Notes in Computer Science* No. 683, Springer Verlag, pp. 68-79, May 1993.

[156] Q. Zhang, "Logic Verification and Design Error Diagnosis for Combinational Circuits," *Ph.D. Thesis*, Université Catholique de Louvain, Belgium, Feb. 1995.

Annexe A

L'environnement PREVAIL

PREVAIL est un environnement de preuve automatique commandé par un système de menus qui se présente à l'utilisateur par une interface graphique interactive unifiée (voir Figure A.1). C'est un environnement multi-HDL, multi-outil, qui vérifie les circuits combinatoires à délai nul, et les circuits séquentiels synchronisés par une horloge. PREVAIL est le résultat d'un effort de coopération entre l'Université de Provence-Marseille, l'Université Joseph Fourier-Grenoble, et Technische Hochschule Darmstadt.

Aujourd'hui, deux langages HDL sont acceptés comme langage d'entrée: SMAX et VHDL. SMAX a été conçu à Darmstadt en s'appuyant sur CONLAN [67]; il est spécialement conçu pour le raisonnement, au niveau des vecteurs de bit, sur les circuits combinatoires et les machines d'états finis. SMAX a une sémantique de logique du premier ordre, et du point de vue sémantique il est équivalent à un sous-ensemble de VHDL.

La standardisation du langage de description VHDL et sa large utilisation dans le domaine de la CAO, surtout en Europe, ont conduit le groupe à étudier son usage pour la vérification. VHDL est malheureusement un langage complexe et il n'existe pas de définition formelle standard de sa sémantique. Notre équipe a procédé de façon pragmatique en choisissant un sous-ensemble de VHDL pour lequel une sémantique a été formellement définie. Le sous-ensemble reconnu par PREVAIL s'appelle P-VHDL [131]. C'est un sous-ensemble synthétisable étendu avec des paramètres génériques.

PREVAIL intègre plusieurs outils de preuves et de diagnostic. Chacun de ces

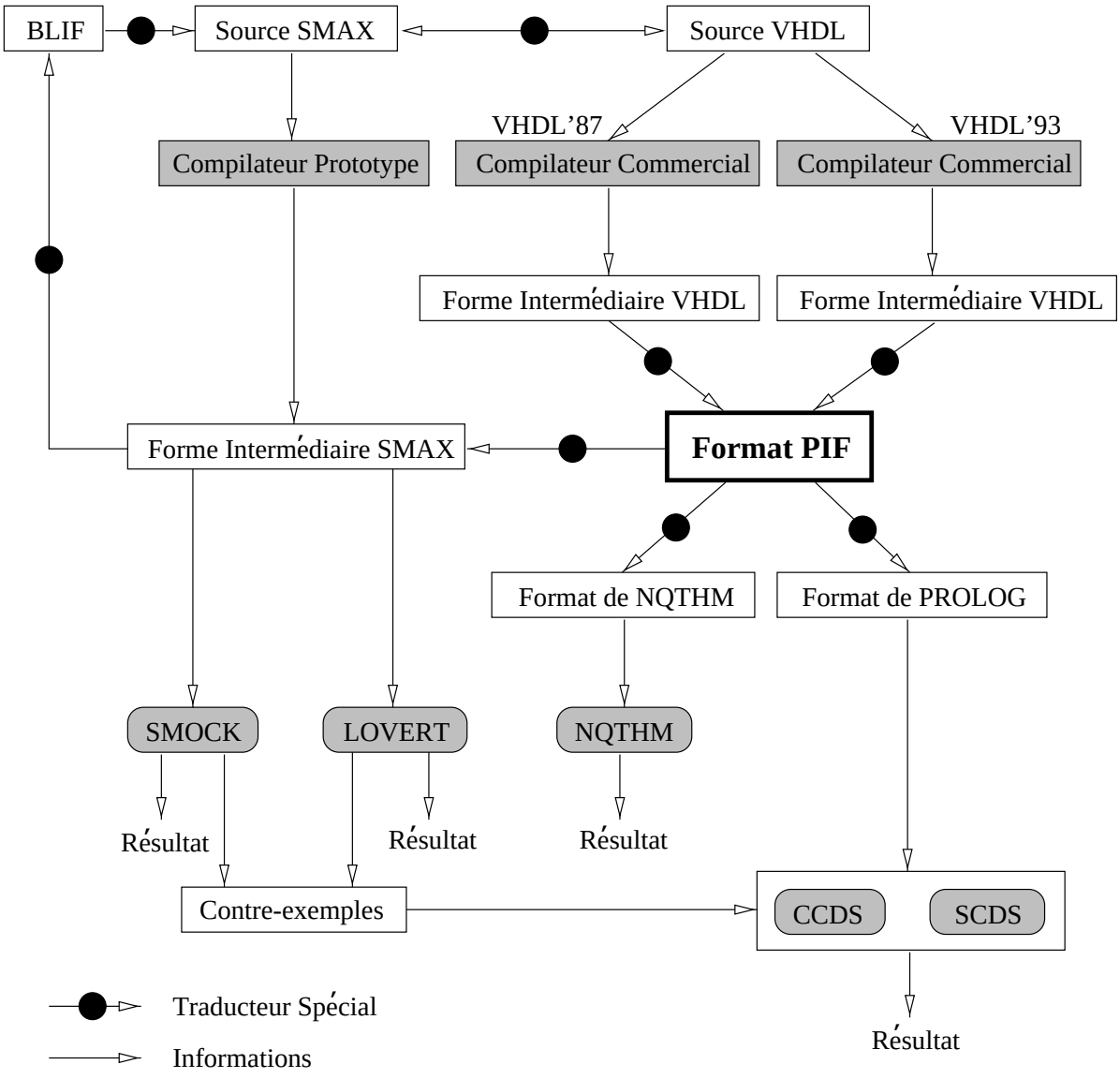

FIG. A.1 - *Environnement PREVAIL*

outils a son propre format d'entrée. Pour surmonter ce problème, un format intermédiaire orienté preuve, nommé *PIF*, a été défini dans PREVAIL. *PIF* est le format de départ à partir duquel des traducteurs spécialisés génèrent la forme requise par chaque outil dans PREVAIL. Le source VHDL décrivant un circuit est compilé par un compilateur commercial, et une forme intermédiaire est générée, après une phase de vérification syntaxique et sémantique [143]. Le format *PIF* est généré à partir de celle-ci par un traducteur spécialisé. Le format intermédiaire SMAX est un sous-ensemble de PIF (par exemple, PIF contient des structures répétitives qui doivent être macro-générées dans le format intermédiaire de SMAX).

Actuellement, PREVAIL est basé sur le compilateur SMAX de Darmstadt, et le logiciel VTIP, fourni par COMPASS, pour VHDL87. Nous sommes en train d'incorporer LVS (fourni par LEDA) pour l'analyse syntaxique de VHDL'93. Dans le cas de VHDL, VTIP et LVS produisent un arbre syntaxique à partir duquel notre logiciel VHDL2PIF produit la structure PIF. PREVAIL peut être étendu pour accepter d'autres langages HDL, sous réserve de développer soit des traducteurs *source-à-source*, soit des compilateurs *source-à-PIF*. Par exemple, le traducteur *BLIF à SMAX* a été réalisé.

A.1 Les outils de vérification formelle

La vérification formelle des circuits consiste à prouver que pour toutes les valeurs acceptables de l'état initial, et pour toutes les valeurs possibles des entrées primaires, l'implémentation produit les mêmes valeurs à ses sorties conformément à sa spécification.

La correction d'un circuit n'est pas un concept absolu; elle comprend plusieurs attributs tel que le comportement fonctionnel, les propriétés (vivacité, sûreté, ...etc), ou les caractéristiques temporelles.

La vérification formelle nécessite l'existence d'un modèle mathématique pour représenter les caractéristiques en question, et d'un système de calcul symbolique sur ce modèle. La plupart des modèles décrivent le circuit par une fonction exprimant les sorties en fonction d'entrées et d'états internes, sans exprimer les comportements temporels et électriques complexes. Le système de calcul doit prouver ou réfuter l'existence d'une relation (équivalence, implication, ...etc) entre les

modèles de l'implémentation et de la spécification. Une telle vérification montre seulement la correction logique du circuit, et elle doit être complétée par d'autres méthodes pour vérifier le circuit après sa fabrication.

Les outils de preuves intégrés dans PREVAIL permettent d'effectuer les tâches suivantes:

- **La preuve d'équivalence entre deux machines d'états finis:**
  L'une des machines est l'implémentation et l'autre est la spécification, qui peut avoir un nombre et un codage de variables d'état différents de ceux de l'implémentation. Nous prouvons qu'à partir d'un certain état initial, les sorties des deux machines sont équivalentes. L'outil LOVERT est un vérificateur de tautologie, et un vérificateur d'équivalence pour les machines d'états finis. Il a été réalisé par le groupe de H.Eveking [16] en se basant sur les principes de Coudert et al. [57].

- **La preuve de propriété (vivacité, sûreté, ... etc) :**
  Les propriétés sont représentées par des formules, F, de la logique temporelle. La preuve consiste à calculer, soit par des méthodes symboliques soit par l'énumération, l'ensemble des états atteignables à partir de l'état initial où F est valable, en utilisant un algorithme de point fixe. SMOCK (Symbolic Model Checker), réalisé par David Déharbe [62], est inspiré du travail du groupe de Ed Clarke [35]. SMOCK vérifie des formules écrites en CTL-P, une extension de CTL [56] pour pouvoir exprimer le passé [61].

LOVERT et SMOCK sont bâtis à partir d'un outil de OBDD (Ordered Binary Decision Diagrams) développé à Technische Hochscule Darmstadt par Stefan Höreth [84, 85]. Comme tous les autres outils qui sont basés sur l'utilisation de BDD, les paramètres génériques peuvent être traités seulement quand ils sont fixés à une valeur constante. Si cette valeur change, la vérification doit être refaite.

Quand la spécification est donnée par une fonction arithmétique, ou quand l'objet à vérifier est une fonction paramétrique, ou un module de bibliothèque qui doit être prouvé correct pour toutes les valeurs admissibles de ses paramètres, dans un ensemble illimité, des méthodes de raisonnement différentes, comme la preuve par induction, sont nécessaires. Pour cette raison, le démonstrateur de

théorèmes NQTHM de Boyer et Moore [22] a été intégré dans PREVAIL. Ce démonstrateur a été choisi pour ses puissantes stratégies de preuve, et pour ses capacités à traiter des gros problèmes automatiquement [88].

Après modélisation des primitives VHDL dans la logique de NQTHM [139], Hakim Bouamama a réalisé un traducteur spécialisé qui transforme la structure PIF dans un ensemble de définitions de fonctions exprimées dans le langage d'entrée de NQTHM, et qui génère les lemmes correspondants [20].

Le contrôle des étapes de vérification et le choix de l'outil à utiliser sont effectués par l'utilisateur.

A.2 Les outils de diagnostic

Quand une implémentation est trouvée erronée, les concepteurs doivent faire face à une tâche difficile, celle de la localisation et de la correction de l'erreur. Nous avons intégré dans PREVAIL les outils de diagnostic dont les principes constituent le sujet de cette thèse. Ces outils, comme nous l'avons montré auparavant, localisent l'erreur et suggèrent une correction avec un taux de réussite de 100 % quand il s'agit d'une seule erreur de composant ou de connexion. CCDS(Combinational Circuits Diagnostic System) est l'outil de diagnostic des circuits combinatoires, et SCDS (Sequential Circuits Diagnostic System) est celui des circuits séquentiels.

A.3 Les commandes et le contrôle

Dans PREVAIL, le but est de fournir un environnement ouvert et convivial qui puisse intégrer des outils formels, une fois qu'ils sont disponibles. Un système multi-tâches a été réalisé par Hakim Bouamama pour permettre l'exécution de plusieurs outils simultanément. Un système de contrôle des tâches permet à l'utilisateur de visualiser, de démarrer, d'interrompre, et d'afficher les résultats d'une ou de plusieurs tâches. Tous les outils sont invoqués à partir d'une interface graphique unifiée, réalisée en Tcl/Tk par Claude Le Faou.

Le système de contrôle de tâches est composé de cinq modules spécialisés: le démarreur, le visualiseur, l'aide en ligne, l'exécuteur, et le contrôleur de verrous

(voir figure A.2). Tous les modules sont basés sur Tcl/Tk, mais seuls les trois premiers sont visibles graphiquement.

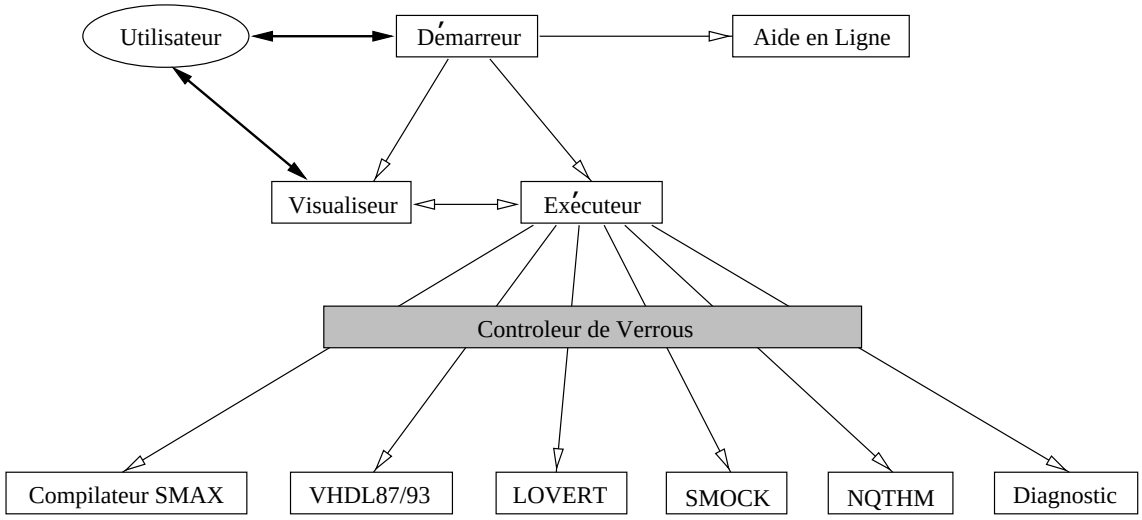

FIG. A.2 - *Architecture de l'interface-utilisateur de PREVAIL*

- **Le démarreur :**
  Le démarreur est le point d'entrée dans PREVAIL. Il coordonne l'environnement dans son ensemble. Les initialisations globales, les interactions avec l'utilisateur, et les passages aux autres modules sont faits à partir de ce module. Un nouveau module peut être intégré dans PREVAIL en établissant son lien avec le démarreur. Ceci a été démontré par l'insertion du système VFORMAL de COMPASS en faisant un tel lien.

- **Le visualiseur :**
  Il visualise les messages et l'état des tâches actuellement lancées. L'utilisateur peut vérifier quel outil est actif, voir les messages affichés par une tâche sélectionnée, voire arrêter une tâche à n'importe quel moment. Il est très utile de pouvoir arrêter les tâches lorsqu'elles sont bloquées, lentes ou ne sont plus nécessaires.

- **L'aide en ligne :**
  Cette aide en ligne contient des informations sur les aspects suivants:

  - L'utilisation de PREVAIL à partir de l'interface graphique.

- L'utilisation de PREVAIL en mode ligne de commande (sans passer par l'interface graphique).
- L'accès aux manuels sur différents outils.
- Les variables d'environnement qui contrôlent le système.

– **L'exécuteur :**
L'exécuteur est directement lié aux outils de PREVAIL: les compilateurs, les outils de preuves, les outils de diagnostic, et les éditeurs de texte (qui ne sont pas montrés dans la figure A.2). Quand un outil est choisi par l'utilisateur, l'exécuteur lance l'outil en arrière-plan. Ainsi, plusieurs outils peuvent être lancés simultanément.

– **Le contrôleur de verrous:**
Il protège les données susceptibles d'être utilisées par plusieurs outils simultanément, et empêche la destruction ou la modification des données utilisées par une tâche au cours de son exécution. Par exemple, si le compilateur VHDL2PIF est en train de générer le format PIF d'une architecture utilisant certains composants, le contrôleur de verrous assure que la description de ces composants ne changera pas tant que le compilateur VHDL2PIF n'aura pas terminé la compilation.

A.4 L'intégration de l'outil de diagnostic

La figure A.3 montre comment l'outil de diagnostic est intégré dans PREVAIL. Le traducteur spécialisé génère à partir de la description PIF d'une architecture donnée un fichier correspondant, *entité.architecture.pl*, contenant des faits PROLOG décrivant l'architecture. Pour chaque composant pouvant être utilisé dans l'architecture (un composant de bibliothèque, ou un composant défini localement) le traducteur génère un fichier décrivant ce composant, toujours en utilisant des faits PROLOG, sous la forme d'un réseau de portes simples: AND, OR, NOR, NAND, NOT et BUF. Les fichiers décrivant les composants ont l'extension *".l.pl"* au lieu de *".pl"*. Le *".l"* signifie que ce fichier représente un composant de bibliothèque (*"library" en anglais*). Un fichier *entité.architecture.pl* contient les faits PROLOG suivants:

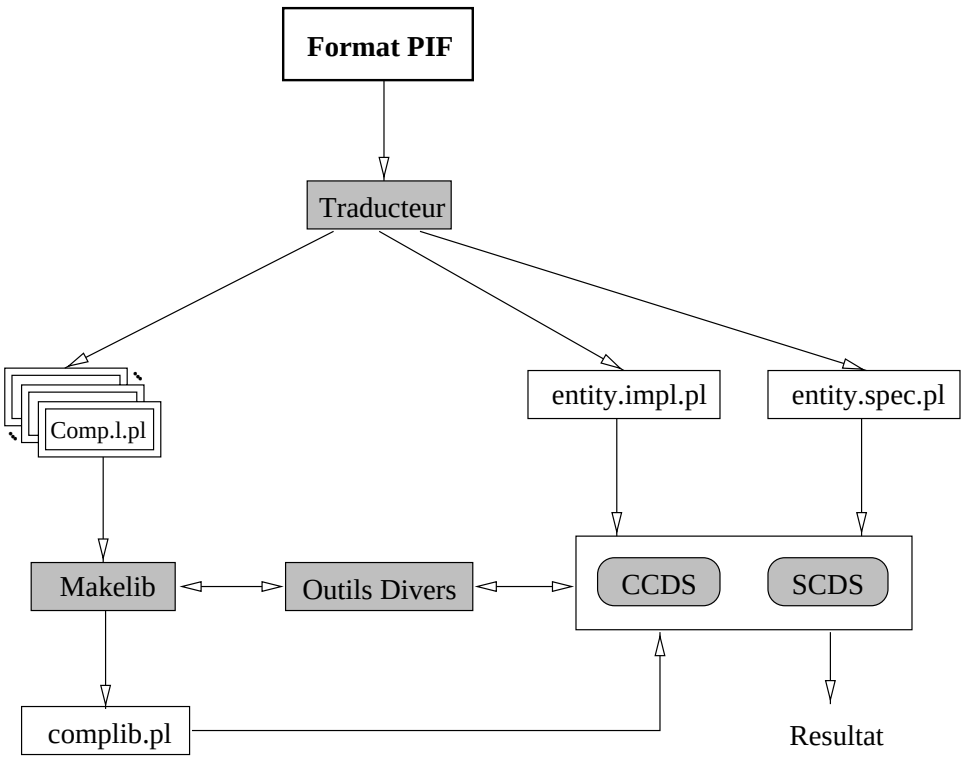

FIG. A.3 - *Intégration de l'outil de diagnostic*

- **input**(<liste-des-entrées>).
  Ce fait donne la liste des entrées primaires de l'entité.

- **output**(<liste-des-sorties>).
  Ce fait donne la liste des sorties primaires de l'entité.

- **gate**(<type>,<liste-des-entrées>,<sortie>, <numéro-de-ligne>).
  Chaque sortie primaire et chaque signal interne est décrit par un fait ayant la forme décrite ci-dessus.

  - <type> est le type du composant dont la sortie est le signal en question.
  - <liste-des-entrées> est la liste des entrées de ce composant.
  - <sortie> est le nom du signal.
  - <numéro-de-ligne> est le numéro de ligne dans le fichier VHDL où l'instanciation de ce composant apparaît.

L'application *makelib* génère le fichier *complib.pl* à partir des fichiers décrivant les composants de la bibliothèque. *complib.pl* contient les règles nécessaires pour calculer les informations suivantes:

- les classes de remplacement;
- les vecteurs de test discriminants et non-discriminants;
- les règles de simulation de composants;
- l'ON-SET et l'OFF-SET les plus faibles de chaque composant;
- les vecteurs d'entrée permettant la propagation de la valeur D à la sortie de chaque composant.

Exemple A.1 :
Nous présentons ici un exemple d'une machine séquentielle à deux variables d'état. Nous donnons le code VHDL et le code PROLOG correspondant généré par le traducteur.

VHDL

```
entity ma_machine is
    port(x: in bit; ck: in clock; z: out bit);
end ma_machine

architecture struct of w_machine is
    component and2
        port(in1,in2: in bit; out1:out bit);
    end component;

    component ao21
        port(in1,in2,in3: in bit; out1:out bit);
    end component;

    component buf
        port(in1: in bit; out1:out bit);
    end component;

    component or2
        port(in1,in2: in bit; out1:out bit);
    end component;

    component mux2
        port(select,in1,in2: in bit; out1:out bit);
    end component;

    component not1
        port(in1: in bit; out1:out bit);
    end component;

– configuration des composants
for all:and2 use entity work.and_2(dataflow);
for all:ao21 use entity work.ao_21(dataflow);
for all:buf use entity work.buf(dataflow);
for all:or2 use entity work.or_2(dataflow);
for all:mux2 use entity work.mux_2(dataflow);
for all:not1 use entity work.not_1(dataflow);

– les signaux et les sorties primaires
signal
    u1,u2,u3,u4,u5,u6,u7,u8,u9,w1,w2,w3: bit;
    y1,y2: bit := '0';
    ny1,ny2,nx: bit;

begin
    reg : block (ck='1' and not ck'stable)
    begin
        DFF1: y1 <= guarded w1;
        DFF2: y2 <= guarded w2;
    end block;

    C1 : not1 port map(y1,ny1);
    C2 : not1 port map(y2,ny2);
    C3 : not1 port map(x,nx);

    C4: ao21 port map(y1,ny2,x,u1);
    C5: or2 port map(ny1,y2,u2);
    C6: or2 port map(u1,u2,u3);
    C7: buf port map(u3,w1);

    C8: and2 port map(y2,x,u4);
    C9: and2 port map(ny2,y1,u5);
    C10: mux3 port map(u2,u4,u5,u6);
    C11: buf port map(u6,w2);

    C12: and2 port map(y2,nx,u7);
    C13: and2 port map(u3,y1,u8);
    C14: or2 port map(u7,u8,u9);
    C15: buf port map(u9,z);
end struct;
```

PROLOG

```
input(['x']).
output(['z']).


% Les variables mémorisantes

gate('DFF',['w1'],['y1'],48).
gate('DFF',['w2'],['y2'],49).


% Les signaux et les sorties primaires

gate('not_1.dataflow',['y1'],['ny1'],52).
gate('not_1.dataflow',['y2'],['ny2'],53).
gate('not_1.dataflow',['x'],['nx'],54).

gate('ao_21.dataflow',['y1','ny2','x'],['u1'],56).
gate('or_2.dataflow',['ny1','y2'],['u2'],57).
gate('or_2.dataflow',['u1','u2'],['u3'],58).
gate('buf.dataflow',['u3'],['w1'],57).

gate('and_2.dataflow',['y2','x'],['u4'],61).
gate('and_2.dataflow',['ny2','y1'],['u5'],62).
gate('mux_2.dataflow',['u2','u4','u5'],['u6'],63).
gate('buf.dataflow',['u6'],['w2'],64).

gate('and_2.dataflow',['y2','nx'],['u7'],66).
gate('and_2.dataflow',['u3','y1'],['u8'],67).
gate('or_2.dataflow',['u7','u8'],['u9'],68).
gate('buf.dataflow',['u9'],['z'],69).
```

Remarque:

Les signaux affectés dans le bloc gardé *reg* dans le source VHDL sont reconnus comme étant des éléments mémorisants, et cela est représenté en PROLOG par les portes spéciales de types *DFF*. Les entrées de ces portes sont les lignes d'état-présent, et leurs sorties sont les lignes d'état-suivant. Dans P-VHDL, les éléments mémorisants sont identifiés par des affectations gardées, où l'expression de garde implique le front d'horloge. Par exemple, *((ck='1') and not ck'stable*) est utilisé pour exprimer le front montant de l'horloge. Les expressions de temps de VHDL (les clauses *after* et *until*, et l'attribut *'delay*) ne sont pas reconnues en P-VHDL, et le temps est représenté par des cycles d'horloge. Le type *clock* est défini dans un paquetage de preuves fourni avec PREVAIL. Dans le fichier PROLOG, l'entrée *ck* de type *clock* ne figure pas dans la liste des entrées du circuit, car sa seule fonction est de synchroniser l'affectation des variables d'état.

Annexe B

Manuel d'utilisation

Nous présentons dans cette annexe un manuel d'utilisation au travers d'un exemple montrant comment utiliser l'outil de diagnostic dans le système PREVAIL en passant d'abord par la phase de vérification.

Quand vous démarrez PREVAIL, une fenêtre nommée *"Prevail"* apparaît sur l'écran (figure B.1).

FIG. B.1 - *Le point de départ de PREVAIL*

Cette fenêtre est le point de départ de PREVAIL à partir de laquelle vous

pouvez continuer avec PREVAIL en appuyant sur GO, quitter PREVAIL en cliquant sur QuitPREVAIL ou voir quelques informations concernant PREVAIL en cliquant sur Apropos. La figure B.1 montre aussi une autre fenêtre nommée "*Prevail Watcher*". C'est un visualiseur qui vous permet de visualiser les messages envoyés par les différents outils pendant leur exécution. Si vous choisissez de continuer avec PREVAIL, en cliquant sur GO, une nouvelle fenêtre nommée *"Prevail Starter"* apparaîtra. De cette fenêtre vous pouvez spécifier le langage de description de vos circuits: "VHDL" ou "SMAX", en cliquant sur le bouton correspondant (figure B.2).

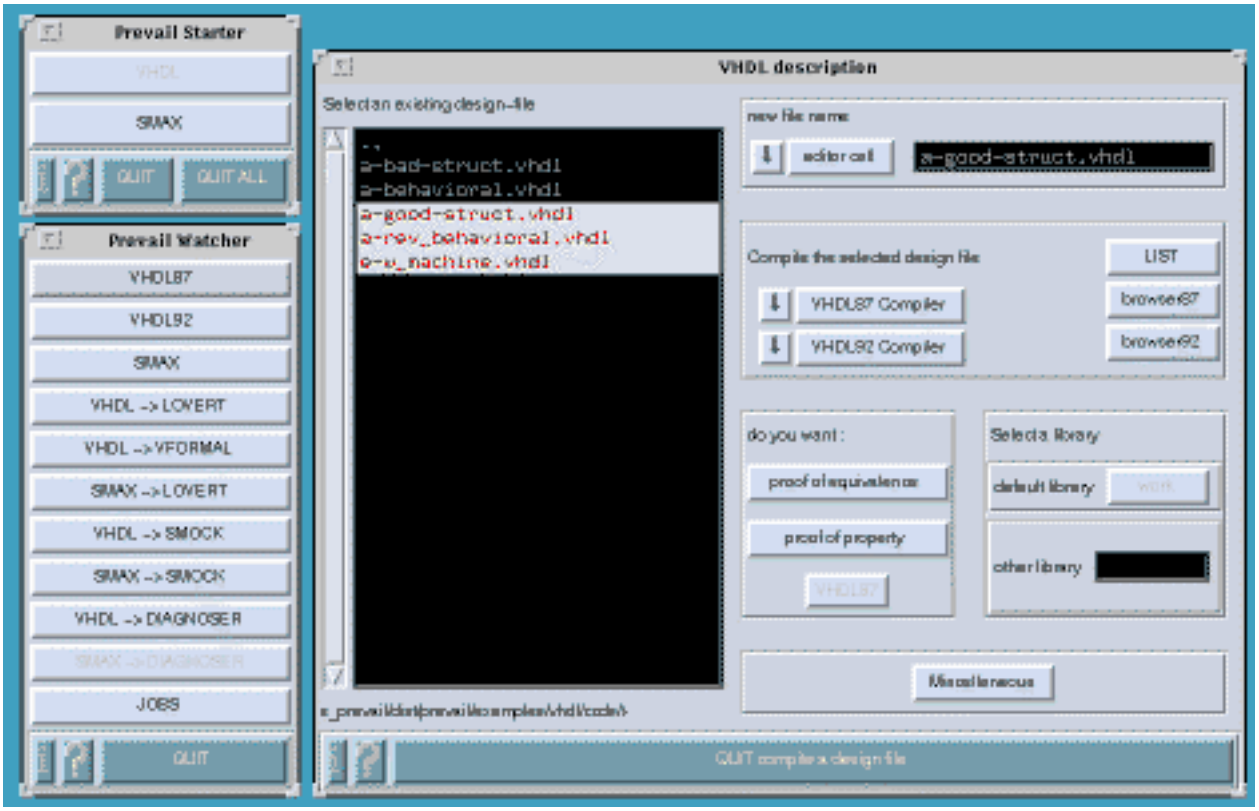

FIG. B.2 - *Choisir et compiler les descriptions VHDL à vérifier*

La première chose à faire après avoir choisi le langage de description est de compiler les fichiers de description. Vous choisissez les fichiers à compiler en cliquant sur leurs noms, puis vous cliquez sur le bouton de compilateur souhaité. Dans le cas de VHDL vous avez le choix entre VHDL87 ou VHDL92. Pendant la compilation vous pouvez regarder les messages générés par le compilateur

en cliquant sur le bouton correspondant dans la fenêtre *"Prevail Watcher"*. Une autre possibilité pour voir les messages est d'attendre jusqu'à la fin de compilation où cette fenêtre apparaîtra automatiquement. L'étape suivante après la compilation est d'effectuer la vérification souhaitée. Vous avez le choix entre la preuve d'équivalence, quand il s'agit de prouver l'équivalence entre deux descriptions, ou la preuve de propriété quand il s'agit de vérifier certaines propriétés d'une description donnée. Vous choisissez le type de vérification voulue en cliquant sur le bouton correspondant. Il faut aussi que vous précisiez la bibliothèque de travail, soit en cliquant sur work (la bibliothèque par défaut), soit en tappant le nom d'une autre bibliothèque.

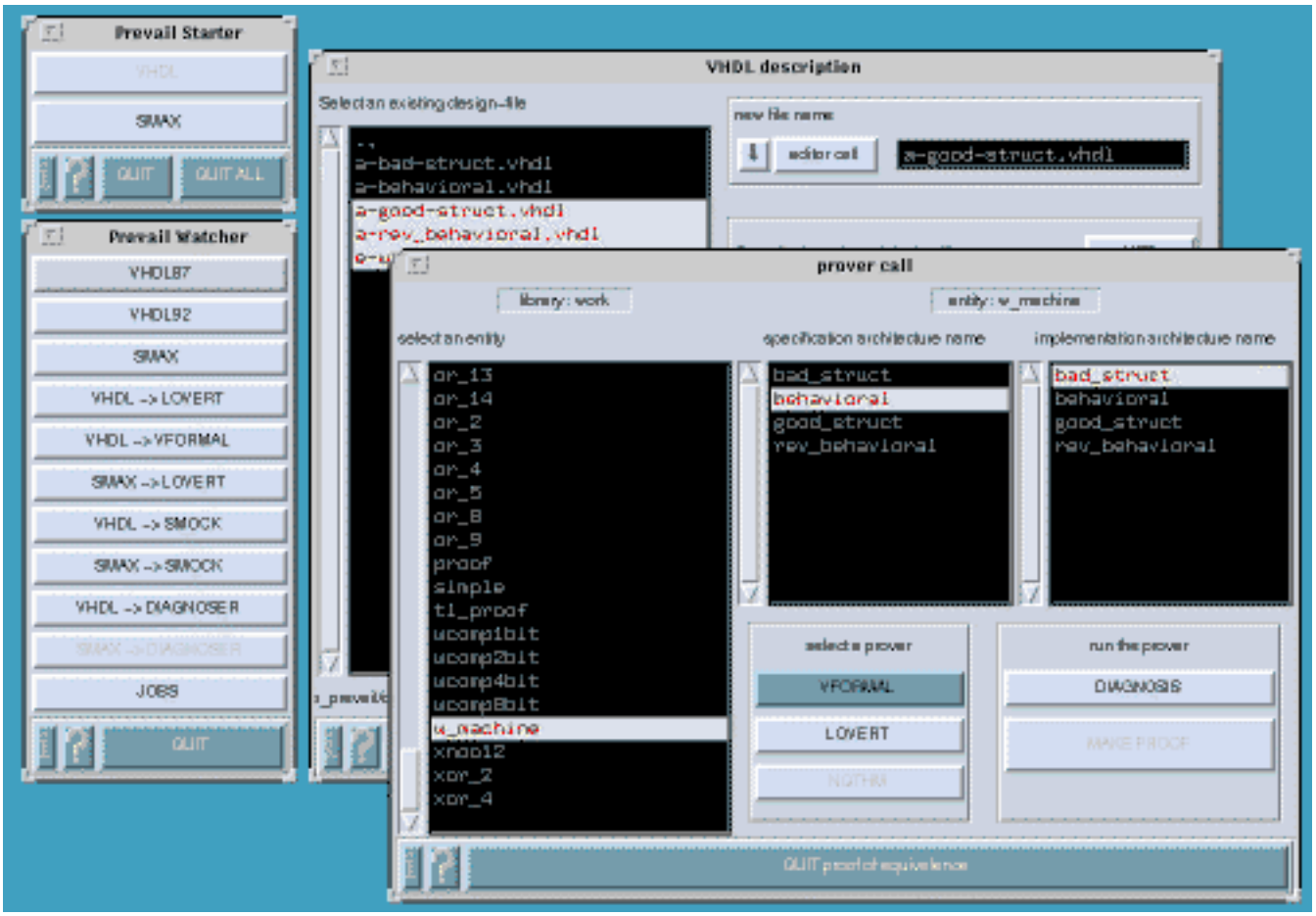

FIG. B.3 - *Sélectionner deux architectures à prouver et appeler le prouveur*

La figure B.3 montre le cas où la preuve d'équivalence est choisie. Dans ce cas vous devez choisir une description en tant que spécification et une autre

description en tant qu'implémentation. Pour commencer la preuve vous devez cliquer sur le bouton correspondant au vérificateur d'équivalence voulu (LOVERT ou Vformal). Quand le vérificateur termine sa tâche il vous affiche le résultat dans une fenêtre spéciale. Si l'équivalence est prouvée, vous aurez un visage souriant et un message annonçant que les deux descriptions sont équivalentes. Sinon, vous aurez un grand point d'exclamation pour vous avertir que l'équivalence est niée (voir la figure B.4). Dans ce cas, le vérificateur génère aussi un contre-exemple mettant en évidence l'erreur, et il l'affiche dans une fenêtre nommée *"INPUT PATTERNS"*. Ce contre-exemple est également sauvé dans un fichier spécial qui sera utilisé ultérieurement par l'outil de diagnostic.

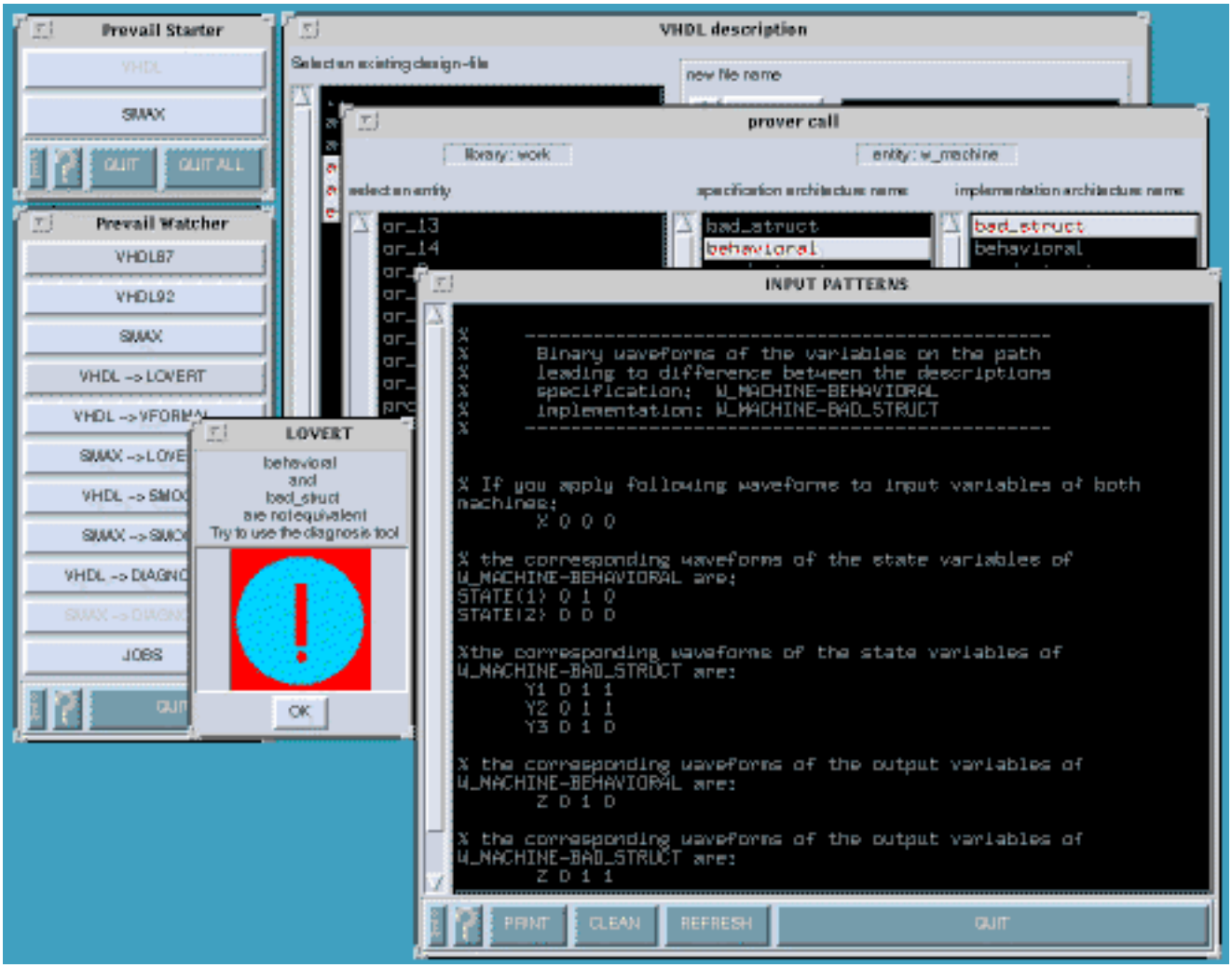

FIG. B.4 - *Le résultat de vérification*

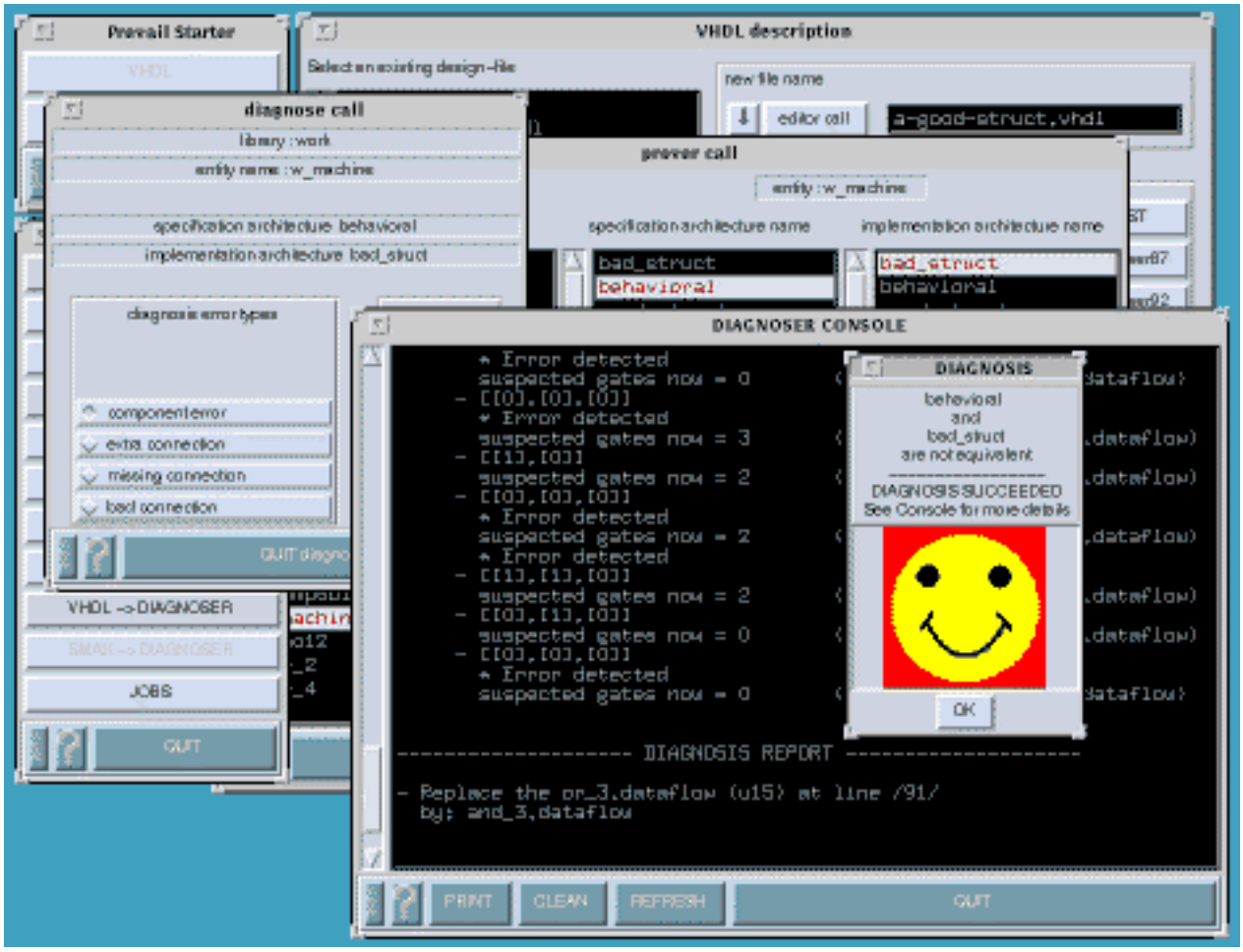

FIG. B.5 - *Le diagnostic*

Si vous décidez d'utiliser l'outil de diagnostic, il vous suffit de cliquer sur OK dans la fenêtre affichant le point d'exclamation, et vous serez guidé vers une autre fenêtre nommée *"diagnose call"* (figure B.5). Avant d'effectuer le diagnostic vous devez choisir une hypothèse d'erreur sous laquelle le diagnostic s'effectuera. Vous avez le choix entre quatre hypothèses:

1. une erreur de composant.
2. une erreur de connexion excédentaire.
3. une erreur de connexion manquante.
4. une erreur de connexion déplacée.

Vous faites votre choix en cliquant sur le bouton correspondant. Une fois que vous avez précisé l'hypothèse d'erreur, cliquez sur GO. S'il s'agit d'un circuit séquentiel un message vous demandera de préciser les états initiaux des deux descriptions. L'état initial par défaut est l'état où tous les bits valent 0, mais vous pouvez changer les valeurs par défaut en cliquant sur le bit correspondant.

Pendant la phase de diagnostic le contre-exemple généré par le vérificateur est utilisé avec d'autres séquences de test spécialement générées pour le diagnostic. A la fin de cette phase, vous aurez deux possibilités:

- Le diagnostic n'est pas possible, et dans ce cas vous aurez une fenêtre affichant un point d'exclamation et un message disant que la correction est impossible sous l'hypothèse d'erreur choisie, et qu'il faut refaire le diagnostic sous une autre hypothèse.

- Une correction est possible, et dans ce cas vous aurez le visage souriant et un message disant que le diagnostic est réussi (figure B.5). Vous aurez dans la fenêtre *"DIAGNOSER CONSOLE"* un rapport de diagnostic donnant les informations suivantes:

  1. Le nom du composant où l'erreur existe.
  2. Le numéro de ligne dans le fichier de description où ce composant est instancié.
  3. La méthode de correction:
     - Le type de composant qui doit remplacer le type actuel (en cas d'erreur de composant).
     - Le nom des connexions à enlever (en cas d'erreur de connexion excédentaire).
     - Le nom de connexion à insérer (en cas d'erreur de connexion manquante).
     - Le nom de connexion à remplacer (en cas d'erreur de connexion déplacée).

Parfois, il se peut que vous ne trouviez pas l'erreur en essayant toutes les hypothèses, ce qui signifie que l'erreur ne suit aucune des hypothèses considérées.

Dans ce cas, le concepteur doit avoir recours aux méthodes manuelles.

Nous donnons dans la suite les descriptions VHDL des architectures utilisées dans l'exemple sur lequel le système a été exécuté et a produit les écrans des figures B.1 à B.5.

Il s'agit d'un petit circuit séquentiel dans lequel la spécification contient 2 variables d'états, et l'implémentation en contient 3. Les codage des états ne sont, dans les deux architectures, reliés par aucune relation simple.

```
-----------------------------------------------------------------------
-- Machine name: w_machine
-- Author      : Ayman Wahba (Ayman.Wahba@imag.fr)
-- Last update : 16/11/1995
-- Description : This example  shows two equivalent  descriptions of
--               the   same  finite  state  machine. The  structural
--               architecture  is  made  directly  from  an  initial
--               design  without  any   state minimization  process.
--               It uses 8 states i.e. 3 state variables.
--               The  behavioral  description  is  written  after  a
--               state minimization process  which led to a  machine
--               with 3 states only. i.e 2 state variables.
--               An error is inserted in the structural architecture
--               by replacing the  component 'u15', which  should be
--               of type 'and3', by another component of type 'or3'.
-----------------------------------------------------------------------
use WORK.PROOF.all;

entity w_machine is
   port ( X: in Bit; CK: in CLOCK; Z: out Bit);
end w_machine;


-----------------------------------------------------------------
------------------------ Specification ------------------------
-----------------------------------------------------------------
ARCHITECTURE behavioral of w_machine IS

-- State encoding
    CONSTANT A: BIT_VECTOR (0 to 1) := B"00";
    CONSTANT B: BIT_VECTOR (0 to 1) := B"01";
    CONSTANT C: BIT_VECTOR (0 to 1) := B"10";
    CONSTANT D: BIT_VECTOR (0 to 1) := B"11";

--Memory elements
   SIGNAL STATE: BIT_VECTOR (0 to 1);

BEGIN
```

```
   PROCESS
   BEGIN
     wait on CK until CK = '1';

      case STATE is
         when A =>
            if (X = '0') then
               STATE <= C;
            else
               STATE <= B;
            end if;
         when B =>
            if (X = '0') then
               STATE <= C;
            else
               STATE <= A;
            end if;
         when  C =>
            if (X = '0') then
               STATE <= A;
            else
               STATE <= C;
            end if;
         when  D =>
            STATE <= D;
      end case;
   END PROCESS;


   Z <= '1'
            when
              (X = '0') and ((STATE = B) or (STATE = C))
            else
        '0';

END behavioral;

-----------------------------------------------------------------
--------------------- Wrong Implementation ----------------------
-----------------------------------------------------------------
```

```
ARCHITECTURE impl_err of w_machine IS

   component and2
      port(input1,input2:In Bit; output:Out Bit);
   end component;
   component and3
      port(input1,input2,input3:In Bit; output:Out Bit);
   end component;

    component or1
      port(input1:In Bit; output:Out Bit);
   end component;
   component or2
      port(input1,input2:In Bit; output:Out Bit);
   end component;
   component or3
      port(input1,input2,input3:In Bit; output:Out Bit);
   end component;

   component not1
      port(input1:In Bit; output:Out Bit);
   end component;

-- configuration for each component used at the top level

   for all:and2 use entity work.and_2(dataflow);
   for all:and3 use entity work.and_3(dataflow);

   for all:or1  use entity work.or_1(dataflow);
   for all:or2  use entity work.or_2(dataflow);
   for all:or3  use entity work.or_3(dataflow);

   for all:not1 use entity work.not_1(dataflow);

-- signals used at the top level

SIGNAL
   U1,U2,U3,U4,U5,U6,U7,U8,U9,U10,
   U11,U12,U13,U14,U15,U16,U17,U18,U19,U20,
   U21,U22,U23,U24,W1,W2,W3: Bit;
```

```
SIGNAL
   Y1,Y2,Y3: BIT := '0';

SIGNAL
   NY1,NY2,NY3,NX: Bit;

BEGIN

  REGVECT : BLOCK (CK='1' AND NOT CK'STABLE)
    BEGIN
      DFF1 : Y1 <= GUARDED W1 ;
      DFF2 : Y2 <= GUARDED W2 ;
      DFF3 : Y3 <= GUARDED W3 ;
    END BLOCK ;

-------------------------------------------
C1 : not1 port map(Y1,NY1);
C2 : not1 port map(Y2,NY2);
C3 : not1 port map(Y3,NY3);
C4 : not1 port map(X,NX);
-------------------------------------------
C5: and2 port map(NY1,NY3,U1);
C6: and2 port map(U1,NX,U2);
C7: and3 port map(Y1,NY3,X,U3);
C8: and3 port map(NY1,Y3,X,U4);
C9: and3 port map(NY1,NY2,Y3,U5);
C10: or2 port map(U2,U3,U6);
C11: or2 port map(U4,U5,U7);
C12: and2 port map(Y1,Y2,U8);
C13: and2 port map(Y3,NX,U9);
C14: and2 port map(U8,U9,U10);
C15: or3 port map(U6,U7,U10,U11);
C16: or1 port map(U11,W1);

C17: and2 port map(NY3,NX,U12);
C18: and2 port map(NY2,NY3,U13);
C19: and2 port map(NY2,NX,U14);
C20: or3 port map(Y2,Y3,X,U15);
C21: or2 port map(U12,U13,U16);
```

```
C22: or2  port map(U14,U15,U17);
C23: or2 port map(U16,U17,U18);
C24: or1 port map(U18,W2);

C25: and2 port map(Y3,X,U19);
C26: and2 port map(NY2,Y3,U20);
C27: or3 port map(U12,U19,U20,U21);
C28: or1 port map(U21,W3);

C29: and2 port map(Y2,NX,U22);
C30: and2 port map(U9,Y1,U23);
C31: or2  port map(U22,U23,U24);
C32: or1 port map(U24,Z);

END impl_err;
```

Résumé

Le diagnostic automatique des erreurs de conception est un problème important dans le domaine de la CAO. Bien que des outils automatisés de synthèse soient employés pour générer des structures de circuits *correctes-par-construction*, celles-ci sont souvent modifiées manuellement pour refléter des petites modifications faites sur la spécification, ou pour améliorer certaines caractéristiques critiques de la conception. Les outils de vérification peuvent révéler l'existence d'erreurs, mais ils ne donnent aucune information sur leurs emplacements ou la façon de les corriger. Ces outils génèrent seulement quelques contres-exemples qui mettent en évidence l'erreur. Les concepteurs utilisent ces contre-exemples pour diagnostiquer manuellement leur conception. Le diagnostic manuel est un processus très lent et très coûteux. Le temps de diagnostic peut être égal, voire supérieur, au temps de conception.

Nous présentons dans cette thèse de nouveaux algorithmes pour la localisation et la correction automatique des erreurs simples de conception dans les circuits logiques sous l'hypothèse d'une seule erreur. Les erreurs traitées ici sont: le remplacement d'un composant dans les circuits combinatoires et séquentiels, et une erreur de connexion dans les circuits combinatoires. Le modèle d'une seule erreur exige une stratégie de *vérification fréquente*, dans laquelle la conception est vérifiée après chaque modification, pour que la probabilité d'insertion de plus d'une erreur ne soit pas trop élevée. Notre approche consiste à simuler et analyser automatiquement le circuit sous l'application de vecteurs de test que nous produisons spécialement pour accélérer le diagnostic. Nous avons réalisé deux logiciels prototypes basés sur ces algorithmes. CCDS est l'outil de diagnostic pour les circuits combinatoires, et SCDS est l'outil de diagnostic pour les circuits séquentiels. Ces outils sont actuellement intégrés dans l'environnement de preuves PREVAIL^{TM}.

Abstract

Automatic diagnosis of design errors is an important problem in digital circuits CAD. Although automated synthesis tools are being used to generate *correct-by-construction* designs, these designs are still modified manually to perform small specification changes, or to enhance some critical design aspects. Verification tools can discover the existence of errors, but they give no information about their location or how to correct them, they give only counter-examples that witness the error. What designers commonly do is simulate their design with these counter-examples, and visually inspect the ciruit and trace each path in it. This operation takes a very long time which may be equal or even greater than the design time itself.

In this thesis we present automated algorithms for the location and the correction of simple design errors in logic circuits under the single error hypothesis. The errors treated here are component replacement errors in combinational and sequential circuits, and connection errors in combinational circuits. The single error model implicitly implies a *frequent verification strategy*, in which the design is verified frequently after each design change, so that the probability of inserting more than one error is not very high. Our approach consists in simulating the circuit under the application of a given test pattern, and then scanning it to analyse its components. To accelerate the process we use special test patterns generated for the diagnosis. Two prototype diagnosis tools are built based on these algorithms. CCDS is the diagnosis tool for combinational circuits, and SCDS is the diagnosis tool for sequential circuits. CCDS and SCDS are currently integrated in the proof environment PREVAIL^{TM}.