View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Espace ÉTS

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

THÈSE PRÉSENTÉE À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DU
DOCTORAT EN GÉNIE
Ph.D.

PAR
Christelle HOBEIKA

MÉTHODOLOGIE DE VÉRIFICATION AUTOMATIQUE BASÉE SUR L'UTILISATION DES TESTS STRUCTURELS DE TRANSITION AVEC INSERTION DE REGISTRES À BALAYAGE

MONTRÉAL, LE 4 OCTOBRE 2011

© Tous droits réservés, Christelle Hobeika, 2011

©Tous droits réservés
Cette licence signifie qu'il est interdit de reproduire, d'enregistrer ou de diffuser en tout ou en partie, le présent document. Le lecteur qui désire imprimer ou conserver sur un autre media une partie importante de ce document, doit obligatoirement en demander l'autorisation à l'auteur.

PRÉSENTATION DU JURY

CETTE THÈSE A ÉTÉ ÉVALUÉE

PAR UN JURY COMPOSÉ DE :

M. Claude Thibeault, directeur de thèse
Département de génie électrique à l'École de technologie supérieure

M. Jean-François Boland, codirecteur
Département de génie électrique à l'École de technologie supérieure

M.Stéphane Coulombe, président du jury
Département de génie logiciel et des TI à l'École de technologie supérieure

M. Marcel Gabrea, membre du jury
Département de génie électrique à l'École de technologie supérieure

M. Bruno De Kelper, membre du jury
Département de génie électrique à l'École de technologie supérieure

M. Othmane Ait Mohamed, membre du jury
Département de génie électrique à l'université concordia

ELLE A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 29 SEPTEMBRE 2011

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Tout d’abord, mes plus sincères remerciements et reconnaissances s’adressent principalement à mon directeur de thèse, le Professeur Claude Thibeault pour son orientation, ses conseils et aussi son soutien moral et financier tout le long de ces années de thèse. Le Professeur Thibeault m’a accompagné tout au long de ma thèse et, par sa présence constante et ses conseils avisés, a grandement contribué à mon développement professionnel, en tant que chercheur, et personnel. J’ai pu accomplir mon doctorat avec succès grâce à ses connaissances, sa sagesse et ses excellentes aptitudes de professeur, chercheur et ami. Je le remercie profondément pour tout ce qu’il a fait pour moi tout au long de ces 5 ans.

J’aimerai remercier aussi mon co-directeur le professeur Jean-François Boland pour son soutien et ses conseils. Je remercie également les membres du jury pour leur collaboration durant l'examen de ce travail, leurs conseils et leur participation à la soutenance. Mes sincères remerciements à tout le personnel du département de génie électrique.

Je remercie aussi tous les collègues et amis, particulièrement ceux du LACIME, du département de génie électrique et de l’association étudiante ainsi que toutes les personnes côtoyées à l’ÉTS. De plus, des remerciements spéciaux pour tous mes amis au Canada et au Liban pour leur présence, leur encouragement, surtout leur amitié pendant toutes ces années. Un remerciement spécial à Mathieu, qui a été le premier à m’accueillir au laboratoire, et qui a partagé avec moi chaque moment de réussite et d’échec depuis le début de ma thèse. Merci pour ton amitié et ton grand amour.

Finalement, à mes parents François et Sonia et mon frère Georges qui m’ont donné l’amour, l’éducation et tout ce qui fait de moi la personne que je suis aujourd’hui. Malgré les longues années de distance, leur soutien, leur affection et surtout leur amour n’a fait qu’augmenter et m’a accompagné tout au long de cette thèse. Je vous aime énormément et je vous remercie pour tout ce que vous avez fait pour moi, cette thèse est la vôtre aussi.
À toute ma famille au Liban, un grand merci.

MÉTHODOLOGIE DE VÉRIFICATION AUTOMATIQUE BASÉE SUR L'UTILISATION DES TESTS STRUCTURELS DE TRANSITION AVEC INSERTION DE REGISTRES À BALAYAGE

Christelle HOBEIKA

RÉSUMÉ

Au cours des dernières décennies, l'évolution de la technologie n'a cessé d'introduire de nouveaux défis dans la vérification des circuits intégrés (IC). L'industrie estime que la vérification fonctionnelle prend environ 50% à 70% de l'effort total d'un projet. Et, malgré les budgets et les efforts investis dans la vérification, les résultats obtenus ne sont pas satisfaisants. La vérification basée sur la simulation, également appelée vérification dynamique, est la technique la plus utilisée dans la vérification fonctionnelle. Par contre, ce type de vérification a clairement échoué à suivre le rythme de croissance de la complexité. Par conséquent, des solutions innovantes sont requises, avec la concurrence sur les produits et les services ainsi que l'implacable loi du temps de mise sur le marché.

Plusieurs techniques ont été développées pour surmonter les défis de la vérification dynamique, allant de techniques entièrement manuelles à des techniques plus avancées. Les techniques manuelles et semi-manuelles ne peuvent être utilisées pour les designs complexes, et les approches les plus avancées qui sont couramment utilisés dans l'industrie ont besoin de compétences particulières et beaucoup d'efforts afin d'atteindre une bonne productivité de vérification.

Au niveau du test par contre, l'utilisation d'approches basées sur des modèles de pannes et sur les concepts de conception en vue du test (DFT), a conduit au développement d'outils automatiques de génération de test (ATPG) efficaces. L'infrastructure de test qui en résulte a grandement aidé la communauté du test à résoudre plusieurs problèmes.

Dans cette thèse, nous nous intéressons principalement à la productivité du processus de vérification, plus particulièrement la vérification de circuits séquentiels. Nous proposons une nouvelle méthodologie qui explore la combinaison du test et de la vérification, plus précisément l'utilisation des tests structurels de transition dans le processus de vérification RT basée sur la simulation. Cette méthodologie a pour but de réduire le temps et les efforts requis pour vérifier un circuit et d'améliorer la couverture résultante, induisant des améliorations significatives de la qualité de la vérification et de sa productivité. La base de la méthodologie proposée est l'intuition (qui est devenu une observation), selon laquelle ce qui est difficile à tester (« Hard Fault ») est probablement difficile à vérifier (« Dark Corner »). L'objectif est de tirer profit des outils de test efficaces tels que les outils ATPG, et les techniques DFT tels que l'insertion des registres a balayage afin de simuler efficacement la fonctionnalité du design avec un minimum de temps et d'efforts. Sur la base de tous ces concepts, nous avons développé un environnement de vérification RTL automatisé composé de trois outils de base: 1) un extracteur de contraintes qui identifie les contraintes

fonctionnelles de conception, 2) un outil générateur de banc d'essai, et 3) un détecteur d’erreurs basé sur une observabilité élevée.

Les résultats expérimentaux montrent l'efficacité de la méthode de vérification proposée. Les couvertures de code et d’erreurs obtenues suite à la simulation avec l’environnement proposé sont égales à, et la plupart des fois plus élevé que, celles obtenues avec d'autres approches connues de vérification. En plus des améliorations de couverture, il y a une réduction remarquable de l'effort et du temps nécessaire pour vérifier les designs.

Mots clés : Vérification fonctionnelle, test, simulation, contraintes fonctionnelles, automatisation, RTL.

MÉTHODOLOGIE DE VÉRIFICATION AUTOMATIQUE BASÉE SUR L'UTILISATION DES TESTS STRUCTURELS DE TRANSITION AVEC INSERTION DE REGISTRES À BALAYAGE

Christelle HOBEIKA

ABSTRACT

Over the last few decades, technology scaling has continuously brought new challenges to the research community, from integrated circuit (IC) design to IC testing. Industry estimates that functional verification takes approximately 50% to 70% of the total effort on a project. And even with verification budgets dominating design budgets, there are increasingly more bug escapes through fabrication and consequently expensive re-spins. Verification methodologies are grouped into two main categories: 1) Simulation-based methods and 2) formal methods. Although both methodologies are now widely established for design verification, simulation-based verification remains the most commonly used technique. Yet, simulation has clearly failed to keep pace with complexity and faces lots of challenges. Therefore, with the continuous competition growth in products and services along with the harsh law of time to market, innovative solutions are required. Hence, a transition to new methodologies and tools is deemed crucial.

Several techniques have been developed to overcome simulation-based verification challenges, ranging from fully manual to advanced testbenches. The manual and semi-manual techniques cannot be scaled on complex designs, and the more advanced approaches that are commonly used in the industry, need special skills and human interaction to achieve good verification productivity.

From a test perspective, the use of structural approaches based on fault models and on design for testability (DFT) concepts (namely scan-based), has led to the development of efficient automatic test pattern generation (ATPG) tools. The resulting test infrastructure has greatly helped the test community to address previous encountered issues and to face the incoming ones.

In this thesis, we are primarily concerned with the productivity of the verification process, more specifically the verification of sequential circuits. We propose a new methodology that explores a new test/verification combination, namely the use of structural test patterns in the RT simulation-based verification process. This methodology is aimed to reduce time and effort required to verify a circuit and to improve the resulting coverage, inducing significant improvements in verification quality and productivity. The cornerstone of the proposed methodology is the intuition (that became an observation) according to which a node that is difficult to test (Hard Fault) is likely difficult to verify (Dark Corner). The goal is to take advantage of the efficient test tools as ATPG, and the advanced test techniques as scan-based DFT to simulate and exercise efficiently the model functionality with minimum time and effort. Based on all these concepts, we introduced an automated RTL verification environment

composed of three basic tools: 1) A constraint extractor that identifies design's functional constraints, 2) a test bench generator tool, and 3) an error tracker based on a high observability.

Experimental results showed the effectiveness of the proposed verification methodology. It could fast provide fault and code coverage that are equal to and even higher than one obtained with other well known simulation-based verification approaches. In addition to the coverage improvements, there is a remarkable reduction in effort and time needed to verify the designs. Unlike the other methodologies, the proposed approach requires little effort, in order to accomplish the simulation.

Keywords: Functional verification, test, simulation, functional constraints, automation, RTL.

TABLE DES MATIÈRES

Page

INTRODUCTION 23

CHAPITRE 1 TEST ET VÉRIFICATION 33
1.1 Introduction...33
1.2 Notions de base de la vérification 33
1.2.1 Méthodologie de vérification 34
1.2.2 Vérification dynamique 35
1.2.3 Vérification statique 38
1.3 Notions de base du test 41
1.3.1 Modèles de panne 42
1.3.2 Génération automatique de vecteurs de test 44
1.3.3 Conception en vue du test 44
1.3.4 Tests de transition avec insertion des registres à balayage 46
1.4 Conclusion 48

CHAPITRE 2 REVUE DE LITTÉRATURE 49
2.1 Vérification dynamique 49
2.1.1 Vérification aléatoire basée sur les contraintes 49
2.1.2 Vérification basée sur les assertions 52
2.1.3 Vérification transactionnelle 53
2.2 Combinaison du test et de la vérification 54
2.3 Les modèles d'erreurs RTL 56
2.4 États illégaux d'un design 58
2.5 Conclusion 59

CHAPITRE 3 UTILISATION DES TESTS STRUCTURELS DANS LA SIMULATION 61
3.1 Introduction...61
3.2 Pannes cachées versus coins sombres 62
3.2.1 Pannes cachées 62
3.2.2 Coins sombres 63
3.2.3 Corrélation entre les coins sombres et les pannes cachées 64
3.3 Le choix des tests structurels 66
3.4 Utilisation des vecteurs de test structurels dans la vérification RTL 68
3.4.1 Génération des vecteurs de test structurels 69
3.4.2 Application des tests de transition LOC avec registres à balayage au modèle RTL 70
3.5 Premiers résultats expérimentaux 75
3.5.1 Corrélation entre les coins sombres et les pannes cachées 75
3.5.1.1 Expériences 75
3.5.1.2 Conceptions 78

3.5.1.3 Les résultats expérimentaux 80
3.5.2 Utilisation des vecteurs de test dans la vérification 82
3.6 Conclusion 83

CHAPITRE 4 ÉTUDE THÉORIQUE DE LA DÉTECTION DES ERREURS DU DESIGN AU NIVEAU RTL À L'AIDE DES TESTS DE TRANSITION LOC 85
4.1 Introduction 85
4.2 Hypothèses 86
4.3 Détection des erreurs RTL à l'aide des vecteurs structurels LOC et limitations 86
4.3.1 Le modèle BOE 87
4.3.2 Le modèle BSE 89
4.3.3 Le modèle BDE 90
4.3.4 Le modèle MSE 91
4.3.5 Le modèle BCE 92
4.3.6 Le modèle MCE 93
4.3.7 Le modèle LCE 95
4.3.8 Le modèle ESE 97
4.3.9 Les erreurs FSM 97
4.3.9.1 Le modèle SCE 98
4.3.9.2 Le modèle NSE 100
4.4 Test de transition versus *stuck-at* 102
4.5 Les états illégaux 105
4.5.1 Effet des états illégaux 105
4.5.2 Exemple : compteur 106
4.5.3 Les fausses erreurs 109
4.6 Conclusion 110

CHAPITRE 5 EXTRACTION AUTOMATIQUE DES CONTRAINTES FONCTIONNELLES À PARTIR D'UN MODÈLE RTL 111
5.1 Introduction 111
5.2 Grammaire d'analyse d'expression VHDL 112
5.2.1 VHDL 113
5.2.2 Grammaire d'analyse d'expression 113
5.2.3 La PEG VHDL 115
5.3 Approche proposée 117
5.4 Implémentation détaillée 119
5.4.1 Analyse du VHDL et identification des différentes structures 120
5.4.1.1 Identification des structures VHDL 122
5.4.1.2 Identification des valeurs initiales des différents signaux d'états 125
5.4.1.3 Identification des HLS 125
5.4.2 Extraction des HLS des module 127
5.4.2.1 Évaluation des opérations 127
5.4.2.2 Implémentation détaillée 131

5.4.3 Analyse hiérarchique du design ... 134
5.4.3.1 Analyse de connectivité des modules ... 134
5.4.3.2 Modèle aplati ... 137
5.4.4 Extraction des HLS du design ... 138
5.4.5 Extraction des contraintes fonctionnelles ... 139
5.5 Résultats expérimentaux ... 140
5.5.1 La vérification ... 141
5.5.2 Le test ... 142
5.6 Conclusion ... 143

CHAPITRE 6 ENVIRONNEMENT DE VÉRIFICATION COMPLET PROPOSÉ ... 145
6.1 Introduction ... 145
6.2 Environnement de vérification proposé ... 145
6.2.1 Le générateur automatique du banc d'essai ... 146
6.2.2 Le détecteur d'erreurs ... 148
6.3 Résultats expérimentaux ... 148
6.4 Conclusion ... 151

CONCLUSION ... 153

RECOMMANDATIONS ... 157

ANNEXE I LES MODÈLES D'UN SYSTÈME ... 159

ANNEXE II CODE PERL DU GÉNÉRATEUR DU BANC D'ESSAI ... 161

ANNEXE III CODE PERL DU DÉTECTEUR D'ERREURS ... 185

ANNEXE IV CODE PERL DE L'EXTRACTEUR DES CONTRAINTES FONCTIONNELLES ... 187

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES ... 223

LISTE DES TABLEAUX

Page

Tableau 3.1 L'ensemble des transitions du design 1, avec Tm= Transition montante et Td= transition descendante. 79

Tableau 3.2 L'ensemble des transitions possibles du design 2, avec Tm= Transition montante et Td= transition descendante. 80

Tableau 3.3 Comparaison des couvertures des différentes techniques de vérification. 82

Tableau 4.1 Les différents tests de transition possibles du nœud X_i, basés sur les comportements de X_i et X_j. 88

Tableau 4.2 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test de transition LOC du nœud X_i, dans le cas d'une erreur de type BOE. 88

Tableau 4.3 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test de transition LOC du nœud A_i, dans le cas d'une erreur de type BSE..90

Tableau 4.4 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test de transition LOC du nœud X_i, dans le cas d'une erreur de type BCE. 93

Tableau 4.5 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test de transition LOC du nœud A_i, dans le cas de la suppression d'un module dans le design.........................95

Tableau 4.6 Exemple d'erreur de type LCE. 96

Tableau 4.7 Les différents tests possibles s@1 (s@0) du nœud X_i, basés sur les comportements de X_i et X_j. 103

Tableau 4.8 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test s@1 du nœud X_i, dans le cas d'une erreur de type BOE. 103

Tableau 4.9 Le test de transition test 3. 103

Tableau 4.10 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test de transition test 3, dans le cas d'une erreur de type BOE. 104

Tableau 4.11 Différence entre les probabilités de détection d'erreurs des tests de transition et celle des tests « *stuck-at* ». 104

Tableau 5.1 Opérateurs de construction d'expression d'analyse....114

Tableau 5.2 L'ensemble des règles d'analyse VHDL. 116

Tableau 5.3 Identification des différentes structures VHDL. 122

Tableau 5.4 Valeurs initiales des signaux d'état basées sur l'exemple de la Figure 5.1. 125

Tableau 5.5 HLS correspondants à l'exemple de la Figure 5.1. 127

Tableau 5.6 Contraintes des signaux d'états. 130

Tableau 5.7 Affectation des signaux d'états. 130

Tableau 5.8 HLS. 132

Tableau 5.9 Procédures, paramètres et code. 132

Tableau 5.10 Contraintes des signaux d'états. 133

Tableau 5.11 Affectation des signaux d'états. 134

Tableau 5.12 HLS. 134

Tableau 5.13 Les représentations des entités du design de la Figure 5.6. 135

Tableau 5.14 Les composants de l'entité Globale du design de la Figure 5.6....135

Tableau 5.15 Les instances de l'entité globale du design de la Figure 5.6. 136

Tableau 5.16 Les composants de E2 du design de la Figure 5.6. 136

Tableau 5.17 Les instances de E2 du design de la Figure 5.6. 136

Tableau 5.18 Structure des données représentant les connexions du design de la Figure 5.6. 137

Tableau 5.19 Caractéristiques des circuits ITC'99 utilisés....141

Tableau 5.20 Nombre de fausses erreurs détectées sans contraintes. 142

Tableau 5.21 ATPG avec contraintes versus sans contraintes....143

Tableau 6.1 Caractéristiques des circuits expérimentaux. 149

Tableau 6.2 Comparaison des couvertures d'états et de transitions. 150

Tableau 6.3 Comparaison des couvertures d'erreurs. 150

LISTE DES FIGURES

Page

Figure 0.1 Écart de productivité. 25

Figure 0.2 Classification des pannes basée sur leur détectabilité. 29

Figure 1.1 Flux de conception et de vérification. 34

Figure 1.2 La vérification dynamique. 36

Figure 1.3 Test à balayage en série. 46

Figure 2.1 Environnement de vérification aléatoire basé sur les contraintes. 51

Figure 2.2 Vérification transactionnelle. 54

Figure 3.1 Flux de conception standard implémentant l'utilisation des tests structurels dans la vérification. 69

Figure 3.2 Test de transition STF (STR) sur un nœud X_i. 71

Figure 3.3 Exemple d'un modèle de test LOC avec registres à balayage. 72

Figure 3.4 Vecteur de simulation RTL formé à partir du modèle de test LOC de la Figure 3.3. 74

Figure 3.5 Description RTL de l'entité du circuit 1....................78

Figure 3.6 Le diagramme TRIO correspondant au circuit 1. 79

Figure 3.7 Description RTL de l'entité du circuit 2....................80

Figure 3.8 Superposition des résultats des deux niveaux RT et porte logique pour le circuit 1. 81

Figure 3.9 Superposition des résultats des deux niveaux RT et porte logique pour le circuit 2. 81

Figure 4.1 Exemple d'erreur de type BOE. 87

Figure 4.2 Exemple d'erreur de type BSE. 89

Figure 4.3 Exemple d'erreur de type BDE. 91

Figure 4.4 Exemple d'erreur de type MSE....................91

Figure 4.5 Exemple d’erreur de type BCE. 92

Figure 4.6 Exemple d’erreur de type MCE. 94

Figure 4.7 Exemple d’erreur de type ESE. 97

Figure 4.8 Exemple d’erreur de type NSE, états supplémentaires. 99

Figure 4.9 Exemple d’erreur de type NSE, états manquants....................................99

Figure 4.10 Exemple d’erreur de type NSE. 101

Figure 4.11 Exemple de deux codes VHDL différents d’un compteur. 107

Figure 4.12 Les machines à états correspondants aux 2 styles de code cas a et cas b du compteur 2 bits présenté dans l’exemple ci-dessus.107Thèse_corrigé_apresoutenance_v3

Figure 4.13 Vecteurs de test ATPG correspondant à l’exemple de la Figure 4.11. 108

Figure 4.14 Résultat de la simulation du compteur (cas a) avec les vecteurs de test de transition.108Thèse_corrigé_apresoutenance_v3.doc - _Toc305486240

Figure 4.15 Résultat de la simulation du compteur (cas b) avec les vecteurs de test de transition. 109

Figure 5.1 Exemple de code VHDL. 115

Figure 5.2 Extraction des contraintes fonctionnelles d'un design VHDL. 118

Figure 5.3 Identification des structures VHDL d’un module..................................121

Figure 5.4 Les représentations de a) entité b) composant c) signal_type d) instance e) affectation de signal f) structure if g) structure case g) condition i) opération. 124

Figure 5.5 Détails de l’implémentation de la procédure *high_level_state_identification.* 126

Figure 5.6 Exemple d’un design hiérarchique...135

Figure 5.7 Pseudo-code de la création d'un modèle aplati. 137

Figure 5.8 Modèle aplati du design de la Figure 5.6. 138

Figure 5.9 Pseudo-code de l'extraction des contraintes fonctionnelles du design. ...139

Figure 6.1 L'environnement de vérification proposé. 146

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ATPG	Automatic test pattern generation
BCD	Binary coded decimal
BDD	Binary decision diagram
BIST	Built in self test
BOE	Bus order error
BDE	Bus driver error
BCE	Bus count error
CC	Contrôlabilité combinatoire
CS	Contrôlabilité séquentielle
DUV	Design under verification
DFT	Design for testability
ESE	Expression structure error
FSM	Finite state machine
HLS	High level state
HDL	Hardware description langage
LOC	Launch on capture
LCE	Label count error
MSE	Module substitution error
MCE	Module count error
NSE	Next state error
OC	Observabilité combinatoire

OS	Observabilité séquentielle
OSCI	Open systemC intiative
OVL	Open verification library
PSL	Property specification langage
PEG	Parsing expression grammar
RT	Register transfer
RTL	Register transfer level
ST-FU	Structurally testable functionally untestable
STR	Slow to rise
STF	Slow to fall
SCE	State count error
SVA	SystemVerilog assertions
TVM	Transaction verification model
TRIO	Input output transition
QVL	Questa verification library

INTRODUCTION

Le processus d'intégration des puces électroniques est divisé en 4 phases majeures : la conception, la vérification, la fabrication et le test. La conception permet de passer du "système" au "silicium", via plusieurs niveaux d'abstraction, allant du modèle dit de haut niveau correspondant à une description fonctionnelle du circuit, au modèle dit de bas niveau correspondant à l'élaboration des plans des masques *(« layout »)* définissant la topologie du circuit. Pour chaque étape de conception, diverses opérations doivent être effectuées pour s'assurer que le travail est bien fait et conforme aux spécifications. On retrouve dans la littérature deux termes représentant ces opérations : validation et vérification. Même s'il existe plusieurs définitions différentes pour ces termes, les plus courantes sont les suivantes (Sommerville, 2000): la validation est la série d'opérations par laquelle on s'assure qu'on conçoit le bon produit (i.e. correspondant aux spécifications fonctionnelles) alors que la vérification est la série d'opérations par laquelle on s'assure que le produit est intégré correctement. Dans le cadre de cette thèse, nous considérons que la validation est la série d'opérations effectuées à haut niveau d'abstraction pour confirmer le choix des algorithmes de traitement, via par exemple des simulations faites à l'aide d'un outil tel que Matlab. Nous considérons également que la vérification est la série d'opérations effectuées pour : 1) s'assurer de l'équivalence entre deux niveaux d'abstraction différents au niveau de la fonctionnalité, et 2) s'assurer du respect des spécifications dites non fonctionnelles, telles la fréquence maximale d'opération, la puissance dissipée, le respect des règles du dessin des maques, etc. Dans le cadre de cette thèse, nous nous intéressons au premier type de vérification, que nous appellerons vérification fonctionnelle.

C'est après la fabrication du circuit que le test survient. Il permet de détecter les défectuosités et les pannes issus des procédés de fabrication afin d'assurer une bonne qualité des produits avant leur mise en marché. Depuis plusieurs décennies, la croissance de la complexité des systèmes numériques est en évolution constante. Au niveau du test, l'utilisation d'approches basées sur des modèles de pannes et sur les concepts de conception en vue du test (DFT*, Design For Testability*) a conduit au développement d'outils de

génération automatique de vecteurs de tests (ATPG, *Automatic Test Pattern Generation*) efficaces (Rabaey, 1996). L'infrastructure de test résultante a permis de palier aux principaux problèmes dus à la croissance de complexité. Dans le cadre de cette thèse, nous proposons de tirer avantage de la maturité de l'infrastructure de test afin d'améliorer la vérification.

En revanche, au niveau de la vérification, la complexité ne cesse de s'accentuer et s'accroit exponentiellement avec le temps. Ceci est du en grande partie à l'absence de procédés automatiques et efficaces permettant de faire face à la croissance de la complexité et de la taille des designs (Dempster, 2001). Cette problématique, qui constitue la problématique principale à laquelle s'attaque cette thèse, est décrite plus en détails dans ce qui suit. Notons également que l'utilisation proposée de l'infrastructure de test pour améliorer la vérification nous oblige à nous intéresser à une problématique secondaire, l'élimination des états illégaux des vecteurs de test. Cette problématique sera également définie plus loin.

Problématique principale

Le développement continu de la technologie induit une augmentation de la complexité et du niveau d'intégration d'un design. En 1965, Gordon Moore a prédit que le nombre de transistors pouvant être intégrés sur une puce augmenterait de manière exponentielle avec le temps. La tendance prédite s'est matérialisée et est devenue la fameuse « loi de Moore ». Nous sommes par exemple passés de 10^5 transistors par chip en 1982 à 10^9 en 2009, ce qui a conduit à des systèmes de plus en plus complexes. L'augmentation de la complexité des dispositifs, que ce soit en termes du nombre de transistors ou encore en termes d'intégration de nouveaux types de modules (analogique, RF, MEMS, …) conduit, outre les problèmes de conception, à de nombreux défis à relever dans le domaine de la vérification. La croissance de la complexité du design induit une augmentation de la complexité de presque toutes les étapes de la vérification allant de l'analyse du modèle jusqu'à l'analyse des résultats de la vérification, conduisant à une augmentation du temps alloué à la vérification et une diminution de la qualité de cette dernière. Dans le cas de la vérification fonctionnelle, comme la stratégie la plus utilisée demeure la simulation, l'explosion du temps se situe à ce niveau.

Le temps et les efforts investis ne se limitent pas à la définition des bancs d'essai et à la simulation en tant que tel, mais aussi à la compréhension et à l'analyse du modèle à simuler ainsi que de son contexte d'utilisation afin de bien vérifier sa fonctionnalité. En général, dans un projet de conception de circuit numérique, la conception et la vérification sont prises en charge par des équipes différentes. La Figure 0.1 montre l'écart grandissant qui se creuse entre la croissance de la complexité d'un circuit d'une part et la croissance de la productivité d'intégration d'une autre part. Cet écart est dû en grande partie à la vérification fonctionnelle, étant donné qu'elle occupe la majeure partie du temps d'un projet (Molina, 2007).

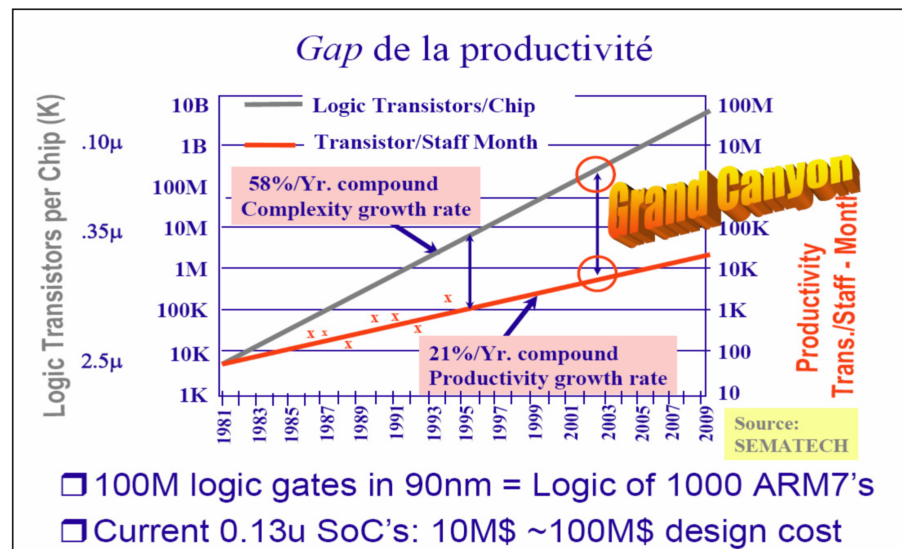


Figure 0.1 Écart de productivité.
Tirée de (Sematech, 1999)

Malgré les diverses méthodologies présentes dans la littérature et qui seront détaillées dans le chapitre 2 de cette thèse, la vérification présente encore plusieurs défis à relever, et ce à différents niveaux :

- le temps alloué à la vérification est très long : en plus du temps de la simulation, s'ajoute le temps d'analyse du modèle, de conception des bancs d'essai et de débogage. La phase de vérification occupe 50% à 70% du temps total d'un projet de conception (Molina,

2007), créant un délai majeur avant la mise sur le marché du produit. Et donc, la réduction du temps alloué à la vérification est devenue primordial pour les entreprises;

- les efforts alloués à la vérification sont colossaux : dépendamment de la méthodologie utilisée, les ingénieurs de vérification déploient beaucoup d'efforts à différentes étapes de la vérification :
  - la lecture, l'analyse et la compréhension du modèle implémenté par les concepteurs;
  - la génération d'un ensemble de bancs d'essai (i.e. vecteurs de simulation) efficaces permettant de couvrir le plus d'erreurs possibles en un temps limité;
  - la détection des erreurs et le débogage;
  - la prédiction et l'analyse de la couverture.
- l'espace d'états des modèles séquentiels est énorme. La taille de l'espace d'états est le défi premier de la vérification. Pour vérifier la fonctionnalité d'un design, l'ingénieur de vérification doit s'assurer que chaque état courant possible et chaque combinaison d'entrée possible permettent une transition du système vers l'état suivant désirable. Un grand espace d'états requiert de longues et complexes séquences de stimuli induisant une simulation longue et complexe;
- la qualité de vérification et de la couverture atteinte est insatisfaisante. Selon une enquête menée par Collett International Research Inc en 2002 (Inc., 2002), 60% de la production défectueuse contient des défauts fonctionnels ou logiques; ce pourcentage a atteint 70% selon une étude menée par FarWest Research en 2007 (FarWest Research, 2007). Parmi ces designs défectueux, plusieurs d'entre eux avaient des erreurs de conception pouvant être détectées lors de la vérification : respectivement 70 et 78% en 2002 et 2007. Il est presque impossible de vérifier complètement un design, et une bonne couverture nécessite beaucoup de temps et d'efforts. Il est très difficile de produire un ensemble de vecteurs de simulation assez diversifié pour couvrir le plus de types et de nombre d'erreurs possibles et assez limité pour permettre une simulation de durée acceptable.

En conclusion, la phase de vérification est une phase complexe et exigeante qui affecte la productivité du design des circuits intégrés et consomme énormément de temps et d'efforts

sans toutefois atteindre des résultats satisfaisants. Bon nombre de problèmes associés aux méthodes de vérification fonctionnelles d'aujourd'hui sont causées par l'absence d'une automatisation efficace pour lutter contre la croissance rapide de la taille et la complexité des designs.

Problématique secondaire

La méthodologie de vérification proposée dans cette thèse cherche à tirer profit de l'infrastructure avancée du test afin d'améliorer la vérification et surmonter les problèmes décrits précédemment. L'utilisation du test au niveau de la vérification impose une problématique secondaire, celle-ci étant l'existence des états illégaux dans les vecteurs de test.

Une machine à états modélisant le comportement d'un circuit séquentiel est une représentation de l'ensemble d'états valides de ce dernier ainsi que l'ensemble des valeurs d'entrées et de leurs combinaisons possibles permettant de transiter d'un état à l'autre. Dans la plupart des cas, une grande partie des espaces d'états des circuits séquentiels n'est pas utilisée. Ces états ne peuvent jamais se produire en mode fonctionnel : il s'agit des états illégaux. Prenons l'exemple d'un compteur 4 bits décimal codé binaire (BCD, « *Binary Coded Decimal* ») : sa sortie s'incrémente de 0 à 9, les états contenant les valeurs 10 à 16 correspondent à des états invalides appelés états illégaux. Bien que ces valeurs fassent partie de l'espace d'états de la sortie, ils ne se produisent jamais en mode fonctionnel.

La technique de DFT basée sur l'insertion des registres à balayage, présentée plus en détails dans cette thèse, permet l'application d'une valeur quelconque sur les nœuds internes d'un circuit. Ceci présente plusieurs avantages au niveau de l'observabilité et de la contrôlabilité d'un circuit, mais peut causer des problèmes au niveau du test et de la vérification lorsque les valeurs appliquées correspondent à des états illégaux et ne peuvent avoir lieu en mode fonctionnel.

Le problème au niveau de la vérification a été soulevé durant le développement de la méthodologie proposée dans cette thèse. La présence d'états illégaux peut dans certaines conditions causer des comportements différents du design en vérification (DUV, « *Design Under Verification* »), et du modèle de référence, causant ainsi la détection de fausses erreurs. Le problème au niveau de la vérification sera présenté en détails dans le chapitre 4 alors que le chapitre 5 présentera la méthodologie proposée pour y remédier.

De plus, la présence des états illégaux dans l'ensemble des vecteurs de test crée des problèmes au niveau du test. En effet, ces vecteurs de test causent la détection de pannes fonctionnellement non testables (ST-FU, *« Structurally Testable Functionally Untestable »),* d'où le phénomène de surtest (« *overtesting* »). La raison pour laquelle une panne structurellement testable est fonctionnellement non testable est l'existence de contraintes au niveau des fonctions et des opérations du design, appelées contraintes fonctionnelles. Des résultats expérimentaux ont montré l'existence des pannes ST-FU (Krstic, 2003; Rearick, 2001). La Figure 0.2 montre la classification des pannes dans un design (Lin, 2006). Le surtest résulte en une perte du rendement qui peut être significative. Toutes les méthodes de test basées sur les registres à balayage font face au surtestage. Les pannes ST-FU sont détectées si l'état de départ est un état illégal. Pour minimiser le surtestage, en plus de l'identification des pannes fonctionnellement non testables, il faut également identifier les états illégaux. En général, l'identification des pannes ST-FU présente une complexité qui augmente exponentiellement avec la taille du circuit (Lin, 2005).

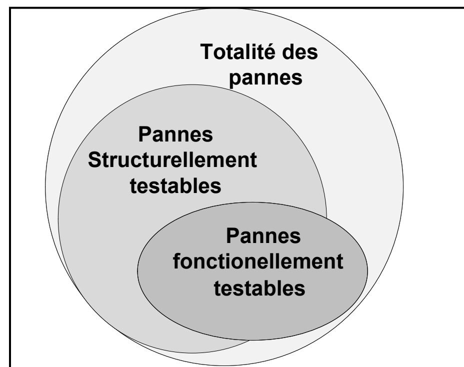


Figure 0.2 Classification des pannes basée sur leur détectabilité.
Adaptée de (Lin, 2006)

En plus du surtest, l'existence des états illégaux accentue la complexité de la génération de tests. En effet, la génération de vecteurs de test consiste essentiellement en 2 étapes : activation de la panne et propagation de cette dernière vers une sortie observable. Les deux étapes requièrent une analyse et une justification des valeurs à travers le circuit. En présence des états illégaux, l'espace de recherche devient plus grand et le générateur de test pourrait tenter de justifier des états qui sont injustifiables. Ceci dégrade l'efficacité de la génération de test. Plusieurs contributions sont faites dans le domaine mais la problématique est toujours là pour les circuits séquentiels complexes (Konijnenburg, 1999). L'identification des états illégaux aide à contraindre l'espace de recherche du STPG aux états légaux et justifiables afin d'éviter les recherches inutiles, sauver du temps CPU et augmenter l'efficacité du test (Konijnenburg, 1999).

En conclusion, l'existence potentielle des états illégaux dans les vecteurs de test générés en conjonction avec la technique DFT d'insertion de registres à balayage crée des problèmes au niveau de la vérification et du test.

Objectifs de la recherche

Dans notre travail de recherche, nous visons à améliorer la qualité et la productivité de la vérification fonctionnelle des circuits séquentiels complexes, plus précisément au niveau

d'abstraction appelé transfert de registres (« Register Transfert, RT »), qui constitue le niveau dominant pour l'entrée du design (via l'utilisation de langages de description matérielle tel le langage de description matériel des circuits intégrés à grande vitesse (VHDL, *Very high speed integrated circuit Hardware Description Langage*) et qui se situe entre le niveau système et celui des portes logiques. Nous ciblons plus particulièrement les circuits séquentiels complexes dominés par le contrôle tels que les machines à états (FSM, *Finite State Machine*), sachant que la plupart de ces circuits peuvent être modélisés sous forme de machines à états finis (Goren, 2002). L'objectif principal des travaux de recherche est de développer un environnement de vérification permettant de surmonter les principaux défis de la vérification fonctionnelle et visant essentiellement à :

- éliminer la phase de lecture et de compréhension du modèle précédant la vérification;
- automatiser la conception des bancs d'essai basés sur des stimuli efficaces et puissants;
- augmenter la contrôlabilité du DUV lors de la simulation;
- réduire le temps de la simulation des circuits séquentiels;
- augmenter l'observabilité pour une meilleure détection d'erreurs;
- améliorer la couverture de la vérification, et permettre la couverture des nombreux états d'un design sans faire appel aux longues séquences de stimuli;
- réduire le temps et l'effort investis dans la vérification en automatisant le processus au complet.

Une nouvelle méthodologie de vérification est proposée. Cette méthodologie explore une combinaison entre le test et la vérification. L'idée est de tirer profit du développement et de l'évolution des techniques et outils de test (DFT, ATPG) pour améliorer la vérification. Cet objectif global se divise en plusieurs objectifs spécifiques :

- définir la relation entre ce qui est difficile à tester et ce qui est difficile à vérifier;
- développer une stratégie pour le passage du niveau porte logique structurel au niveau RT structurel ou comportemental;
- choisir un modèle de panne adéquat pour une simulation efficace de la fonctionnalité;
- éliminer les états illégaux des vecteurs de test structurels;
- émuler les techniques de conception en vue du test au niveau de la simulation RT;

- automatiser la conception des bancs d'essai au niveau RT (RTL, « Register Transfer Level») en exploitant les vecteurs de test structurels.

On suppose que les circuits sont au niveau RT (VHDL) et que leur modèle de référence (SystemC) est disponible (Séméria, 2002; Wagner I., 2007). Tous les outils présentés dans la thèse peuvent être adaptés pour couvrir d'autres langages de description matérielle (ex. Verilog).

Contributions de la thèse

Les contributions majeures de cette thèse sont les suivantes :

- dans (Hobeika, 2008), nous avons montré que ce qui est difficile à tester est difficile à vérifier mais ce qui est difficile à vérifier n'est pas nécessairement difficile à tester. Cette corrélation a permis d'orienter notre recherche vers l'utilisation des vecteurs de test dans la vérification afin d'améliorer la qualité de la vérification fonctionnelle. Une justification du choix du type de vecteurs de test structurel (vecteurs de transition LOC *Launch On Capture*) ainsi qu'une description détaillée de la stratégie de passage du niveau logique au niveau RT sont aussi présentées;
- deux articles ont été publiés (Hobeika 2009; 2010) et un troisième de journal (Hobeika, 2011) (sujet du chapitre 5), présentant l'environnement de vérification automatique basé sur l'utilisation des tests structurels. De plus, nous avons reçu le prix de la meilleure affiche étudiante lors de l'IEEE VLSI Test Symposium qui a eu lieu en 2009 en Californie, toujours sur le même sujet. L'environnement de vérification proposé est composé de trois outils de base:
    - un extracteur automatique identifiant les contraintes fonctionnelles de conception. Dans (Hobeika, 2010), nous avons développé une nouvelle approche pour l'extraction des contraintes fonctionnelles au niveau RT. La publication en cours de rédaction présente l'outil avec les algorithmes et les implémentations détaillées. Dans nos travaux, nous avons montré l'efficacité des contraintes extraites :
        - au niveau de la vérification : Avec l'application des contraintes fonctionnelles aucune fausse erreur n'a été détectée;

    - au niveau du test : Les contraintes fonctionnelles extraites ont permis une diminution du surtest.
- un générateur automatique de bancs d'essai. Sur la base de la corrélation présentée dans (Hobeika, 2008), nous avons proposé dans (Hobeika, 2009) un générateur automatique de bancs d'essai permettant d'appliquer les vecteurs de test structurel lors de la simulation du DUV. L'outil permet d'adapter les vecteurs de test structurel au niveau RT, et d'émuler la technique DFT d'insertion de registres à balayage durant la vérification ;
- un détecteur automatique d'erreurs basé sur une observabilité élevée. Cet outil est présenté également dans (Hobeika, 2009). Il prend en entrée les réponses de la simulation du DUV et celles du modèle de référence, et produit en sortie une liste des erreurs détectées.

Organisation de la thèse

La thèse est structurée comme suit. Le chapitre 1 présente les notions de bases des domaines du test et de vérification. Une revue de littérature englobant ces deux domaines est présentée dans le chapitre 2. D'un côté, nous présentons les algorithmes et techniques de tests avancées qui seront utilisés dans nos travaux, et d'un autre côté nous décrivons les différentes techniques et méthodologies les plus utilisées de nos jours dans le domaine de la vérification. Afin de justifier l'orientation de nos recherches, le chapitre 3 établit la relation existante entre la difficulté à tester et la difficulté à vérifier et présente la stratégie requise pour passer du niveau porte logique structurel au niveau RT comportemental ou structurel. Ensuite, une étude théorique montrant la capacité des tests structurels de transition LOC à détecter les erreurs RTL est présentée dans le chapitre 4. Le chapitre 5 décrit en détails la méthodologie proposée pour l'extraction des contraintes fonctionnelles d'un design afin d'éviter la production des états illégaux. Et finalement le chapitre 6 présente l'environnement de vérification complet incluant les 3 outils automatiques ainsi qu'une comparaison des résultats obtenus avec les techniques de vérification existantes afin d'évaluer la méthodologie proposée. La thèse se termine par les conclusions et recommandations.

CHAPITRE 1

TEST ET VÉRIFICATION

1.1 Introduction

Les travaux de recherche effectués dans le cadre de cette thèse englobent les domaines de la vérification fonctionnelle et du test. Une présentation de ces deux domaines, de leurs concepts et limitations s'avère essentielle pour introduire, par la suite, la méthodologie de recherche.

Ce chapitre est divisé en deux grandes parties. La première introduit le domaine de la vérification en présentant ses deux faces, statique et dynamique, et montrant les caractéristiques de chacune ainsi que les différents types de méthodologies impliquées. La deuxième partie présente le domaine du test, plus précisément les tests manufacturiers ainsi que les techniques développées pour améliorer leur qualité.

1.2 Notions de base de la vérification

Le processus de conception consiste à transformer un ensemble de spécifications d'un design en une implémentation détaillée. Cette implémentation est raffinée en augmentant le niveau de détails lors du passage d'un niveau d'abstraction à l'autre. Les principaux niveaux d'abstraction des implémentations sont : les spécifications fonctionnelles, la description algorithmique, le RTL, les portes logiques (« *netlist* »), les transistors et les masques. La vérification est le processus inverse de la conception. Elle permet de s'assurer qu'une représentation du design à un niveau d'abstraction donné répond bien au niveau d'abstraction supérieur ou ultimement aux spécifications. On parle de vérification fonctionnelle lors qu'on s'intéresse plus particulièrement aux aspects de fonctionnalité d'un design (par opposition aux éléments dits non-fonctionnels telles la puissance dissipée, la fréquence maximale de l'horloge, etc.). À chaque étape de la conception, i.e pour chaque passage d'un niveau

d'abstraction à l'autre, correspond une étape de vérification. La vérification accompagne donc chaque raffinement durant la conception du design, chaque changement dans l'algorithme, chaque ajout de détail. Les deux processus de conception et de vérification sont décrits dans la Figure 1.1 (Lam, 2005).

Dans ce qui suit, nous présentons en quoi consiste une méthodologie de vérification et nous décrivons ses deux faces, statique et dynamique.

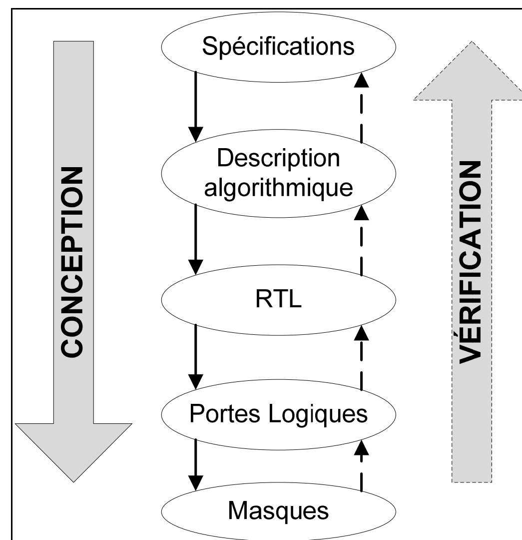


Figure 1.1 Flux de conception et de vérification.
Adaptée de (Lam, 2005)

1.2.1 Méthodologie de vérification

Une méthodologie de vérification débute par un plan de vérification mettant en relief l'ensemble des fonctionnalités à vérifier dans le but de satisfaire les spécifications d'un système.

Pour surveiller l'avancement du plan de vérification, un historique des éléments vérifiés ainsi que les éléments qui restent à vérifier est conservé. La réalité et l'expérience démontrent que la vérification complète d'un ensemble de spécifications est très difficilement réalisable, voir même impossible. C'est pourquoi il existe une mesure de la qualité de la vérification, appelée couverture, qui consiste en un estimé du pourcentage de spécifications vérifiées. En général, la qualité d'une méthodologie de vérification dépend de la couverture atteinte et du temps requis pour atteindre cette couverture. Une bonne méthodologie de vérification cherche donc généralement à réduire le temps et maximiser la couverture de la vérification.

Les méthodologies de vérification peuvent être regroupées en deux catégories (Lam, 2005):

- **la vérification dynamique** basée sur la simulation;
- **la vérification statique** basée essentiellement sur des techniques et des modèles mathématiques pour vérifier le design.

1.2.2 Vérification dynamique

La vérification dynamique est l'approche la plus utilisée en vérification (Lam, 2005). Elle est basée sur la simulation du design. En effet, le DUV est soumis à un flux de vecteurs de vérification appelés stimuli, qui sont émis par un banc d'essai ou *testbench*, alors que la sortie du design est comparée à une sortie de référence afin de détecter les erreurs. Dans le cas où la sortie observée ne correspond pas à la sortie de référence, une erreur est détectée et le débogage est déclenché. La simulation prend fin lorsque la couverture atteinte est jugée suffisante. En utilisant une métrique de couverture, on peut voir les parties du design non simulées et créer des stimuli pour les couvrir. La Figure 1.2 décrit le processus de la vérification dynamique, le composant à l'intérieur du carré en pointillés étant optionnel. En effet, les sorties de référence peuvent être générées avant ou durant la simulation. Afin de les générer durant la simulation, le modèle de référence est simulé en parallèle avec le DUV.

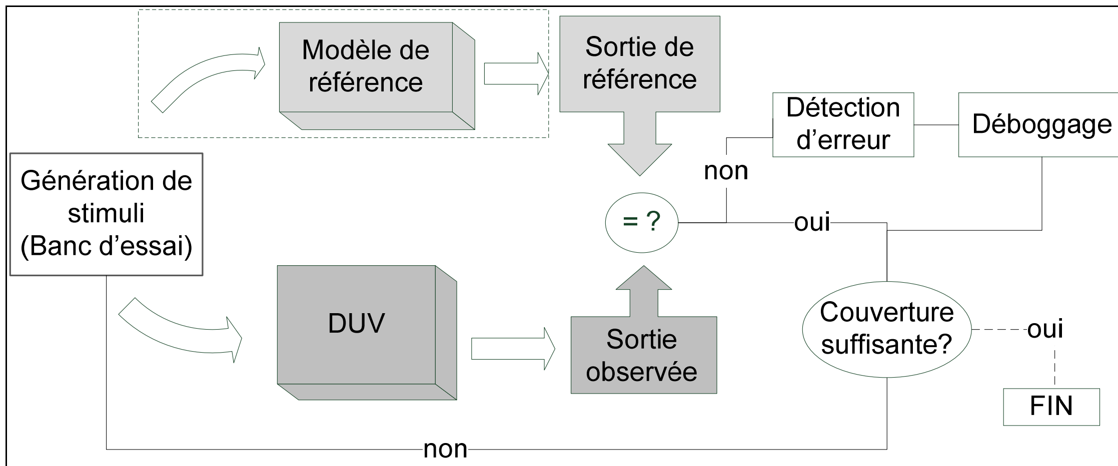


Figure 1.2 La vérification dynamique.
Tirée de (Lam, 2005)

Génération des stimuli

Les stimuli peuvent être générés préalablement à la simulation et lus à partir d'une base de données, ou encore être générés au cours de la simulation. La qualité de la simulation dépend de la couverture atteinte lors de leur application. Les techniques de génération de ces vecteurs de vérification diffèrent selon le type de ces derniers. On distingue :

- les vecteurs de vérification aléatoires: ces vecteurs de vérification sont générés à partir d'un algorithme de génération de valeurs aléatoires. Cet aspect aléatoire permet de couvrir les coins du design et les scénarios auxquels les concepteurs n'auraient peut-être pas pensé. Ces vecteurs permettent typiquement de détecter des erreurs dans ces espaces, ce qui élargit la couverture de la vérification. L'effort requis par cette méthode est minime dans la mesure où la génération aléatoire de vecteurs peut se faire automatiquement. La vérification basée sur ce type de stimuli est appelée vérification aléatoire (« *random verification* »);
- les vecteurs de vérification directs : ces vecteurs de vérification sont très spécifiques et visent des espaces déterminés du design, où les concepteurs sont conscients de la présence de certaines erreurs potentielles. Le concepteur établit une liste de scénarios spécifiques, où chaque scénario cible une erreur ou une partie de l'espace de design. Ceci demande un effort substantiel, le concepteur devant implémenter chacun des scénarios manuellement.

La vérification basée sur ce type de vecteurs est appelée vérification dirigée (« *directed verification »*);

- les vecteurs de vérification aléatoires basés sur les contraintes : ces vecteurs de vérification sont générés aléatoirement mais en respectant des contraintes spécifiques modélisant la fonctionnalité du DUV. Ce type de vérification conserve l'aspect aléatoire pour couvrir les détails manquants d'un scénario donné et sont plus productifs car ils sont dirigés par des contraintes représentant le design. Chaque slave de vérification décrit plusieurs scénarios. La vérification basée sur ce type de vecteurs est appelée vérification basée sur les contraintes (« *Constrained-based verification »*).

La génération de vecteurs de vérification reste une des étapes les plus exigeantes de la vérification, le but étant de produire avec un minimum d'effort un ensemble efficace de vecteurs permettant une bonne couverture et une simulation rapide.

Avantages et limitations de la vérification dynamique

La vérification dynamique est la plus utilisée grâce à ses nombreux points forts (Lam, 2005) :

- l'aspect « *input driven* ». La vérification dynamique est basée sur des vecteurs d'entrée. Dans un premier temps, on génère ces vecteurs et dans un second temps on génère les sorties de référence correspondantes. Ceci est un avantage par rapport à la vérification formelle, dans laquelle le processus est inversé : le concepteur pense aux sorties désirables et laisse le vérificateur le prouver, d'où sa qualification de « *output driven* ». Il est beaucoup plus facile de penser « *input driven »* que « *output driven »;*

- l'aspect pseudo-aléatoire. En effet, les bugs ou les erreurs ont lieu souvent dans des régions que le concepteur ignore, et ce genre de vérification permet d'explorer ces régions en question. Les simulations pseudo-aléatoires peuvent utiliser des vecteurs au voisinage des vecteurs directs. En effet, si les vecteurs de vérification directs sont des points dans l'espace, les vecteurs aléatoires couvrent les régions entourant ces points. Donc pour décentrer la vérification et explorer toutes les régions possibles du design, les vecteurs pseudo-aléatoires sont utilisés en conjonction avec les vecteurs directs. Cette

approche permet une amélioration considérable de la couverture et du temps de simulation;

- la possibilité de vérification au niveau système. L'un des grands défis de la vérification est d'être exploitée au niveau système, ceci à cause de la complexité croissante des designs. La simulation ou la vérification dynamique permet de vérifier le design au niveau système sans se soucier de la taille et de la complexité de ce dernier.

La vérification dynamique présente aussi plusieurs limitations. Les défis auxquels elle fait face sont de plus en plus nombreux avec la croissance de la complexité des circuits (Lam, 2005):

- le temps de conception des bancs d'essai est énorme;
- la qualité des bancs d'essai est souvent médiocre;
- la couverture atteinte est habituellement insuffisante lorsque le temps alloué pour la vérification est limité;
- l'ingénieur n'est jamais sûr si le design est bien vérifié ou non;
- il y a souvent échec dans la détection des erreurs difficiles à vérifier, appelées « *corner case errors* » ou aussi les coins sombres.

Plusieurs méthodologies ont été développées pour améliorer la qualité de la vérification dynamique et surpasser ses limitations. Certains travaux seront présentés dans le chapitre 2 de revue de littérature.

1.2.3 Vérification statique

La vérification statique (Lam, 2005) diffère de celle basée sur la simulation par le seul fait qu'elle ne requiert pas de stimuli. Elle est basée essentiellement sur des techniques et des modèles mathématiques pour vérifier le design. Elle peut être classifiée en 2 grandes catégories : la vérification de propriété (« *property checking* ») (Baumgartner, 2002) et la vérification d'équivalence (« *equivalence checking* ») (Huang, 1998).

La vérification de propriété

Un certain nombre de propriétés doivent être satisfaites lors du développement d'un design. Une approche classique de vérification, la vérification de propriété, consiste à : 1) construire un modèle complet *S* du comportement du système étudié (par exemple un graphe de flux de données, les différents modèles sont présentés en détails dans l'annexe I), 2) formaliser la propriété de correction attendue par une formule *Φ*, et 3) utiliser un algorithme de vérification de modèle (« *Model-Checking »)* (Saidi, 2000) pour vérifier que *S* satisfait ou non *Φ*. Une propriété est une description dupliquée du design et permet la vérification du design à travers la redondance. L'idée est d'explorer l'espace dans le but de trouver un point qui contredit la propriété. Si ce point existe alors la propriété échoue et ce point servira de contre-exemple, sinon la propriété est satisfaite.

La limitation majeure rencontrée dans ce genre de vérification est le temps mis pour déterminer les bonnes contraintes et pour débugger le système en cas d'échec de propriétés. Une alternative à l'exploration de l'espace existe et permet ainsi la vérification de propriétés. Il s'agit d'une approche basée sur les preuves de théorèmes. Dans une telle approche, la propriété est modélisée sous forme d'une proposition mathématique et le design sous forme d'entités mathématiques (i.e. un ensemble d'axiomes). L'objectif est de déterminer si les propositions (propriétés) peuvent être déduites des axiomes (système). Si c'est le cas, la propriété est vérifiée sinon elle échoue. Cette approche fait également face à plusieurs limitations, l'usager devant être familier avec les opérations internes des outils ainsi qu'avec les processus mathématiques formels de démonstration.

La vérification d'équivalence

Lors de la conception d'un système, le concepteur manipule plusieurs modèles dédiés pour modéliser le même système de base, chacun pouvant correspondre à un niveau d'abstraction différent. Donc, outre la vérification des propriétés et de la fonctionnalité, une autre forme de vérification est essentielle : la vérification de l'équivalence des modèles entre eux. Il s'agit de vérifier que les descriptions d'un système à deux niveaux d'abstraction différents ont une

fonctionnalité équivalente et correspondent donc au même système. Deux méthodologies sont utilisées pour accomplir ce genre de vérification.

- la première crée des représentations canoniques des deux systèmes et les compare. Une représentation canonique est telle que deux fonctions logiques sont équivalentes si et seulement si leurs représentations sont isomorphes. Une des représentations les plus utilisées est le diagramme de décision binaire *BDD (« Binary Decision Diagram »)* (Prasad P., 2004). Une description des modèles BDD est présentée dans l'annexe I;
- la deuxième cherche d'une façon systématique dans l'espace des entrées, un ou plusieurs vecteurs d'entrée permettant de distinguer les deux modèles, i.e. créer des comportements différents. Cette approche est appelée l'approche SAT (« *satisfiability* ») (Parthasarathy, 2004).

Si la vérification d'équivalence échoue, l'outil génère une séquence de vecteurs de vérification qui, une fois simulés, montre la différence entre les deux modèles. Cette séquence représente un contre-exemple pour la démonstration de l'équivalence des deux circuits.

Avantages et limitations de la vérification statique

Les techniques de vérification statiques offrent des avantages non négligeables (Lam, 2005) :

- bonne performance dans la détection des coins sombres, les « *corner case errors* ». En effet, la vérification statique ne peut négliger aucun point dans l'espace d'entrée du système, un problème auquel la vérification dynamique fait habituellement face. Pour une propriété donnée, la vérification statique explore tous les points de l'espace d'entrée et toutes les transitions possibles pour aboutir à une couverture complète de cette propriété;
- possibilité de vérification simultanée de plusieurs points de l'espace. La vérification statique opère par propriété et non par point de l'espace, de ce fait la vérification d'une propriété implique la vérification de plusieurs points de l'espace simultanément;

- possibilité de vérification de l'équivalence de deux systèmes (« *Equivalence Checking* ») à travers l'équivalence de leurs modèles formels respectifs. Avec la vérification par simulation, c'est quasiment impossible de vérifier l'équivalence de deux systèmes car ceci requiert un nombre quasi infini de vecteurs en entrée à vérifier.

La vérification statique offre une garantie dans la vérification des propriétés, toutefois plusieurs facteurs remettent en question cette forme de vérification (Lam, 2005):

- la limitation dans l'espace et dans le temps empêche la vérification complète d'un système. On vérifie seulement une partie du système, ce qui fait qu'un grand nombre d'erreurs peuvent rester non couverts;
- la complexité liée à la recherche de l'erreur qui a causé l'échec de la propriété rend cette approche très consommatrice en temps et en effort;
- la vérification au niveau système est quasiment impossible dans ce genre de vérification surtout pour les conceptions complexes qui présentent des modèles de graphes de grande envergure.

La vérification statique est une manière systématique pour traverser l'espace d'états dans le but de réaliser une vérification complète vis-à-vis la propriété à vérifier. Donc la vérification statique ne peut pas prouver que toutes les propriétés dans un design ont été énumérées mais peut garantir pour une propriété donnée si elle est satisfaite ou non.

1.3 Notions de base du test

Il est important de lever l'ambigüité entre le test et la vérification, en expliquant la différence essentielle entre ces deux domaines. Tel que décrit dans la section précédente, la vérification est basée sur l'analyse comportementale d'un design et est effectuée avant la fabrication du circuit. Elle cherche à prouver la bonne fonctionnalité d'un modèle donné et l'équivalence de deux modèles lors du passage d'un niveau d'abstraction à un autre. Elle permet de s'assurer de la cohérence entre deux niveaux d'abstraction, et de vérifier que la description matérielle/logicielle exécute bien la fonctionnalité choisie et respecte les spécifications. Alors que le test, lui, survient après la fabrication des circuits. Il permet de s'assurer que le

matériel implanté réponde aux spécifications. Un design correct ne garantit pas qu'un composant manufacturé soit opérationnel, en raison des défectuosités inhérentes au procédé de fabrication. Comme le coût du test d'une composante défectueuse est proportionnel au temps pris pour en faire la détection, l'objectif est de couvrir le plus de défectuosités possibles, et ce, en un minimum de temps. Le processus de test a dû, comme le reste, s'adapter à l'augmentation de la complexité. La recherche et le développement de nouvelles techniques tels les approches structurées de DFT, et les outils de génération automatisée de vecteurs de test les ATPG ont permis une grande évolution dans le domaine. Les tests peuvent être classés en différentes catégories :

- les tests fonctionnels (comportementaux) : ces tests ne sont pas structurés et ne visent pas de pannes en particulier mais émulent le circuit intégré en le considérant comme une boîte noire;
- les tests structuraux : ces tests sont générés en ciblant des pannes affectant la structure du circuit sous test. Les principaux modèles de pannes sont présentés plus loin. Dans le cadre de cette thèse, nous nous intéressons plus particulièrement à ce type de test.

1.3.1 Modèles de panne

Un modèle de panne est la représentation structurelle d'une défectuosité physique. Il s'agit d'une abstraction et d'une simplification de la réalité. Plusieurs types de défectuosités peuvent être représentés avec un nombre restreint de modèles qui sont habituellement indépendants de la technologie et requièrent peu d'informations. La génération des vecteurs de test structuraux se base sur des modèles de panne bien spécifiques. L'objectif est de choisir un modèle de panne qui représente le plus grands nombre de pannes physiques dans le circuit afin d'atteindre une bonne couverture.

Les modèles de pannes visés par les tests manufacturiers sont :

- modèle collé (« *stuck-at* »): selon ce modèle les nœuds affectés sont en permanence collés à 0 (la masse) ou à 1 (Vcc);

- modèle collé-ouvert (« *stuck-open* ») : ce modèle représente une connexion brisée du circuit résultant en une déconnexion de certains nœuds, créant des nœuds flottants qui ne sont reliés ni à la masse ni au Vcc;
- modèle court-circuit (« *bridging faults* ») : ce modèle représente les pannes qui résultent d'une connexion indésirable entre deux nœuds différents;
- pannes de délai : ces pannes ont pour effet de ralentir les transitions qui sont propagées dans le circuit. Elles n'altèrent pas la valeur logique finale, seulement le temps pris pour que cette valeur soit atteinte.

Dans le cadre de cette thèse, nous nous intéressons plus particulièrement aux pannes de délai. Tester un circuit logique pour les pannes de délai correspond à s'assurer que celui-ci respecte les spécifications temporelles et qu'il puisse opérer à la période d'horloge désirée. Il existe deux modèles principaux de pannes de délai : 1) les pannes distribuées sur les chemins combinatoires (« *path-delay fault* ») et 2) les pannes ponctuelles dites transitionnelles (« *transient-delay fault* »). Le premier type distribue le délai supplémentaire sur la totalité du chemin ciblé et sert surtout pour la modélisation des chemins critiques affectés par les variations du procédé de fabrication. Le second type, qui est le plus utilisé pour la détection des pannes, suppose qu'une panne de délai affecte un point particulier du réseau combinatoire (ex. la sortie d'une porte) et que le délai supplémentaire est suffisamment important pour causer une violation des spécifications temporelles. Si toutes les pannes sont locales, alors le modèle de pannes transitionnelles est le modèle adéquat à utiliser durant le test. Par contre, si les pannes sont plus globales, il est alors nécessaire d'adopter le modèle de pannes de délai distribuées sur les chemins. Notons cependant que le modèle de pannes transitionnelles est le plus utilisé, car le modèle de pannes de délai distribuées se heurte à l'explosion combinatoire de chemins à tester. Dans cette thèse, nous nous intéressons plus particulièrement au second type de panne, qui est le plus adapté à nos besoins.

1.3.2 Génération automatique de vecteurs de test

Afin d'améliorer la qualité et la productivité du test et diminuer sa complexité, différentes techniques et méthodologies ont été développées pour automatiser la génération des vecteurs de test. L'objectif des outils ATPG est de générer, basé sur un modèle de pannes déterminé, un ensemble minimal de vecteurs de test efficace permettant une très bonne couverture. Les classes majeures des méthodes utilisées par les outils ATPG sont :

- pseudo-aléatoire : cette approche utilise un ensemble de vecteurs de test pseudo-aléatoires, et effectue une simulation de pannes pour déterminer l'ensemble des pannes potentiellement détectables. Les caractéristiques des pannes influence la qualité de cette approche;
- basés sur des algorithmes : ces approches se basent sur une étude de la structure logique du circuit pour générer le vecteur de test requis pour activer et propager une panne sur un nœud bien déterminé.

Toutes ces techniques ont permis le développement de plusieurs outils ATPG très puissants, efficaces et rapides.

1.3.3 Conception en vue du test

Tester un composant après sa fabrication est un processus complexe, notamment car le concepteur n'a accès qu'aux entrées et sorties principales du circuit. Si on considère un circuit complexe avec une centaine de millions de transistors et un nombre quasi-infini d'états, il peut s'avérer difficile d'emmener ce composant dans un état particulier pour observer ses sorties. Il s'avère donc avantageux de considérer le test à partir des phases primaires de la conception afin de faciliter et simplifier la tâche : on parle alors de conception en vue du test ou DFT (Rabaey, 1996).

Deux propriétés, la contrôlabilité et l'observabilité, sont de grande importance lorsqu'on considère la testabilité d'une conception. Elles affectent la qualité des tests manufacturiers.

La contrôlabilité mesure la facilité avec laquelle on peut contrôler un nœud donné à partir de l'entrée, tandis que l'observabilité est la facilité avec laquelle on peut observer la valeur d'un nœud aux broches de sortie d'un circuit intégré. Donc, plus le circuit est observable et contrôlable, plus il est facile à tester et plus la couverture du test est élevée.

Vu la complexité de la génération des vecteurs de test et de production d'une couverture de test, l'idée est de considérer le test bien avant le stade final, en influençant le design tout au long du processus de conception afin d'augmenter la contrôlabilité et l'observabilité de ces circuits et d'améliorer au final la qualité de test. Les circuits purement combinatoires sont généralement facilement observables et contrôlables, alors que les circuits séquentiels peuvent présenter des lacunes a ce niveau. Les techniques DFT, qui ont été développées pour les circuits séquentiels, se classent en 3 catégories (Rabaey, 1996) :

- les méthodes ad-hoc : elles sont basées sur le principe «diviser pour conquérir ». Il s'agit de partitionner les grands circuits séquentiels et d'ajouter des points de test ainsi que des multiplexeurs ou d'autres matériels supplémentaires à l'intérieur du circuit afin d'améliorer la qualité du test;
- méthodes d'auto test (*BIST, « Built In Self Test »*): Le circuit se teste essentiellement tout seul, en utilisant des générateurs de stimuli pseudos aléatoires ou exhaustifs et vérifiant l'exactitude des résultats (avec un support minimal d'un testeur externe);
- méthodes basées sur l'insertion des chaînes de registres à balayage : lors de la conception, des éléments de mémoire synchrones appelés registres à balayage (« *scan registers »)* connectés en chaîne sont ajoutés au circuit pour augmenter l'observabilité et la contrôlabilité de ce dernier. Une approche très utilisée est le test à balayage en série illustré dans la Figure 1.3 (Rabaey, 1996). Les registres sont modifiés pour supporter deux modes : un mode normal où ils opèrent comme des registres synchrones normaux et un mode de test où ils sont enchainés ensemble pour former un seul (ou plusieurs) registre(s) de décalage en série appelé chaine de registres à balayage ou chaine de scan. Une procédure de test s'effectue alors de la manière suivante :

- un vecteur de test est appliqué à travers le port ScanIn et est décalé dans les registres à l'aide d'une horloge de test;
- le vecteur est appliqué sur la logique et se propage à la sortie du module;
- le résultat est décalé à l'extérieur du circuit à travers le port ScanOut et est comparé aux résultats désirés.

Ainsi le chemin séquentiel est transformé en un chemin combinatoire, réduisant la longueur des séquences de test requises pour forcer le design dans un état donné.

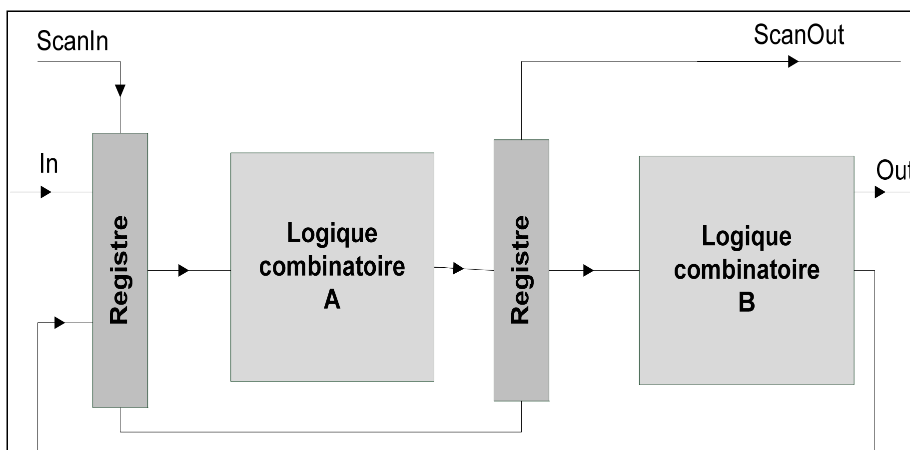


Figure 1.3 Test à balayage en série.
Tirée de (Rabaey, 1996)

1.3.4 Tests de transition avec insertion des registres à balayage

Le test de transition permet de détecter les pannes de délai transitionnelles liées à une entrée ou une sortie d'une porte logique. L'activation d'une panne de délai nécessite d'appliquer une transition sur le site de la panne, de la propager vers un nœud observable et de vérifier si les spécifications temporelles sont respectées. Cette transition est obtenue en appliquant successivement deux valeurs complémentaires sur le site. Elle est propagée suite à l'ouverture d'un chemin entre le site de la panne et un point observable du circuit. L'observation de la panne consiste à capturer les valeurs de sortie et à les comparer aux valeurs de référence pour détecter les délais. Et donc, le test de délai de transition suppose

l'application de deux vecteurs de test (V_1, V_2) tel que: V_1 est un vecteur d'initialisation qui établit les conditions initiales, et V_2 un vecteur de test correspondant au vecteur de test « *stuck-at* », qui lance la transition à un ou plusieurs nœuds cibles et établit les conditions nécessaires à leur propagation vers une sortie primaire ou un nœud observable (Waicukauski, 1987). Sur chaque nœud du circuit, deux transitions sont possibles : montantes (STR, « *slow-to-rise* ») ou descendante (STF, « *slow-to-fall* »).

Les tests de transition avec insertion des registres à balayage peuvent être appliqués de 3 façons :

- *« launch On Capture »* ou *« Broad-side »* (Savir, 1994) : l'ensemble des vecteurs (V_1, V_2) est tel que le second vecteur de test V_2 est la réponse du circuit au premier vecteur de test V_1 qui réside dans les chaines de scan. Un seul vecteur est stocké dans la mémoire de scan du testeur. Ce type de test requiert deux cycles d'horloge, un pour chacun des vecteurs;
- *« skewed-load »* (Savir, 1993) : L'ensemble des vecteurs (V_1, V_2) est tel que le second vecteur de test V_2 est le résultat du décalage de 1 bit du premier vecteur de test V_1 qui réside dans les chaines de scan un vecteur de N bits est chargé en décalant les $N-1$ premiers bits, avec N la longueur de la chaine de scan. Le dernier décalage est utilisé pour lancer la transition. Dans ce cas aussi, un seul vecteur est stocké dans la mémoire de scan du testeur. Ce type de test requiert un seul cycle d'horloge mais présente un problème avec les dépendances de décalage;
- *« enhanced-scan »* (Cheng, 1991): deux vecteurs (V_1, V_2) sont stockés. Le vecteur V_1 est chargé en premier pour initialiser le circuit et le vecteur V_2 permet la création de la transition et sa propagation.

Dans le cadre de cette thèse, nous utilisons le LOC.

1.4 Conclusion

Nous avons présenté dans ce chapitre, deux phases importantes du processus de mise en œuvre d'un circuit intégré, à savoir la vérification et le test. Pour chacune de ces deux phases, nous avons décrit les différentes catégories, les caractéristiques correspondantes ainsi que les avantages et limitations qui y sont associés. Cette présentation s'avère indispensable pour notre recherche qui se base essentiellement sur ces deux phases ou domaines.

Nous avons vu à travers ce chapitre les nombreuses limitations de la vérification par rapport au test où plusieurs techniques et méthodologies ont permis le développement d'outils automatiques de test conduisant à une meilleure qualité et efficacité de ce dernier. Basé sur cette observation, l'idée de tirer profit de ces avancements au niveau du test et de les utiliser pour surpasser les limitations au niveau de la vérification s'est développée pour aboutir à la méthodologie de vérification présentée dans cette thèse.

CHAPITRE 2

REVUE DE LITTÉRATURE

Dans le but de situer les travaux de recherche effectués dans le cadre de cette thèse, ce chapitre présente dans un premier temps une revue de littérature couvrant le domaine de la vérification dynamique. Les différentes techniques utilisées dans l'industrie pour la vérification des designs seront décrites en détails. En outre, une revue des travaux de recherche combinant les deux domaines du test et de la vérification est présentée. De plus, étant donné que cette recherche vise aussi à remédier le problème des états illégaux, les différentes méthodologies d'extraction des contraintes fonctionnelles d'un design et des états illégaux sont exposées.

2.1 Vérification dynamique

Les pratiques industrielles de nos jours se basent essentiellement sur la vérification dynamique pour effectuer une vérification fonctionnelle des designs. Dans cette section, nous présentons les méthodologies les plus utilisées dans l'industrie pour ce type de vérification.

2.1.1 Vérification aléatoire basée sur les contraintes

Les deux approches traditionnelles initialement utilisées dans la vérification dynamique sont la vérification dirigée et la vérification aléatoire. Tel que mentionné précédemment, la première se base sur des vecteurs dirigés implémentés manuellement par les ingénieurs de vérification et visant des scénarios bien spécifiques, et la deuxième se base sur la génération automatique de vecteurs aléatoires cherchant à couvrir les erreurs insaisissables par des vecteurs dirigés. Le grand nombre de stimuli requis pour couvrir tous les scénarios possibles combinés à la nature complexe des erreurs, qui se trouvent généralement dans des états demandant une longue séquence de vecteurs, rendent ces deux approches traditionnelles inefficaces et exigeantes en termes de temps et d'efforts. Ainsi, l'approche aléatoire basée sur des contraintes (Yuan, 2006) a été développée pour remédier à ces limitations et combiner les

avantages des deux approches traditionnelles. Tel qu'indiqué au chapitre précédent, la génération de stimuli se fait d'une façon aléatoire mais dirigée par un ensemble de contraintes permettant de respecter certaines règles afin de produire des vecteurs de vérification plus productifs et efficaces. L'ensemble des contraintes est défini par les ingénieurs de vérification pour modéliser (Naveh, 2007) :

- les détails de l'architecture matérielle du modèle à vérifier permettant de générer des vecteurs de vérification valides. En effet, l'architecture impose des règles à respecter pour avoir un programme de vérification valide, comme les différentes combinaisons possibles des valeurs d'entrées;
- la requête de l'ingénieur de vérification décrivant l'ensemble des scénarios de simulation. Ceci agit comme modèle pour le générateur à partir duquel il génère des vecteurs de vérifications différents en couvrant les détails manquants par des valeurs aléatoires;
- les recommandations des experts pour mieux diriger les vecteurs de vérification afin de couvrir le plus des erreurs possibles. Leurs connaissances sur l'emplacement probable des erreurs pour certaines architectures peuvent être utilisées pour améliorer la qualité des stimuli générés.

Les contraintes modélisant l'ensemble de ces règles définissent essentiellement comment combiner l'ensemble des valeurs d'entrées et à quel moment les appliquer lors de la simulation du design. Plusieurs techniques peuvent être utilisées pour définir une contrainte ou un ensemble de contraintes. Elles peuvent être divisées en deux catégories majeures (Yuan, 2006):

- une expression booléenne : Une contrainte peut être définie à l'aide d'une formule booléenne définie sur les différents signaux du design;
- une logique de modélisation auxiliaire :
  - des règles de grammaire comme celles utilisées pour les analyseurs syntaxiques offerts dans ce cas à travers le langage « *SystemVerilog »;*
  - des machines à états modélisant des comportements séquentiels complexes;

- des modèles provenant de la librairie de vérification (OVL*, «Open Verification Library »)* de moniteurs ou d'assertions qui peuvent être réutilisés dans ce cas pour définir une propriété ou une contrainte.

L'ensemble des contraintes définies est envoyé vers le générateur de stimuli pseudo-aléatoire. Ce dernier est constitué essentiellement d'un solveur de contraintes permettant de générer des vecteurs de simulation aléatoire tout en respectant les contraintes. Ces vecteurs sont appliqués aux ports du design sous simulation, et dans certains cas, les résultats de simulation sont renvoyés vers le modèle de contraintes afin de le mettre à jour. La Figure 2.1 représente l'environnement complet de vérification aléatoire basée sur les contraintes.

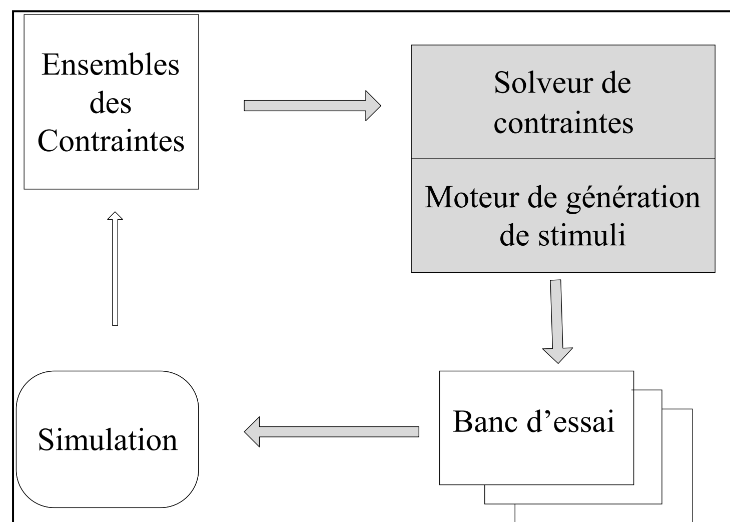


Figure 2.1 Environnement de vérification aléatoire basé sur les contraintes.
Adaptée de Naveh et al. (2007, p. 1722)

La complexité de ce type de vérification s'accroit avec celle du design. Afin d'atteindre de bons résultats, les ingénieurs de vérification investissent beaucoup d'efforts pour acquérir les connaissances et expertises spécifiques pour ce type de vérification. De plus, ils doivent bien comprendre le code et la fonctionnalité spécifique au design afin de définir une liste complète des contraintes modélisant le mieux possible le design et permettant une bonne simulation de sa fonctionnalité.

2.1.2 Vérification basée sur les assertions

Une autre technique permettant de faire face à la complexité de la vérification est la vérification basée sur les assertions. Une assertion est une description précise d'un comportement désiré ou non désiré. Dans cette méthodologie, la vérification est prise en considération dès l'étape de conception en insérant plusieurs assertions dans le design.

L'insertion des assertions améliore la qualité de la vérification en agissant à plusieurs niveaux. Les assertions permettent :

- une meilleure observabilité de différents coins du design. Ceci est avantageux surtout dans le cas des designs complexes;
- une meilleure détection de la source de l'erreur et donc une diminution du temps et des efforts associés à l'étape de débogage;
- une meilleure couverture de la structure interne. Les assertions non activées représentent les coins du design non couverts par le banc d'essai, ainsi des vecteurs dirigés peuvent être implémentés pour couvrir ces derniers.

Les assertions sont définies à l'aide d'un langage d'assertion, tel que le langage de spécification par propriétés (PSL, *«Property Specification Langage »*) (Accellera, 2004) ou les assertions du SystemVerilog (SVA, *« SystemVerilog Assertions »*) (Society, 2005) qui sont des langages très puissants permettant de :

- définir les assertions sous forme :
  - d'expressions booléennes pour modéliser la relation entre différents signaux;
  - d'expressions séquentielles pour modéliser une séquence d'évènements et donc la relation entre les expressions booléennes à travers le temps.
- définir des propriétés pour modéliser le comportement de différentes interfaces et blocs du design ainsi que la relation temporelle entre les séquences. La vérification de la propriété se fait à deux niveaux :
  - si elle est satisfaite ou pas;
  - si elle a été couverte au moins une fois;

Il existe aussi des librairies contenant des fonctions d'assertions déjà vérifiées et prêtes à être utilisées telles l'OVL (Foster, 2006) d'Accellera qui procure des fonctions d'assertion sous forme de librairies VHDL et Verilog, et la librairie de vérification Questa (QVL*, « Questa Verification Libratry »*) de Mentor qui inclut une librairie d'assertion SystemVerilog contenant des contrôleurs et moniteurs d'assertions.

La vérification basée sur les assertions est une première technique de conception en vue de la vérification. Elle présente plusieurs avantages au niveau de l'amélioration de l'observabilité, du débogage et de la couverture. Cependant, elle requiert beaucoup d'efforts en raison de la complexité liée à l'implémentation des assertions et à l'analyse de toutes les données de couverture générées suite à une simulation. Il faut bien comprendre le comportement du design, apprendre le langage d'assertion et surtout choisir les meilleurs emplacements dans le design en identifiant les coins difficiles à vérifier et en s'assurant que les situations illégales sont bien gérées. L'efficacité de cette méthode est donc grandement dépendante des compétences de l'ingénieur de vérification.

2.1.3 Vérification transactionnelle

Une des tâches les plus longues et ardues de la vérification réside dans la génération des bancs d'essai. La vérification transactionnelle (Brahme, 2000) cherche à simplifier la production du banc d'essai d'une part et permettre sa réutilisation dans la vérification des modèles à différents niveaux d'abstraction d'une autre part (Boland, 2007), et ceci en se basant sur le principe de transactions.

Une transaction est un transfert de données ou de contrôle entre le banc d'essai et le DUV à travers une interface. Elle modélise les données utiles pour un scénario de vérification, sans se soucier des différents signaux et des séries de bits nécessaires pour lancer ces stimuli au niveau d'abstraction du DUV. Ainsi, ce principe permet au concepteur de modéliser les bancs d'essai à un niveau d'abstraction plus haut d'une façon plus intuitive, et un adaptateur se charge de traduire les transactions de haut niveau en une série de signaux envoyés vers le

DUV. Le principe est décrit à la Figure 2.2 (Brahme, 2000). La figure montre la séparation du banc d'essai (« Testbench ») en deux couches. La couche supérieure est formée par l'ensemble des scénarios qui génèrent des transactions au niveau système sans tenir compte des détails des protocoles de communication au niveau signal, et la couche inferieure est l'adaptateur ou aussi appelé le modèle de vérification transactionnelle (TVM*, « Transaction Verification Model »*). Cette séparation des responsabilités en deux couches permet d'une part le développement de plusieurs scénarios complexes de vérification et d'une autre part la réutilisation de certaines composantes du banc d'essai.

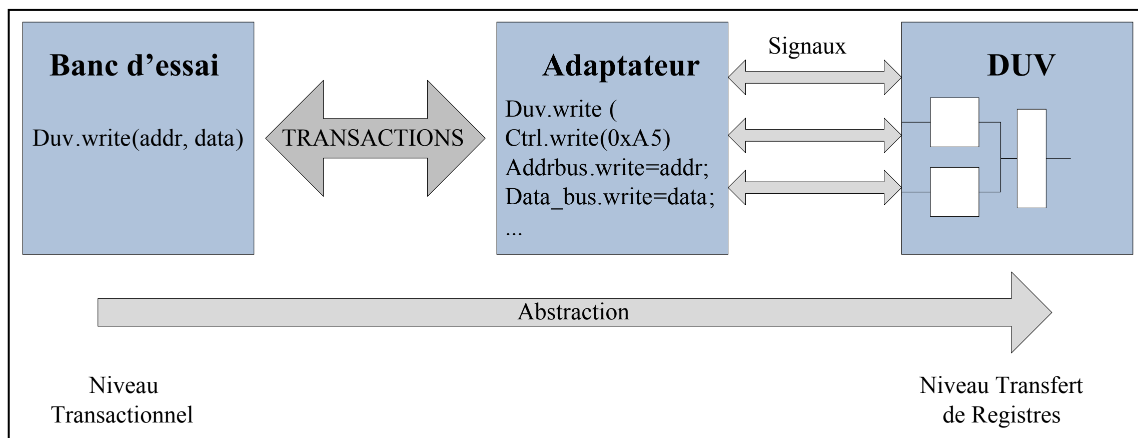


Figure 2.2 Vérification transactionnelle.
Adaptée de (Cadence, 2000)

De même que les méthodologies définies dans cette section, ce type de vérification requiert une bonne connaissance du comportement et de la fonctionnalité du design afin de définir la liste de scénarios de vérification possibles. De plus une étape d'apprentissage et de familiarisation avec les langages dédiés à ce type de vérification est nécessaire afin d'implémenter les adaptateurs requis et les transactions modélisant les différents scénarios.

2.2 Combinaison du test et de la vérification

Les phases de test et de vérification sont généralement considérées comme deux phases indépendantes. Mais les problèmes auxquels font face ces deux phases présentent plusieurs

similarités ce qui a conduit au développement de certaines méthodologies combinant les deux.

Le test fonctionnel (Rabaey, 1996), par exemple, utilise des vecteurs de vérification appelés aussi des vecteurs fonctionnels pour tester le design afin de couvrir les pannes n'appartenant à aucun modèle. Le design est simulé comme une boite noire, les tests fonctionnels sont forcés sur les entrées et les sorties sont observées pour la détection d'erreur.
De plus, les approches ATPG ont été adaptées et utilisées au niveau de la vérification statique pour remédier aux problèmes rencontrés lors de la vérification d'équivalence et de propriété (Varma, 2003) telles l'explosion de l'espace d'états des designs et la croissance de la complexité des algorithmes. En effet, certaines méthodologies de vérification d'équivalence utilisent les techniques ATPG combinatoires pour vérifier l'équivalence de deux représentations données, alors que d'autres méthodologies de vérification de propriété utilisent les techniques séquentielles pour vérifier si une propriété est satisfaite sur un nombre limité de cycles pour une implémentation donnée.

Dans (Varma, 2003), l'auteur souligne le besoin d'investiguer l'exploitation du test dans la vérification dynamique, qui est la technique la plus utilisée de nos jours, tout en soulevant certains enjeux :

- les tests manufacturiers visent le niveau porte logique alors que la vérification se fait généralement à des niveaux d'abstraction plus haut;
- il existe certains vecteurs de tests manufacturiers invalides ne pouvant pas simuler la fonctionnalité du design.

Dans (Abadir, 1988) les auteurs montrent qu'un ensemble de test « *stuck-at* » est capable de détecter des erreurs fonctionnelles. Ils utilisent les vecteurs de test « *stuck-at* » pour vérifier qu'une implémentation d'un design au niveau logique est structurellement équivalente aux spécifications, mais la simulation est effectuée au niveau porte logique seulement et les erreurs détectés sont de type structurels tels l'ajout, la suppression ou la substitution de portes logiques et de bus etc.

Dans cette thèse, nous présenterons un autre type de combinaison entre le test et la vérification, précisément l'utilisation des tests structurels dans la vérification dynamique RTL dans le but de détecter des erreurs de conception RTL.

2.3 Les modèles d'erreurs RTL

L'un des plus grands défis de la vérification réside dans la quantification et l'évaluation des résultats de la simulation. La plupart des méthodologies utilisent des métriques, telles que le nombre de lignes de code et le nombre d'états de transition, pour quantifier la couverture. Ces métriques reflètent la qualité de la simulation de la machine à états ou des différents lignes de code, mais leurs relation avec la détection des erreurs de conception n'est pas claire. Une simulation des erreurs devrait permettre une meilleure évaluation des méthodologies de vérification (Campenhout, 1998).

Des modèles d'erreurs du niveau RT adaptés à la vérification fonctionnelle et semblables à ceux élaborés au niveau porte logique sont présentés dans (Campenhout, 1998). En effet, ce genre de modèles présente un pas vers l'automatisation de la génération des vecteurs de vérification et un critère nécessaire pour une meilleure quantification de l'efficacité de la vérification. Le manque de données publiées sur la nature et la fréquence ainsi que la sévérité des erreurs de design à ce niveau, présentait pendant longtemps un obstacle à l'élaboration et le développement d'un modèle d'erreurs pour la vérification. Dans (Campenhout, 1998), les auteurs ont recueilli les erreurs rencontrées au niveau RT lors de la conception de différents projets, impliquant l'implémentation de différents microprocesseurs et unités arithmétiques et logiques. Ces erreurs ont été analysées et classifiées en différentes catégories. Basé sur le rapport de l'analyse des erreurs détectées, les auteurs ont développé des modèles d'erreurs au niveau RT adaptés à la vérification fonctionnelle d'un design.

Les modèles d'erreurs RTL proposés se divisent en deux catégories (Campenhout, 1998) :

- les modèles d'erreur de base, extraits à partir des modèles de pannes des circuits au niveau portes logiques (Al Asaad, 1995); ces modèles regroupent les cinq modèles d'erreur de base suivants :
  - l'ordre des bits dans un bus (BOE, *« Bus Order Error »*) *:* Ce modèle représente un ordonnancement erroné des différent bits dans un bus;
  - source d'un bus (SBE, *« Source Bus Error »*) *:* Ce modèle représente une mauvaise connexion entre une entrée d'un module et une source;
  - destination d'un bus (BDE, *« Bus Driver Error »*) : Ce modèle représente une connexion d'un bus à deux sources différentes,
  - substitution d'un module (MSE, *« Module Substitution Error »*) : Ce modèle représente le remplacement d'un module par un autre ayant le même nombre d'entrées et de sorties.
- les modèles d'erreurs conditionnelles, développés pour compléter la couverture atteinte avec les modèles de base. Une grande fraction (67 à 75%) des erreurs de conception sont couverts par un ensemble de stimuli complet couvrant les erreurs de base (Campenhout, 1998). Ceci montre que des modèles d'erreurs supplémentaires sont requis pour modéliser l'ensemble total des erreurs de design, d'où les modèles d'erreurs conditionnelles développés dans (Campenhout, 1998) et regroupant l'ensemble des modèles suivants :
  - nombre de bits d'un bus (BCE, *« Bus Count Error »*) *:* Ce modèle représente les entités ayant un nombre erroné de bus d'entrées;
  - nombre de modules d'un design (MCE, *« Module Count Error »*) : Ce modèle représente l'ajout ou la suppression d'un module du design, incluant les portes logiques et les registres;
  - nombre d'étiquettes d'une structure de *case* (LCE, *Label Count Error*) : Ce modèle représente l'ajout ou la suppression de certaines étiquettes d'une structure de *case;*
  - structure d'expression (ESE, *« Expression Structure Error »*) *:* Ce modèle inclut toutes les déviations possibles d'une expression correcte, tels l'ajout ou la suppression d'opérandes et d'opérateurs;

- nombre d'états (SCE*, « State Count Error »*) : Ce modèle réfère au nombre incorrect d'états dans une FSM, le concepteur peut insérer ou supprimer par erreur des états de la FSM;
- Transition (NSE*, « Next State Error »)* : Cette erreur représente les erreurs de transition dans une FSM.

Ces modèles d'erreurs permettent de représenter la majorité des erreurs rencontrées au niveau RT lors de la conception d'un design, il a été démontré dans (Campenhout, 1998) qu'un ensemble de stimuli permettant une couverture complète des modèles d'erreurs prévoit une très bonne couverture des erreurs de conception réelles. Ces modèles d'erreurs seront utilisés pour l'évaluation de la qualité de la méthodologie de vérification proposée dans cette thèse.

2.4 États illégaux d'un design

Dans la littérature, plusieurs travaux et méthodologies ont été développés pour extraire les états illégaux d'un design et les contraintes fonctionnelles de ce dernier. Ces contraintes peuvent être imposées facilement à l'ATPG sans changer son algorithme de fonctionnement (Lin, 2005) afin d'éviter la génération des tests contenant des états illégaux. La génération des contraintes fonctionnelles a été prouvée un problème NP-complet (Liu, 2005).

Certaines des techniques développées sont basées sur l'analyse de structure de la conception et la résolution d'équations modélisant cette dernière (Lin, 2006; Zhang, 2005). Leur complexité de calcul est extrêmement élevée, surtout celles utilisant les solveurs-SAT (Liu, 2008). En raison de leur complexité de calcul ces techniques, utilisées le plus souvent dans des outils ATPG séquentiels, ne sont pas applicables aux circuits complexes de nos jours. D'autres techniques utilisent la simulation pour identifier les états illégaux (Liang, 1997; Lin, 2006; Lin, 2005; Wu, 2007). Les résultats ne sont pas toujours fiables, car les états légaux non-couverts sont considérés comme des états illégaux.

Toutes ces méthodes sont appliquées au niveau logique, d'où la limitation au niveau de la taille et de la complexité du design.

L’identification des états illégaux au niveau RT est une alternative pour résoudre ce problème. Dans (Giomi, 1995), une technique d’extraction de machines à états finis (FSM) est présentée. L'extraction n'est pas directement appliquée au code du langage de description matérielle (HDL, « *Hardware Description Langage* »), mais sur un graphe orienté représentant le modèle (en supposant qu'un tel graphe existe). En outre, les informations extraites ne contiennent que les différents états, ce qui signifie qu'une analyse doit être faite pour extraire les contraintes fonctionnelles sur tous les signaux d'état. Un outil, appelé ATKET, est proposé dans (Lin, 2006) pour l'extraction automatique de contraintes fonctionnelles. L'un des inconvénients de cette méthodologie est que le système de génération de tests auxquels les contraintes sont appliquées doit être conçu sur mesure. Un nouveau procédé de génération de tests fonctionnels a été publié dans (Tupuri, 1997; Vedula V. M., 2000). Bien que l'extraction des contraintes se fait automatiquement, cette méthode vise un module à la fois, et les contraintes imposées par les connexions structurelles et de la hiérarchie modulaire sont générées suite à la synthèse de la logique entourant chaque module. Par conséquent, le processus de synthèse des contraintes et des connexions des modules rend la méthode très complexe à utiliser. Une approche modifiée et automatisée est proposée dans (Vedula V. M., 2000; Vedula, 2000) mais elle implique de nombreuses étapes d’analyse à différents niveaux d’abstraction, résultant en une méthodologie complexe qui consomme beaucoup de temps.

2.5 Conclusion

Dans ce chapitre, nous avons présenté une revue de littérature concernant le domaine de la vérification étant le domaine d’intérêt de cette thèse. On peut voir que, malgré les nouvelles méthodologies développées et implémentées à ce niveau, un grand chemin reste à faire que ce soit au niveau de la diminution des efforts et du temps investis ou de l’amélioration de la couverture. Des combinaisons entre les techniques de test et de vérification ont été élaborées surtout au niveau de la vérification statique. Des recherches restent à faire pour que la vérification dynamique puisse aussi bénéficier de cette combinaison.

CHAPITRE 3

UTILISATION DES TESTS STRUCTURELS DANS LA SIMULATION

3.1 Introduction

Les chapitres précédents présentent deux étapes majeures du processus d'intégration : la vérification et le test. On rappelle que la vérification est désormais une des étapes les plus importantes et les plus exigeantes en termes d'effort et de temps. Son goulot d'étranglement réside dans la détection de ce qu'on appelle les coins sombres (ou « *dark corners* »). Les coins sombres constituent un ensemble de scénarios dont la simulation est effectuée par un sous-ensemble limité (s'il n'est pas vide) des vecteurs de vérification. Une meilleure couverture de ces coins sombres permettra une amélioration de la qualité de la vérification.

En revanche le test, survenant plus tard dans le processus, a évolué plus rapidement que la vérification. Avec l'introduction des techniques DFT et des outils ATPG (Graphics, 2006; synopsys, 2008), l'infrastructure du test s'est améliorée : les circuits complexes peuvent être testés en quelques secondes et le nombre de pannes cachées (« *hard faults* »), a diminué énormément. Les pannes cachées constituent un ensemble de pannes dont la détection est effectuée par un sous-ensemble limité (s'il n'est pas vide) des vecteurs de test.
Afin de tirer profit de l'infrastructure de test pour améliorer la vérification, l'utilisation des vecteurs de tests structurels dans la simulation RT est la méthodologie proposée dans cette thèse.

Dans ce chapitre, nous cherchons tout d'abord à établir une relation entre les coins sombres et les pannes cachées afin de pouvoir justifier et explorer l'approche proposée. Ensuite, nous justifions notre choix de tests structurels utilisés lors de la simulation et qui repose sur les tests de transition LOC avec insertion des registres à balayage. De plus, nous décrivons en détails l'approche adoptée pour l'utilisation de l'ensemble de tests générés par les outils ATPG dans le processus de vérification RTL et nous présentons toutes les transformations

requises pour adapter ces vecteurs au niveau visé. Basé sur ces résultats, l'environnement de vérification complet sera automatisé et présenté dans les prochains chapitres.

Le chapitre 3 est organisé comme suit. La section 3.2 présente la relation existante entre les coins sombres et les pannes cachées. La section 3.3 explique le choix des tests structurels à utiliser dans la vérification. Ensuite la section 3.4 présente le processus d'utilisation de tests structurels dans le processus de vérification. Et enfin, les premiers résultats expérimentaux, issus d'une application manuelle de la méthodologie proposée, sont présentés dans la section 3.5. Ils constituent une preuve de concept pour notre méthodologie, qui sera automatisée dans les chapitres suivants.

3.2 Pannes cachées versus coins sombres

L'origine de la méthodologie proposée vient d'une intuition selon laquelle ce qui est difficile à tester est difficile à vérifier. Dans cette section, nous cherchons à trouver la relation et la corrélation entre ce qui est difficile à tester, i.e. les pannes cachées et ce qui est difficile à vérifier, i.e. les coins sombres.

3.2.1 Pannes cachées

Les pannes cachées sont l'ensemble des pannes difficiles à détecter au niveau portes logiques. Elles représentent des nœuds avec peu ou pas de testabilité et présentent un goulot d'étranglement pour le processus de test. En effet, la détection de ces pannes est effectuée par un sous-ensemble limité (s'il n'est pas vide) et dirigé de l'espace de vecteurs de test. Durant la simulation des pannes, les pannes cachées sont généralement celles couvertes par des tests dirigés par opposition aux pannes plus facilement testables qui sont couvertes par des tests aléatoires.

L'observabilité et la contrôlabilité des nœuds sont les caractéristiques principales pour l'évaluation de la testabilité d'un circuit (la testabilité d'un circuit montre à quel point il sera facile ou difficile de le tester). Afin de mieux modéliser ces deux caractéristiques (ou

paramètres), une approche pour l'analyse de testabilité est proposée dans (Chen C.H., 1989). Dans cette approche, des mesures de contrôlabilité et d'observabilité des circuits sont définies et raffinées afin de gérer les comportements combinatoires et séquentiels. Les paramètres combinatoires sont définis en termes de probabilité alors que les paramètres séquentiels sont définis en termes d'estimation des longueurs des séquences de test. Les quatre paramètres caractérisant les propriétés de la contrôlabilité/observabilité des nœuds internes sont :

- **la contrôlabilité combinatoire** (CC), soit la probabilité que le nœud ait une valeur spécifique;
- **la contrôlabilité séquentielle** (CS), soit la longueur de la séquence attendue (nombre de délais) nécessaires pour qu'un nœud ait une valeur spécifique;
- **l'observabilité combinatoire** (OC), soit la probabilité qu'un changement dans un nœud provoque un changement dans une sortie observable;
- **l'observabilité séquentielle** (OS), soit le nombre de cycles nécessaires pour propager l'effet d'un changement au niveau d'un nœud vers une sortie.

Donc, les pannes cachées sont l'ensemble des pannes associées à des nœuds présentant une faible contrôlabilité et observabilité séquentielle/combinatoire.

3.2.2 Coins sombres

Les coins sombres sont associés à des parties de la fonctionnalité rarement visitées et présentent un goulot d'étranglement au niveau du processus de vérification. Pour couvrir une certaine fonctionnalité d'un design, il faut créer et propager des transitions sur des nœuds bien spécifiques du design.

La difficulté de vérification d'une fonctionnalité donnée dépend de:

- la probabilité de produire la fonctionnalité désirée. Ceci correspond à la probabilité de créer un ensemble de transitions sur différents bits (de différents signaux) du module sous

vérification afin de produire la fonctionnalité désirée. Ceci correspond à la CC des nœuds en question;

- la probabilité d'observer la fonctionnalité désirée. Ceci correspond à la probabilité de propager l'ensemble des transitions créées sur les différents bits (des différents signaux) vers des nœuds observables. Ceci correspond à l'OC des nœuds en question;
- la longueur de la séquence de simulation (nombre de cycles) nécessaire pour forcer la fonctionnalité désirée. Ceci correspond au nombre de cycles et de stimuli nécessaires pour créer les transitions requises sur différents signaux du design afin de le transiter vers l'état désiré. Ceci correspond à la CS des nœuds en question;
- la longueur de la séquence de simulation (nombre de cycles) nécessaire pour observer la fonctionnalité désirée. Ceci correspond au nombre de cycles et de stimuli requis pour propager l'ensemble des transitions créés sur différents signaux vers des nœuds observables. Ceci correspond à l'OS des nœuds en question;

Il apparaît que les mêmes paramètres (CC, CS, OC, OS) peuvent être utilisés et évalués au niveau RT pour analyser la difficulté de vérification du circuit. Ainsi, les coins sombres sont l'ensemble des transitions associées à des nœuds du design, présentant une faible contrôlabilité et observabilité séquentielle/combinatoire. L'ensemble de ces transitions résultent en un ensemble de fonctionnalités difficilement vérifiables.

3.2.3 Corrélation entre les coins sombres et les pannes cachées

On voit que les paramètres principaux affectant l'aptitude à tester et à vérifier un circuit convergent vers la contrôlabilité et l'observabilité de ses nœuds. Ces paramètres peuvent être évalués à deux niveaux: portes logiques et RT. En effet, l'aptitude à créer et propager une transition au niveau des nœuds d'un circuit, que ce soit dans le test ou dans la vérification, est liée directement à la difficulté de contrôler et d'observer ces nœuds à partir des entrées et sorties primaires.

Afin de permettre d'établir une corrélation entre les coins sombres et les pannes cachées qui se trouvent à deux niveaux d'abstraction différents (portes logiques et RT), et étant donné que le niveau porte logique est muni d'un degré de détails et d'un nombre de nœuds beaucoup plus élevé que celui du niveau RT, nous nous restreignons aux nœuds observables et communs aux deux niveaux qui sont : les entrées et les sorties primaires, ainsi que les différents signaux d'état. De plus, vu qu'au niveau RT, les coins sombres de la fonctionnalité sont associés à un ensemble de transitions créés et propagées sur des nœuds spécifiques, nous allons considérer les pannes cachées par rapport aux pannes de délai dont la couverture nécessite la création et la propagation de transitions sur les différents nœuds du design.

Ainsi, parmi l'ensemble des nœuds communs aux deux niveaux RT et portes logiques, les nœuds du design synthétisé qui s'avèrent difficiles à contrôler/observer sont nécessairement issus de certaines fonctionnalités RTL, dont l'ensemble des transitions sont associées à des nœuds présentant une contrôlabilité ou/et observabilité séquentielle/combinatoire faibles. En d'autres termes, les pannes cachées au niveau portes logiques sont le résultat de la synthèse de l'ensemble des coins sombres au niveau RT. Ce raisonnement vient appuyer notre intuition initiale selon laquelle ce qui est difficile à tester est bien difficile à vérifier.

En outre, tel que décrit dans le chapitre 1, différentes techniques ont été élaborées pour améliorer la testabilité des circuits et réduire l'ensemble des pannes cachées d'un circuit :

- la technique DFT basée sur l'insertion des registres à balayage par exemple, permet une amélioration des 4 paramètres décrivant la contrôlabilité et l'observabilité des nœuds. Considérant les nœuds communs aux deux niveaux (entrées sorties primaires et signaux d'états) et les modèles de pannes de délai, les registres à balayage permettent une maximisation de leur CS, CC et OS ainsi qu'une amélioration de leur OC sans atteindre nécessairement le maximum étant donné que ce paramètre dépend du trajet de la propagation de la transition;
- les outils ATPG utilisant des algorithmes avancés de génération de vecteurs de test génèrent un ensemble de vecteurs de tests dirigés permettant une bonne couverture des pannes cachées.

Donc, les nœuds ayant des CS, CO, OC limitées au niveau de la vérification ne présentent pas nécessairement la même limitation au niveau du test à cause des techniques DFT et ATPG et donc on peut déduire que ce qui est difficile à vérifier n'est pas nécessairement difficile à tester.

En conclusion, dû aux différentes techniques développées au niveau du test pour réduire le nombre de pannes cachées, que ce soit les techniques DFT ou les algorithmes avancés implémentés dans les outils ATPG, les pannes cachées deviennent un sous-ensemble des coins sombres. Et donc, il est raisonnable de supposer que ce qui est difficile à tester est sûrement difficile à vérifier mais ce qui est difficile à vérifier n'est pas nécessairement difficile à tester. Plus précisément, en considérant les pannes de délai, nous pouvons dire que les transitions difficiles à créer et propager au niveau de certains nœuds lors du test seront vraisemblablement difficiles à créer et propager au niveau de ces mêmes nœuds lors de la vérification mais l'inverse n'est pas nécessairement vrai.

Cette conclusion, qui sera appuyée plus loin par des résultats expérimentaux, nous encourage à orienter nos recherches vers l'utilisation des tests structurels de transition capables de couvrir une grande partie des pannes cachées, dans le processus de vérification afin d'améliorer la couverture des coins sombres.

3.3 Le choix des tests structurels

Tel qu'indiqué précédemment, la méthodologie proposée dans cette recherche regroupe deux niveaux d'abstraction différents, le niveau RT et celui des portes logiques. Dans le but de limiter les divergences et de faciliter la corrélation entre les deux niveaux d'abstraction et surtout d'avoir une bonne couverture de vérification, le choix des tests structurels se porte sur les tests de délai de transition LOC avec insertion des registres à balayage. Le choix de chacun des critères est expliqué dans ce qui suit :

- **tests de délai** : avant de générer et d'appliquer les tests structurels dans le processus de vérification, nous devons choisir un modèle de panne pour lequel les vecteurs de test

générés permettent de simuler et d'exercer efficacement les fonctionnalités du design. Dans un processus de vérification des designs dominés par le contrôle, notre objectif principal est de vérifier tous les états et les transitions de la machine à états. La simulation de tels systèmes cherche ainsi à transiter le système d'un état à l'autre en couvrant toutes les transitions possibles, et donc de créer et propager des transitions sur les différents signaux du design, plus particulièrement les entrées, signaux d'état et sorties du système. Les tests visant les pannes de délai se basent sur le même principe. Tel que décrit dans le chapitre 1, ce type de test cherche à créer des transitions sur différents nœuds du design et à les propager vers des sorties observables. Parmi les tests structurels, ce type de tests paraît le plus adéquat pour mieux simuler la fonctionnalité du design et couvrir le plus d'états et de transitions possibles de la machine à états;

- **panne de délai transitionnelle:** parmi les deux types de modèles de pannes de délai populaires dans la littérature, nous avons choisi les pannes de délai transitionnelles (Waicukauski, 1987) par rapport aux pannes de délai de chemin (Smith, 1985), ceci en raison de la complexité de la génération des vecteurs de tests de ce dernier. En effet, un circuit logique de n nœuds dispose d'un maximum de $2n$ pannes de délai transitionnelle (une montante et une descendante sur chaque ligne) alors qu'il dispose de n^2 de pannes de délai de chemin (Savir, 1993);
- **tests basés sur les registres à balayage** : dans ce genre de test, les vecteurs de test générés sont appliqués et observés non seulement au niveau des entrées/sorties primaires mais aussi au niveau des registres à balayage. L'utilisation de ces vecteurs de test au niveau de la vérification permettra de transformer le comportement séquentiel des circuits en un comportement combinatoire, et remplacer les longues séquences de vecteurs de test requis pour couvrir certains états, par un seul vecteur forçant les signaux d'état à l'état en question. L'émulation des registres à balayage durant la simulation permet une meilleure contrôlabilité et observabilité, diminuant ainsi le temps de la vérification tout en améliorant sa couverture atteinte;
- **tests LOC :** on rappelle que le test de délai suppose l'application de deux vecteurs de test (V_1, V_2) tels que: V_1 est un vecteur d'initialisation et V_2 un vecteur qui lance et propage la transition vers une sortie primaire ou un nœud observable (Waicukauski, 1987). Les tests

de transition appliqués selon le LOC ou *« Broad-side »* (Savir, 1994) permettent une meilleure simulation de la fonctionnalité du design étant donné que c'est le seul type où le second vecteur de test est la réponse du circuit au premier vecteur de test qui réside dans les chaines à balayage, alors que dans les autres types le second vecteur est un vecteur forcé tout comme le premier.

En conclusion, les tests de transition LOC basés sur les registres à balayage sont les tests choisis pour être utilisé au niveau de la vérification afin de permettre une meilleure simulation de la fonctionnalité du design.

3.4 Utilisation des vecteurs de test structurels dans la vérification RTL

Cette section décrit en détails l'utilisation des vecteurs de test LOC avec registres à balayage dans la vérification. La Figure 3.1 décrit un flot de conception standard et sa relation avec le test et la vérification, plus particulièrement le test basé sur des registres à balayage et la vérification des modèles RTL. Dans cette figure, nous avons introduit la méthodologie proposée qui consiste en une utilisation des tests structurels dans la simulation au niveau RT.

Cette méthodologie est basée essentiellement sur deux étapes : la génération des vecteurs de test structurels et l'application de ces derniers dans la simulation RTL.

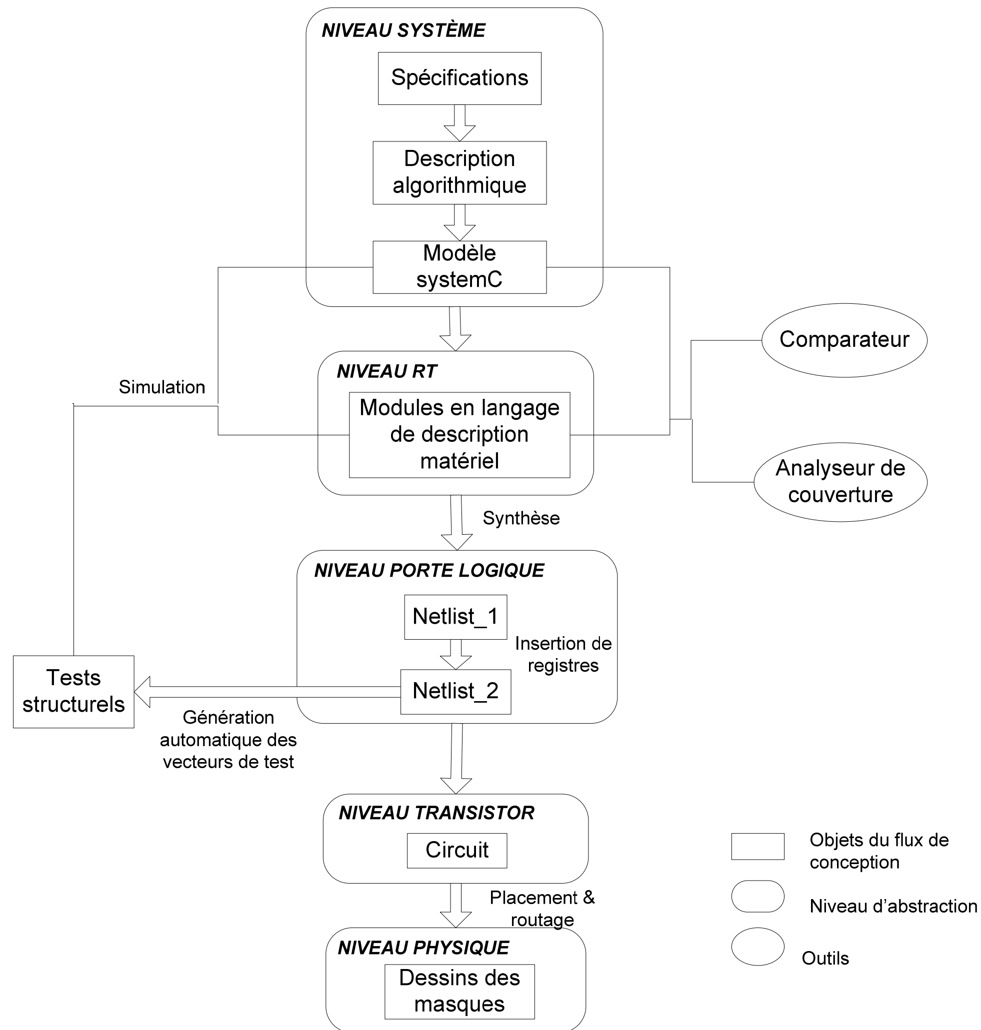


Figure 3.1 Flux de conception standard implémentant l'utilisation des tests structurels dans la vérification. Adaptée de Lam (2005, p.3)

3.4.1 Génération des vecteurs de test structurels

Tel que décrit dans la section 3.3, notre choix du modèle des tests structurels utilisés dans la vérification se porte sur les tests de transition LOC basés sur les registres à balayage, afin d'assurer une meilleure simulation de la fonctionnalité ainsi qu'une amélioration de la couverture et du temps de simulation. Les étapes nécessaires pour la génération de ces tests structurels ainsi que leur emplacement dans un flux de conception standard sont décrits dans la Figure 3.1. Le modèle RTL est d'abord synthétisé. Notons que cette synthèse peut être

effectuée sans qu'aucune contrainte ne soit appliquée, afin de réduire l'effort de synthèse. Ainsi cette étape peut être utilisée pour vérifier si le code VHDL est synthétisable et donner une première estimation du nombre de portes logiques requis. Après la synthèse, les registres à balayage sont insérés, générant un nouveau « *netlist* » pour lequel l'outil ATPG (Graphics, 2006; synopsys, 2008) génère l'ensemble des vecteurs de test de transition LOC.

Ces étapes de génération de vecteurs de test peuvent être effectuées en quelques minutes (pour les exemples considérés dans cette thèse) et sont prêts à être utilisés dans le processus de vérification du modèle RTL non synthétisé.

3.4.2 Application des tests de transition LOC avec registres à balayage au modèle RTL

Les vecteurs de test utilisés dans la simulation visent initialement un niveau plus bas que celui de la vérification, le niveau porte logique. Il est difficile d'obtenir un mappage direct entre un modèle porte logique et un modèle RTL. Différentes transformations doivent être prévues pour permettre l'adaptation et l'application des tests structurels au niveau RT.

Test de transition LOC avec registres à balayage

Tel que décrit dans le chapitre 1 et illustré dans la Figure 3.2 Test de transition STF (STR) sur un nœud X_i., un vecteur de test de transition STF (STR) LOC (V1, V2) visant une panne de délai sur un nœud X_i est constitué :

- d'un vecteur d'initialisation V_1 qui force la valeur 1 (0) sur le nœud X_i.
- d'un vecteur V_2 qui est le vecteur de test « *stuck-at* » 0(1) créant la transition *T* au niveau X_i et la propageant vers un nœud observable, Y_l.

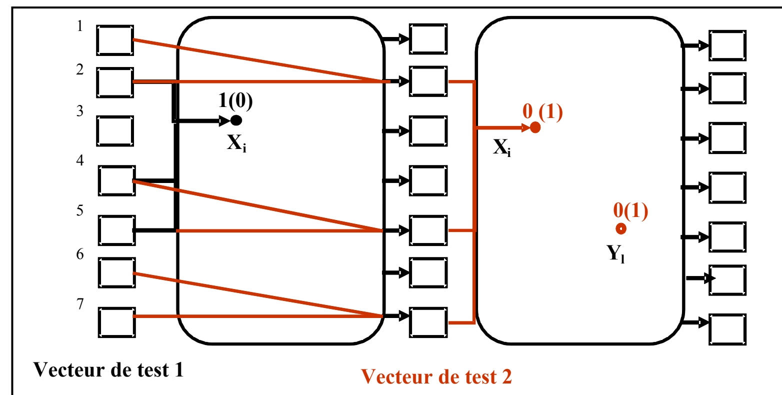


Figure 3.2 Test de transition STF (STR) sur un nœud X_i.

Ainsi, l'application d'un vecteur de test LOC typique inclut les étapes suivantes :

- charger la chaine registres à balayage;
- forcer les entrées primaires;
- créer un front d'horloge;
- forcer les entrées primaires;
- mesurer les sorties primaires;
- front d'horloge;
- décharger la chaine de scan.

Un exemple d'une partie d'un modèle de test structurel généré par un outil ATPG (*Fastscan de Mentor* (Graphics, 2006)) est décrit dans la Figure 3.3.

```
//ENSEMBLE DES PORTS D'ENTRÉES REGROUPÉS DANS LE BUS PI
declare input bus "PI" = "/CLOCK", "/INPUT_1", "/INPUT_N",
"/scan_in1", "/scan_en";

Pattern = 0 clock_sequential;
//FORCER LA CHAINE DE REGISTRES AVEC LE PREMIER VECTEUR
    Apply "grp1_load" 0 =
    Chain "chain1" = "01000000101011110";
    End;
//FORCER LES ENTRÉES PRIMAIRES
    Force   "PI" "00011" 1;
//SECOND VECTEUR RÉPONSE DU CIRCUIT AU PREMIER
    Pulse "clock" 2;
    Force   "PI" "00100" 3;
    MEASURE "PO" "00000" 4;
    pulse "/CLOCK" 5;
    apply "grp1_unload" 6 =
    chain "chain1" = "10010011010011011";
  End;
//LA CHAINE DE REGISTRES REGROUPÉS DANS LE BUS CHAIN1
    SCAN_CELLS =
    scan_group "grp1" =
    scan_chain "chain1" =
    scan_cell =    0  MASTER  FFFF  "/STATE_SIGNAL1_reg[2]"
"_i2/mlc_dff2"  "SD"  "q";
    scan_cell =    0  MASTER  FFFF  "/STATE_SIGNAL1_reg[0]"
"_i2/mlc_dff2"  "SD"  "q";
    scan_cell =    0  MASTER  FFFF  "/STATE_SIGNAL1_reg[1]"
"_i2/mlc_dff2"  "SD"  "q";
...
```

Figure 3.3 Exemple d'un modèle de test LOC avec registres à balayage.

Donc un modèle de test structurel, généré avec les outils ATPG après insertion des registres à balayage, exerce non seulement les entrées primaires mais aussi les chaines de registres à balayage insérés dans le design. De plus, les valeurs de vecteurs de test contenues dans ce type de modèle se trouvent sous forme de séries de bits '0' ou '1', où toutes les entrées primaires sont regroupées en une seule variable, de même que pour les registres à balayage.

Transformation du test structurel LOC avec registre à balayage en un vecteur de simulation RTL

La génération d'un vecteur de simulation RTL à partir des modèles de tests structurels LOC basés sur l'insertion des registres à balayage requiert une adaptation vers un niveau

d'abstraction plus élevé d'un tel vecteur conçu essentiellement pour être appliqué au niveau logique et ceci se fait en plusieurs étapes:

- **identification des différents ports et registres à balayage ainsi que leurs vecteurs de test correspondants**. Dans un modèle de test structurel les vecteurs de test sont appliqués seulement à deux variables, une correspond à l'ensemble des ports primaires regroupés et l'autre à l'ensemble des registres à balayage regroupés. Pour cette raison, une analyse du fichier contenant le résultat de la génération des vecteurs de test est requise pour identifier :
  - chacun des ports primaires et des registres à balayage forcés par l'ensemble des tests;
  - leur position au niveau des deux variables regroupant l'ensemble des ports primaires et des registres à balayage;
  - les valeurs des vecteurs de test correspondantes à chacun des ports primaires et des registres à balayage contenus dans chacun des patrons de tests. Les registres à balayage et les ports primaires sont exercés avec des vecteurs de séries de bit au niveau structurel, mais peuvent avoir des types différents aux niveaux d'abstraction plus élevés. Considérant le niveau RT et les deux langages de description matérielle ciblés (VHDL et SystemC), et dépendamment des types des signaux et ports forcés aux différents niveaux, une transformation des vecteurs de séries de bits des tests structurels vers les valeurs correspondantes aux types en question doit être effectuée.
- **émulation des registres à balayages au niveau RT**. Les vecteurs de test structurels exercent les entrées primaires et les registres à balayage. La différence entre ce modèle de test et un vecteur typique de vérification est le chargement des chaines de registres à balayage. Ces nœuds peuvent être associés aux signaux d'état internes, au niveau RT. Donc le forçage des registres à balayage est transformé en un forçage des signaux d'état internes du DUV. Donc l'application des vecteurs de test de transition avec registres à balayage au niveau de la simulation induit une insertion des registres à balayage virtuels, cette fois non pas au niveau porte logique mais au niveau RT, permettant une amélioration de la contrôlabilité et l'observabilité combinatoire et séquentielle. Ainsi, pour couvrir un état donné d'une FSM, un seul vecteur de test suffit, ce vecteur peut forcer le design dans l'état voulu. Les longues séquences de test ne sont plus nécessaires.

La fonction « *force_signal* » est utilisée avec Modelsim de Mentor afin de forcer les signaux internes ;

- **génération du vecteur de vérification RTL**. La génération du vecteur de simulation RTL correspondant au vecteur de test structurel sera effectuée sur deux cycles d'horloge comme suit :
  - 1er cycle : l'application du vecteur d'initialisation V_1; ce vecteur permet de forcer chacun des ports d'entrées primaires et des signaux d'état à des valeurs initiales correspondantes à leurs types respectifs;
  - 2ème cycle : L'application du vecteur de propagation de transition V_2; les entrées primaires sont forcées et les signaux d'état auront les valeurs correspondantes à la réponse du circuit suite au premier cycle d'horloge.

On peut voir que ces vecteurs de vérification permettent de forcer les entrées comme pour n'importe quel autre vecteur de vérification, mais en plus une émulation des registres à balayage est faite en forçant les signaux d'état internes. Le code équivalent adapté à la vérification RTL de l'exemple de test présenté dans la figure est décrit ci-dessous.

```
//1er CYCLE:
//PREMIER VECTEUR V1: FORCER LES SIGNAUX D'ÉTATS
  Force_signal (''duv/module_name/state_signal1'',''3'','''',
  "verbose");
  Force_signal
  (''duv/module_name/sub_module_instance/sub_sub_module_instan
  ce/state_signal2'', "6",'' '', "verbose");
// PREMIER VECTEUR V1 : FORCER LES ENTRÉES PRIMAIRES
  input_1<=''3'';
  input_n<=''0'';
//2ème CYCLE : LE SECOND VECTEUR V2 RÉPONSE DU CIRCUIT AU PREMIER
  Wait for (clock_period);
  input_1<=''0'';
 input_n<=''1'';
```

Figure 3.4 Vecteur de simulation RTL formé à partir du modèle de test LOC de la Figure 3.3.

Ainsi, l'application des vecteurs de test structurels au niveau RT reste possible en dépit de la faible corrélation entre les deux niveaux.

3.5 Premiers résultats expérimentaux

Ces premiers résultats expérimentaux sont divisés en deux parties :

- la première permet de démontrer la corrélation déduite précédemment entre les pannes cachées et les coins sombres;
- la deuxième présente quelques résultats préliminaires de l'efficacité de l'utilisation des vecteurs de test dans la simulation RTL des designs.

Pour chacune des parties nous présentons les expériences effectuées, les différents designs utilisés ainsi que les résultats correspondants.

3.5.1 Corrélation entre les coins sombres et les pannes cachées

Cette première partie des résultats présente des expériences exécutées aux deux niveaux d'abstraction RTL et portes logiques. Ces expériences permettent de comparer l'ensemble des coins sombres au niveau RT avec l'ensemble des pannes cachées au niveau portes logique pour le même design. Le but étant de démontrer la corrélation entre ce qui est difficile à vérifier et ce qui est difficile à tester.

3.5.1.1 Expériences

Afin de démontrer la corrélation existante entre ce qui est difficile à vérifier et ce qui est difficile à tester, nous procédons à la vérification RTL et au test des mêmes designs et ensuite, nous identifions l'ensemble des coins sombres et des pannes cachées correspondants afin de déduire la relation existante entre les deux ensembles. Les expériences sont réalisées à deux niveaux :

- au niveau RT, la vérification est effectuée basée sur une simulation pseudo-aléatoire du design. Un banc d'essai simule le design en deux étapes : dans une première étape des vecteurs de test aléatoires sont appliqués et dans une deuxième étape des tests directs

implémentés manuellement complètent la simulation. L'outil utilisé est Modelsim. Suite à la vérification, l'analyse de la couverture et de la testabilité des nœuds est réalisée pour déduire la liste des coins sombres. Tel que défini dans les sections précédentes, les coins sombres sont l'ensemble de transitions associées à des nœuds du design présentant une faible contrôlabilité et observabilité séquentielle/combinatoire. Donc, afin de les identifier, nous commençons par définir l'ensemble complet des transitions possibles des différents nœuds du design RTL et ceci en se basant sur un modèle d'erreur appelé le modèle d'erreur de transition entrée sortie (TRIO, « *Input Output Transition* ») (Kang, 2007). Ensuite nous procédons à l'identification des coins sombres. Les deux étapes sont réalisées de la manière suivante :

- **génération du diagramme TRIO.** Le modèle d'erreur TRIO est défini sur un ensemble S des variables RTL d'un module donné, où S consiste en l'ensemble des entrées et des sorties primaires, ainsi que les variables d'état (registre, bascule). Une erreur TRIO est une paire $(<V_i, T_i>, <V_j, T_j>)$, où V_i est un bit de l'entrée primaire ou d'une variable d'état et V_j est un bit d'une sortie primaire ou d'une prochaine variable d'état, T_i est une transition montante ou descendante sur V_i et T_j est une transition montante ou descendante sur V_j. Donc il s'agit d'une propagation de la transition de V_i jusqu'en V_j. À partir de cette définition, nous générons le diagramme TRIO correspondant à chacun des designs vérifiés et nous déduisons l'ensemble des transitions à couvrir;
- **identification des coins sombres**. Une transition au niveau d'un nœud fait partie des coins sombres si la transition au niveau de ce nœud :
  - n'est pas couverte par aucun vecteur du banc d'essai de la simulation. Dans ce cas, cette transition est associée au degré de difficulté maximal;
  - est couverte par des vecteurs de test peu fréquents, avec une faible probabilité d'être générée comme les tests directs complexes;
  - est couverte suite à l'application d'une longue séquence de vecteurs de test.

  Basé sur ces critères, on associe à chacune des transitions un degré de difficulté différent qui peut varier de 0 à 4. Les transitions non couvertes par la simulation ont le degré 4 et représentent les transitions les plus difficiles à vérifier. Celles couvertes par

les tests directs et non aléatoires (tests complexes, longues séquences) ont le degré 3. Celles couvertes par un petit pourcentage des vecteurs aléatoires (<30%) ont le degré 2. L'ensemble de ces transitions ayant les degrés 2 à 4 représentent les coins sombres. Finalement l'ensemble des transitions couvertes par un pourcentage des vecteurs aléatoires variant entre 30% et 70% ont le degré 1 et celles couvertes par un pourcentage de vecteurs aléatoires > 80 % ont le degré 0.

- au niveau porte logique, nous effectuons un test du design basé sur le modèle de transition avec insertion des registres à balayage, suivant les étapes ci-dessous :
  - synthèse du circuit. (Design_Vision de Synopsys);
  - insertion des registres à balayage . (Dft_advisor);
  - génération des vecteurs de test pseudo-aléatoires (Dft_advisor);
  - simulation des pannes de transition. (Fast_scan).

  Ensuite, l'analyse de la couverture des pannes de transition et de la testabilité des nœuds est réalisée pour l'identification de la liste des pannes cachées. En se limitant aux nœuds observés au niveau RTL (entrées/sorties primaires et signaux d'état), nous calculons le degré de difficulté associé à toutes les propagations de transitions possibles entre ces nœuds. En effet, ce degré de difficulté varie non seulement avec le pourcentage de nombre de tests qui l'ont couvert mais aussi avec le type de ces derniers, les tests déterministes visent les propagations difficiles à tester alors que les tests aléatoires visent ceux qui sont faciles à tester.

  Afin d'identifier les pannes cachées, nous associons un degré de difficulté à chacune des transitions à tester. Le degré de difficulté pour le test est défini tel que, une transition est dite panne cachée si cette transition est couverte par :

  - aucun test structurel. Le degré de difficulté est maximal dans ce cas ;
  - les vecteurs déterministes ;
  - un nombre limité de tests structurels.

  Basé sur ces critères, on associe à chaque panne de transition un degré de difficulté différent qui peut varier de 0 à 4. Les pannes non couvertes par la simulation ont le degré 4 et représentent les pannes les plus difficiles à tester. Celles couvertes par les tests directs

et non aléatoires ont le degré 3. Celles couvertes par un petit pourcentage des vecteurs aléatoires (<30%) ont le degré 2. L'ensemble de ces pannes ayant les degrés 2 à 4 représentent les pannes cachées. Finalement l'ensemble des transitions couvertes par un pourcentage des vecteurs aléatoires variant entre 30% et 70% ont le degré 1 et celles couvertes par un pourcentage de vecteurs aléatoires > 80 % ont le degré 0.

3.5.1.2 Conceptions

Deux circuits ont été l'objet de nos manipulations :

- **le circuit 1** : Il s'agit d'un compteur modulo 4 (Kang, 2007). La description RTL de l'entité est présentée dans la Figure 3.5, et le diagramme TRIO correspondant est présenté dans la Figure 3.6.

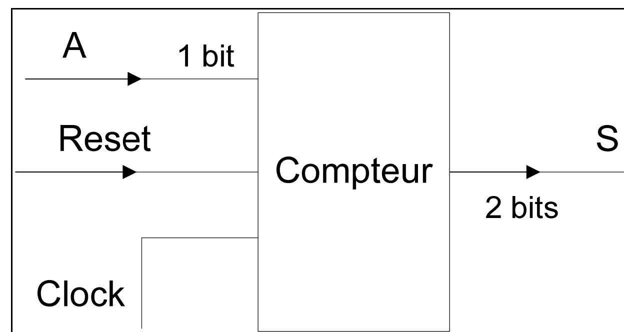


Figure 3.5 Description RTL de
l'entité du circuit 1.
Tirée de (Kang, 2007)

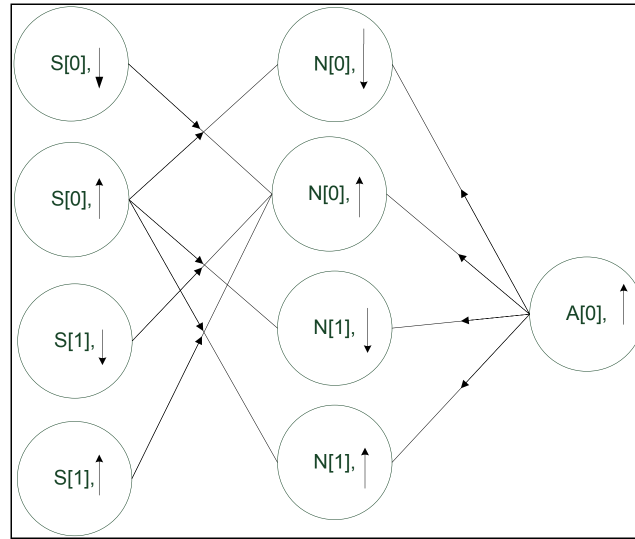


Figure 3.6 Le diagramme TRIO
correspondant au circuit 1.
Tirée de (Kang, 2007)

D'après le diagramme TRIO, nous déduisons l'ensemble des transitions possibles à vérifier et à tester, qui représente l'ensemble des arcs présents dans le diagramme. Les transitions sont présentées dans le Tableau 3.1, *N* étant un signal interne du circuit;

Tableau 3.1 L'ensemble des transitions du circuit 1,
avec Tm= Transition montante et Td= transition descendante.

Étiquette	**Transition**	**Étiquette**	**Transition**
T0	A, Tm=>N[0], Tm	**T1**	A=>N[0], Td
T2	A, Tm=>N[1], Tm	**T3**	A=>N[1], Td
T4	S[0], Td=>N[0], Tm	**T5**	S[0], Tm=>N[0], Td
T6	S[0], Tm=>N[1], Td	**T7**	S[0], Tm=>N[1], Td
T8	S[1], Td=>N[0], Tm	**T9**	S[1], Tm=>N[0], Tm

- **le circuit 2**: Il s'agit d'un contrôleur de feux de circulation. La description RTL de l'entité est présentée dans la Figure 3.7.

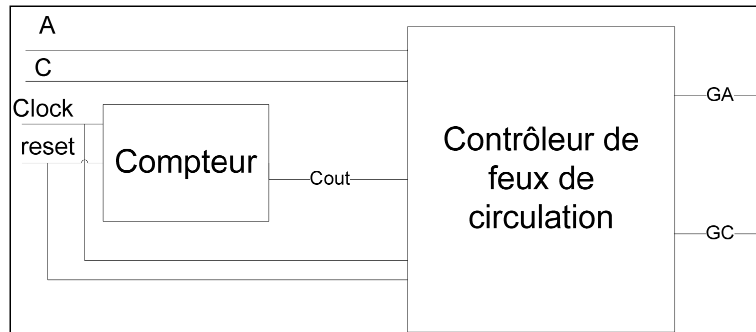


Figure 3.7 Description RTL de l'entité du circuit 2.

D'après le diagramme TRIO, nous déduisons l'ensemble des transitions possibles à vérifier et à tester, qui représente l'ensemble des arcs présents dans le diagramme. Les transitions sont présentées dans le Tableau 3.2.

Tableau 3.2 L'ensemble des transitions possibles du design 2, avec Tm= Transition montante et Td= transition descendante.

Étiquette	Transition	Étiquette	Transition
T0	A, Tm=>S[0], Tm	**T1**	A,Tm => S[1], Td
T2	A, Td=> S[0], Td	**T3**	C, Tm => S[0], Td
T4	C,Tm =>S[1], Tm	**T5**	C, Td=> S[1], Td
T6	Cout, Tm=> S[0], Td	**T7**	Cout, Tm=> S[0], Tm
T8	Cout, Tm=> S[1], Tm	**T9**	Cout, Tm=> S[1], Td
T10	S[0], Tm=> GA, Tm	**T11**	S[0], Td=> GA, Td
T12	S[0], Td=> GC, Tm	**T13**	S[1], Tm=> GC, Tm
T14	S[1], Tm=> GA, Td	**T15**	S[1], Td=> GC, Td

3.5.1.3 Les résultats expérimentaux

Les deux modules sont simulés avec des vecteurs pseudo-aléatoires et testés avec des vecteurs de transition LOC basés sur les registres à balayage. Des degrés de difficulté de vérification et de test sont assignés à chacune des transitions des 2 circuits. Les résultats sont

superposés dans le diagramme de la Figure 3.8 pour le design 1 et celui de la Figure 3.9 pour le design 2.

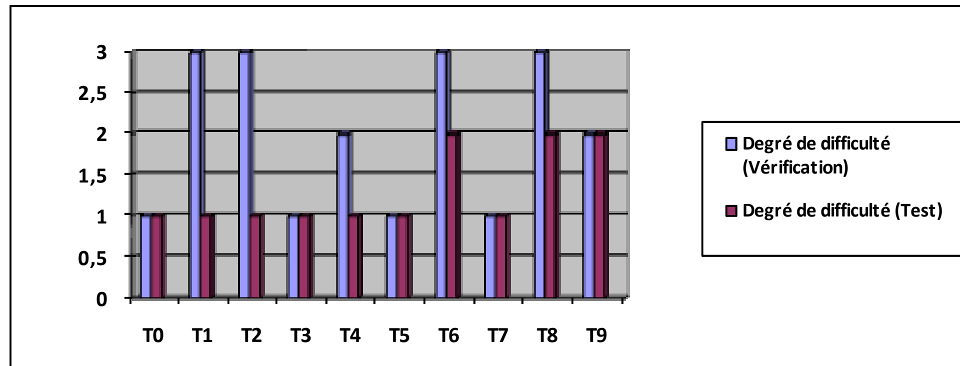


Figure 3.8 Superposition des résultats des deux niveaux RT et porte logique pour le circuit 1.

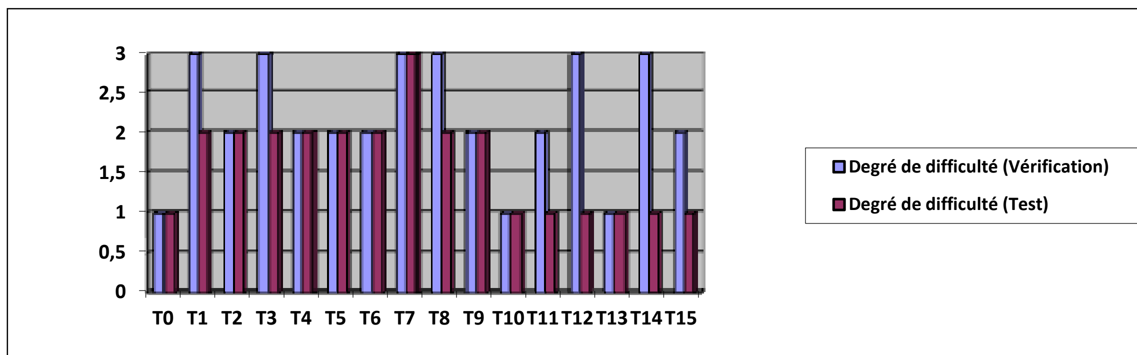


Figure 3.9 Superposition des résultats des deux niveaux RT et porte logique pour le circuit 2.

D'après les deux diagrammes, nous pouvons constater que les transitions difficiles à tester sont difficiles à vérifier, alors que celles qui sont difficiles à vérifier ne sont pas nécessairement difficiles à tester. Si nous associons les coins sombres à l'ensemble des transitions au niveau RT ayant un degré de difficulté de vérification élevé (2,3,4) et les pannes cachées à l'ensemble des transitions au niveau porte logique ayant un degré de difficulté de test élevé (2,3,4), nous pouvons conclure que les pannes cachées sont un sous-ensemble des coins sombres. Cette corrélation justifie la démarche proposée basée sur

l'utilisation des vecteurs de test dans la vérification permettant de réduire l'ensemble des coins sombres.

3.5.2 Utilisation des vecteurs de test dans la vérification

Les circuits expérimentaux utilisés ici sont deux des benchmarks ITC99 (Davidson): B01, une FSM qui compare des flux en série et B02 une FSM qui reconnaît des numéros BCD. Ces circuits ont été synthétisés par un synthétiseur logique commercial (Synopsys de Design Vision). Après la synthèse, l'insertion des registres à balayage se fait à l'aide d'un outil commercial (DFTAdvisor Mentor). Les séquences de tests structurels sont générées par un outil ATPG (FASTSCAN de Mentor) pour le modèle de panne de transition LOC. Basés sur les principes présentés dans la section 3.4, ces vecteurs de tests sont adaptés et insérés manuellement dans un banc d'essai permettant la simulation du design au niveau RT.

Nous comparons les résultats obtenus avec les techniques de vérification basées sur les contraintes et pseudo-aléatoire (Lam, 2005). La comparaison des couvertures est présentée au Tableau 3.3.

Tableau 3.3 Comparaison des couvertures des différentes techniques de vérification.

Circuits	Couverture	Pseudo-aléatoire	Basée sur les contraintes	Utilisation des tests structurels
B01	Couverture d'états	100%	100%	100%
	Couverture de transitions	81.5%	100%	98%
B02	Couverture d'états	71.5%	85.8%	100%
	Couverture de transitions	62.5%	81.2%	87.5%

Ces résultats révèlent que la couverture d'états et de transitions fournie par l'utilisation des tests structurels est supérieure à celle de l'approche aléatoire et comparable à celle basée sur les contraintes (tout en requérant beaucoup moins d'efforts). En effet, les outils ATPG génèrent des vecteurs de test, basés sur des algorithmes avancés, afin de couvrir les pannes cachées. Par conséquent, ces modèles aident à couvrir la plupart des coins sombres que les

vecteurs pseudo-aléatoires sont incapables d'atteindre, conduisant à une couverture très élevée. Dans le chapitre 6, nous présenterons l'environnement de vérification complet et automatisée basée sur l'approche présenté dans ce chapitre.

3.6 Conclusion

Dans ce chapitre, nous avons établi une relation entre les pannes cachées et les coins sombres montrant que ce qui est difficile à tester est difficile à vérifier, mais ce qui est difficile à vérifier n'est pas nécessairement difficile à tester. Ceci découle du développement de plusieurs techniques et outils ATPG basé sur des algorithmes complexes permettant une amélioration de la qualité du test et surtout de sa couverture. Cette corrélation a permis une orientation de notre recherche vers l'utilisation des vecteurs de test dans la vérification dans le but de couvrir les coins sombres afin d'améliorer la qualité de la vérification dynamique en réduisant le temps de conception des bancs d'essai d'une part, et en améliorant la qualité de la couverture d'une autre part. Le choix des tests structurels ainsi que les différentes étapes et transformations requises pour l'adoption de cette méthodologie ont été présentés dans ce chapitre. Nos premiers résultats justifient en quelque sorte l'intérêt de la méthodologie proposée sur la vérification dynamique.

Afin de démontrer que les tests structurels peuvent détecter des erreurs au niveau RT, le chapitre suivant portera sur une étude théorique basée sur un modèle de pannes RTL montrant la capacité d'un vecteur de transition LOC basé sur les registres à balayage à détecter chacun de ces modèles de panne.

CHAPITRE 4

ÉTUDE THÉORIQUE DE LA DÉTECTION DES ERREURS DU DESIGN AU NIVEAU RTL À L'AIDE DES TESTS DE TRANSITION LOC

4.1 Introduction

Dans ce chapitre, nous présentons une étude théorique de l'efficacité de l'utilisation des vecteurs de test structurels de transition LOC pour détecter les erreurs de design RTL durant la vérification. L'idée est de justifier et de démontrer l'efficacité de la méthodologie proposée dans cette thèse qui permet de renforcer l'approche de la vérification fonctionnelle avec des vecteurs de test LOC basés sur les registres à balayage et générés à partir du DUV.

L'étude effectuée dans ce chapitre cherche à montrer que malgré les erreurs que peut contenir le DUV, les tests structurels générés à partir de ce design sont en mesure de couvrir la plupart des erreurs de conception lors de la simulation RTL. Afin de couvrir toutes les erreurs RTL possibles, cette étude se base sur les modèles de pannes RTL développés dans (Campenhout, 1998) pour modéliser l'ensemble des erreurs rencontrées à ce niveau.

En outre, étant donné la présence des états illégaux dans ce type de vecteurs, leur utilisation lors de la simulation peut provoquer un comportement différent du DUV et du modèle de référence, si ces derniers possèdent des styles de code différents, induisant ainsi la détection de fausses erreurs. Nous décrirons en détails le problème dû à l'utilisation de tels vecteurs dans la simulation.

Le chapitre est divisé comme suit. Une première partie présente l'étude théorique sur l'efficacité de l'utilisation des vecteurs de test de transition LOC basés sur les registres à balayage, dans la détection de chacun des modèles de pannes RTL. Ensuite, nous décrirons l'utilité de l'utilisation des vecteurs de tests de transition par rapport aux vecteurs de test « *stuck-at* ». Enfin, le problème causé par l'existence des états illégaux dans l'utilisation de ce type de vecteurs dans la vérification est présenté et illustré par un exemple pratique.

4.2 Hypothèses

H1 : Un vecteur de test de transition LOC, généré pour couvrir une panne de transition STR (STF) sur un nœud X_i du circuit, crée une transition montante (descendante) sur le nœud X_i et la propage à un nœud observable Y_l, avec $Y_l = F(X_i)$ et F représentant la logique combinatoire reliant X_i à Y_l.

Étant donné que la transition se propage de X_i vers Y_l à travers la fonction F, alors F est tel que : $F(X_i=0) \neq F(X_i=1)$ et donc:

- Toute transition T sur X_i crée une transition T ou T' sur Y_l, avec T'= inverse (T).
- $F(X_i = T') \neq F(X_i=T)$. Une transition T sur X_i crée un comportement sur Y_l qui est différent que celui créé par une transition T'.

H2 : Un ensemble de tests de transition complet peut être généré pour un circuit (erroné ou non). Un ensemble de tests d'un circuit erroné est dit complet si l'ensemble des tests suffisent pour détecter toutes les pannes de transition. Pour chaque nœud, on admet deux types d'erreur STR et STF. La complétude de l'ensemble des tests signifie que chacune des pannes est détectée au moins une fois par les vecteurs de cet ensemble(Abadir, 1988).

Définition : nous appelons une erreur RTL redondante si, quelque soit les conditions appliquées, le circuit erroné et le circuit idéal ont le même comportement. Une erreur RTL redondante ne peut donc être détectée, ceci quelque soit la condition. On suppose dans ce qui suit, que les erreurs sont non redondantes et donc il existe au moins un stimuli qui permet de les détecter et ceci en provoquant des comportements différents au niveau des deux modèles DUV et de référence.

4.3 Détection des erreurs RTL à l'aide des vecteurs structurels LOC et limitations

Dans cette section, une étude est effectuée sur l'efficacité et les limitations de la détection des erreurs au niveau RT à l'aide des vecteurs de test de transition LOC basés sur les registres à

balayage et générés par des outils ATPG au niveau portes logiques. Nous distinguons trois cas de figures :

- la détection du modèle d'erreur par les tests de transition est impossible;
- la détection du modèle d'erreur par les tests de transition est garantie;
- la détection du modèle d'erreur par les tests de transition est possible mais non garantie.

Nous discuterons dans ce qui suit les différents modèles d'erreurs RTL et nous présenterons le cas de figure correspondant à chacun d'eux. De plus, nous décrirons les méthodes à utiliser pour maximiser la détection des erreurs dans le cas où la détection n'est pas garantie.

4.3.1 Le modèle BOE

Ce modèle représente un ordonnancement erroné des différents bits d'un bus. On considère X un bus où le i^{ème} et le j^{ème} bits (X_i, X_j) sont renversés, tel que décrit dans la Figure 4.1.

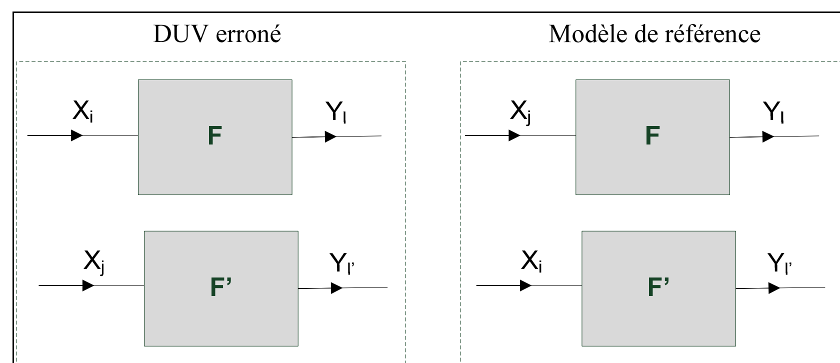


Figure 4.1 Exemple d'erreur de type BOE.

Les vecteurs de test de transition, STF ou STR, LOC (V1, V2) générés pour détecter les pannes de transition au niveau du nœud X_i (X_j) génèrent une transition T sur X_i (X_j) et la propage vers des nœuds observables Y_l ($Y_{l'}$).

Considérons un test de transition du nœud X_i (le même raisonnement s'applique pour le nœud X_j). Il existe 3 cas de test possibles basés sur les comportements de X_i et X_j, tel que décrit au Tableau 4.1.

Tableau 4.1 Les différents tests de transition possibles du nœud X_i, basés sur les comportements de X_i et X_j.

Test	X_i	X_j	Description
Test 1	T	T	Le test crée la même transition T sur X_i et X_j
Test 2	T	T' avec T'= inv(T)	Le test crée une transition T sur X_i et une transition inverse T' sur X_j
Test 3	T	Constante (0 ou1)	Le test crée une transition T sur X_i et garde une valeur constante sur X_j

Ainsi, en appliquant ces vecteurs de test de transition du nœud X_i, lors de la simulation du DUV et du modèle de référence, nous observons le comportement décrit dans le Tableau 4.2. Dans ce tableau, nous prenons en considération que le DUV est affecté par une erreur de type BOE. Les bits X_i et X_j du modèle de référence sont échangés dans le DUV, donc :

- X_i(duv)= X_j(modèle de référence) alors
  Y_l(duv)= f(X_j(modèle de référence))= $Y_{l'}$(modèle de référence);
- X_j(duv)= X_i(modèle de référence) alors
  $Y_{l'}$(duv)= f(X_i(modèle de référence))= Y_l(modèle de référence).

Tableau 4.2 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test de transition LOC du nœud X_i, dans le cas d'une erreur de type BOE.

Test	Modèle	X_i	X_j	Y_l	$Y_{l'}$	Détection de l'erreur
Test 1	DUV	T	T	F(X_i= T)	F(X_j= T)	non car F(X_i= T)= F'(X_i= T) et F(X_j= T)= F'(X_j= T)
	Référence	T	T	F(X_i= T)	F(X_j= T)	
Test 2	DUV	T	T'	F(X_i= T)	F(X_j=T')	Oui car F(X_i= T)≠ F(X_i= T') et F(X_j= T)≠ F(X_j= T')
	Référence	T'	T	F(X_i=T')	F(X_j= T)	
Test 3	DUV	T	Cte	F(X_i= T)	F(X_i=cte)	Oui car F(X_i= T)≠ F(X_i= cte) et F(X_j= T)≠ F(X_j= cte)
	Référence	Cte	T	F(X_i= cte)	F(X_j= T)	

D'après le Tableau 4.2, un test de transition LOC ciblant les pannes de transition du nœud X_i ou du nœud X_j détecte l'erreur d'ordonnance des bus, sauf si le vecteur de test utilisé crée des

transitions similaires sur les deux nœuds inversés. Avec les tests 2 et 3, la détection est garantie. Cependant avec le test 1, elle ne l'est pas.

En conclusion, pour ce modèle d'erreurs la détection de l'erreur est possible mais non garantie, dépendamment de la nature du vecteur de test utilisé. En supposant que chaque type de test a la même probabilité d'être généré, la probabilité de détection de l'erreur est de 67%.

4.3.2 Le modèle BSE

Ce modèle représente une mauvaise connexion de l'entrée d'un module à une source donnée. Soit X le bus connecté à une source A à la place de la source B. Donc, dans le DUV erroné, X est connecté à *A* alors que dans le modèle de référence, X est connecté à B, tel que décrit dans la Figure 4.2.

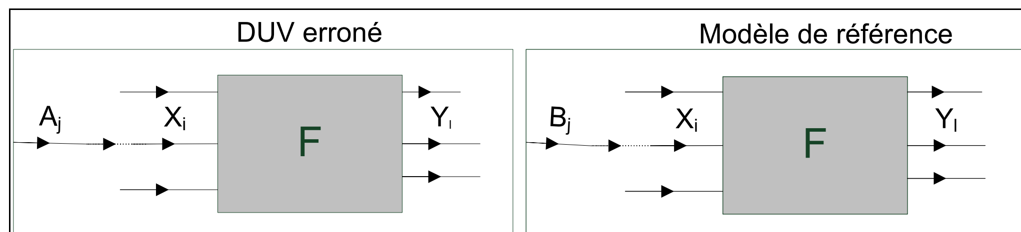


Figure 4.2 Exemple d'erreur de type BSE.

Les vecteurs de test de transition LOC (V1, V2) générés pour détecter les pannes de transition au niveau des différents bits A_j du bus *A* du DUV, créent des transitions T sur les nœuds A_j et propagent les transitions vers des nœuds observables Y_l.

Pour un test de transition STR (STF) du nœud A_j (le même raisonnement s'applique pour tous les bits de A), il existe 3 cas de figures possibles basés sur les comportements de A et de B :

- cas 1 : $\forall j$ A_j et B_j présentent le même comportement. Une transition T sur A_j crée la même transition sur B_j;
- cas 2 : $\exists j$ pour lequel une transition T sur A_j crée une transition opposée T' sur B_j;

- cas 3 : $\exists j$ pour lequel une transition T sur A_j crée une valeur constante de 0 ou de 1 sur B_j.

Le premier cas induit une erreur BSE redondante qui ne peut être détectée par aucun test. En considérant les cas 2 et 3 et en appliquant un vecteur de test de transition du nœud A_j, lors de la simulation du DUV et du modèle de référence, nous observons le comportement décrit dans le Tableau 4.3.

Tableau 4.3 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test de transition LOC du nœud A_i, dans le cas d'une erreur de type BSE.

Test	Modèles	A_i	B_i	X_i	Y_l	Détection de l'erreur
Cas 2	DUV	T	T'	T	$F(X_i=T)$	Oui car $F(X_i=T) \neq F(X_i=T')$
	Référence	T	T'	T'	$F(X_i=T')$	
Cas 3	DUV	T	cte	T	$F(X_i=T)$	Oui car $F(X_i=T) \neq F(X_i=cte)$
	Référence	T	cte	cte	$F(X_i=cte)$	

Donc les vecteurs de test de transition des différents bits du bus A permettent la détection des erreurs de type BSE lors de la simulation, en créant des comportements différents au niveau des deux modèles DUV et de référence.

En conclusion, pour ce modèle d'erreurs la détection de l'erreur est garantie si elle n'est pas redondante.

4.3.3 Le modèle BDE

Ce modèle représente la connexion d'un bus à deux sources différentes, tel que décrit dans la Figure 4.3. Cette erreur est détectable quelque soit le vecteur de test appliqué. En effet, lors de la simulation DUV, quelque soit le vecteur appliqué, le bus en question, relié à deux sources différentes aura toujours une valeur indéfinie, alors que dans le modèle de référence ce même signal aura une valeur bien spécifique, permettant la détection de l'erreur à tout moment de la simulation. Donc pour ce modèle d'erreurs, la détection est garantie.

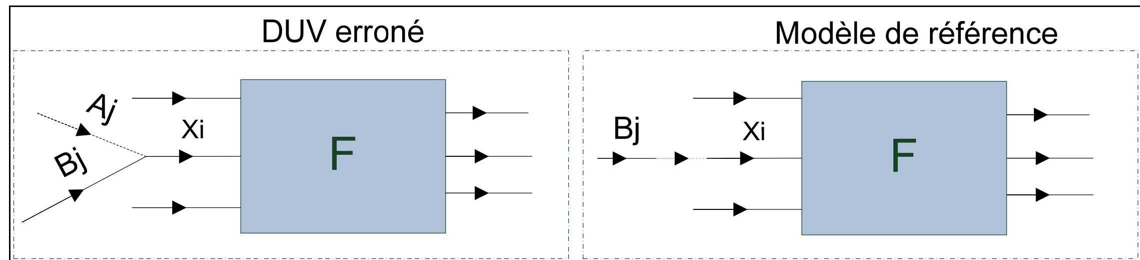


Figure 4.3 Exemple d'erreur de type BDE.

4.3.4 Le modèle MSE

Ce modèle représente les erreurs de remplacement d'un module M par un autre M' ayant le même nombre d'entrées et de sorties, tel que décrit dans la Figure 4.4. Lorsque le module M est complètement remplacé par un autre module M' dans le DUV, la détection de l'erreur est très probable étant donné la différence évidente de comportement entre le module de référence et le DUV pour la majorité des vecteurs de test. De plus, dans ce cas la synthèse du DUV crée un « *netlist* » différent de celui qui aurait été créé avec le modèle de référence, plus précisément le nombre et le nom des registres à balayage est différent et donc les vecteurs de test générés avec le DUV ne correspondent pas au modèle de référence et ils ne peuvent s'appliquer qu'au DUV. L'erreur sera détectée par le simulateur lors de l'application des vecteurs soulignant ainsi la différence entre les deux modules. La détection de ce modèle d'erreur est donc garantie.

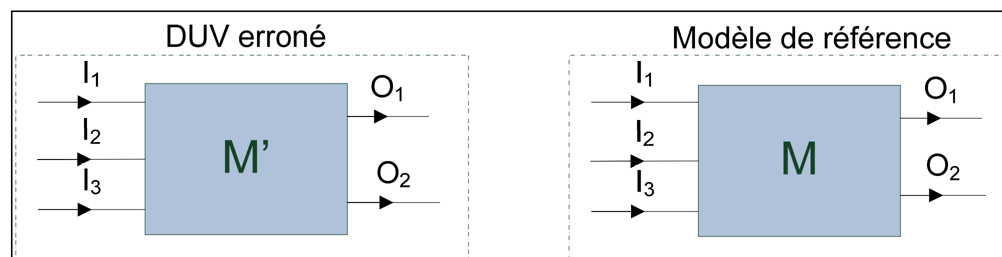


Figure 4.4 Exemple d'erreur de type MSE.

.

4.3.5 Le modèle BCE

Ce modèle représente l'ajout ou la suppression d'un bus d'entrée d'un module, tel que décrit à la Figure 4.5.

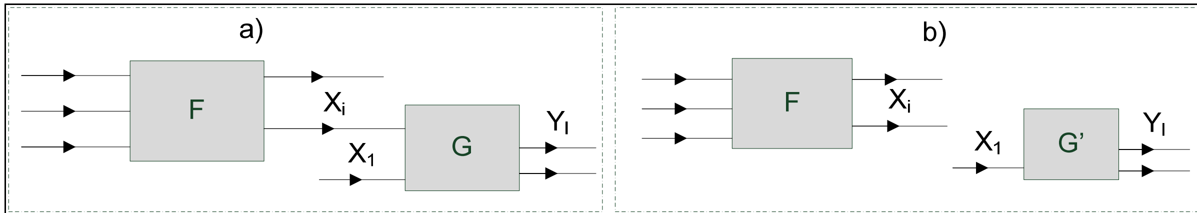


Figure 4.5 Exemple d'erreur de type BCE.

Donc il existe deux erreurs possibles correspondantes à ce modèle :

- **bus supplémentaire**: Supposons que le concepteur ait ajouté par inadvertance un bus supplémentaire. À la Figure 4.5, nous considérons alors que le circuit a) représente le DUV erroné contenant un bus supplémentaire X_i au niveau de l'entité G, et le circuit b) représente le modèle de référence :
  - X_i est un port de l'entité du module G. Si le bus supplémentaire n'a pas de source, l'erreur sera détectée en comptant le nombre d'entrées primaires du DUV. Si le bus n'a pas de destination, il sera détecté en comptant les sorties primaires;
  - X_i est un bus supplémentaire interne et G une porte logique ou une fonction logique interne au module. Dans ce cas, les vecteurs de test de transition LOC (V1, V2) générés pour détecter les pannes de transition au niveau des différents bits d'entrées de G plus spécifiquement l'entrée X_i, créent des transitions sur les différents nœuds X_i et les propagent vers des nœuds observables Y_l.

  Avec l'application des vecteurs de test, le comportement des deux modèles de DUV et de référence est décrit dans le Tableau 4.4.

Tableau 4.4 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test de transition LOC du nœud X_i, dans le cas d'une erreur de type BCE.

Modèles	$\mathbf{X_i}$	$\mathbf{Y_l}$	**Détection de l'erreur**
DUV	T	T	Oui
Modèle de référence	T	cte	

La différence de comportement au niveau Y_l permettra de détecter l'erreur du bus supplémentaire ;

- **Bus manquant**: supposons que le concepteur ait supprimé par inadvertance un bus. À la Figure 4.5 on considère alors que le circuit b) représente le DUV erroné avec un bus X_i manquant au niveau de l'entité G et le circuit a) représente le modèle de référence :
  - Si X_i est un port de l'entité du module G, le port manquant sera détecté en comptant le nombre d'entrées et de sorties primaires du DUV;
  - Si X_i est un bus manquant interne et G une porte logique ou une fonction logique interne au module. Dans ce cas, les vecteurs de test de transition générés pour les différents nœuds, créent des transitions et les propagent vers un nœud observable. Si pour un de ces vecteurs, le bus manquant X_i du modèle de référence a une valeur v pour laquelle : $G(X_0, X_1, \ldots) \left| X_i = v \neq G'(X_0, X_1, \ldots) \right.$, l'erreur est alors détectée.

  De plus, si le nombre de nœuds est grand il y a une probabilité accrue qu'au moins un des vecteurs de transition généré ait la valeur v souhaitée sur la ligne manquante.

En conclusion, la détection de l'erreur d'ajout d'un bus est garantie à l'aide des vecteurs de test de transition. Alors que dans le cas d'une suppression, la détection est garantie si le bus est un port d'entrées/sorties d'un module sinon la détection est forte probable.

4.3.6 Le modèle MCE

Ce modèle permet de détecter les erreurs d'ajout ou de suppression d'un module dans le design, incluant les portes logiques et les registres. La Figure 4.6 montre un exemple de ce type d'erreur. G représente une simple porte logique ou une entité formée de plusieurs portes

logiques et de plusieurs registres. Les vecteurs de test de transition LOC (V1, V2) générés pour détecter les pannes de transition au niveau des différents bits d'entrées de G, plus spécifiquement l'entrée A_i, crée une transition T au niveau du nœud A_i et la propage vers un nœud observables Y_l.

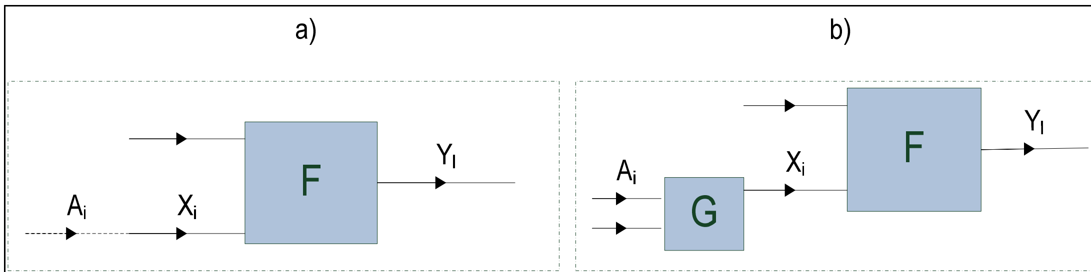


Figure 4.6 Exemple d'erreur de type MCE.

Pour ce modèle d'erreurs, on dispose des cas suivants :

- **porte logique supplémentaire/manquante** : il a été démontré dans (Abadir, 1988) que tout ensemble de vecteurs de test « *stuck-at* » permet de distinguer le circuit idéal du circuit erroné. Nous concluons qu'un ensemble de tests de transition générés pour le même ensemble de nœuds permettra de détecter ce type d'erreur au niveau logique et RT. En effet, le modèle de test de transition LOC est constitué de deux vecteurs V_1 et V_2 dont le second vecteur correspond au vecteur de test « *stuck-at* » de la valeur en question dépendamment s'il s'agit d'une panne STR ou STF. La valeur initiale pour une panne STR (STF) est 0 (l), et le vecteur final est un test « *stuck-at* » 0 (1). Alors un vecteur de test de transition contient le vecteur « *stuck-at* » du nœud correspondant et donc si un ensemble de vecteurs « *stuck-at* » pour un ensemble de nœuds donné est capable de détecter les erreurs sur ses nœuds alors les tests de transition correspondants détecteront les erreurs aussi;
- **module supplémentaire** : dans ce cas, le circuit b) représente le DUV erroné contenant le module manquant G avec A_i une entrée de G et X_i un bus reliant G au reste du design, et le circuit a) représente le modèle de référence où A_i et X_i sont reliés directement. Quand un module est ajouté par erreur, la synthèse crée des nœuds supplémentaires

n'existant pas dans le modèle de référence et donc les vecteurs de test générés ne peuvent être appliqués au niveau de la simulation, l'erreur est alors détectée par le simulateur;

- **module manquant** : dans ce cas, le circuit a) représente le DUV erroné avec un module G manquant, et le circuit b) représente le modèle de référence avec A_i une des entrées de G, et X_i un des bus connectant le module manquant au reste du design. Dans le DUV, A_i et X_i sont connectés directement ensemble. En appliquant le vecteur de test de transition généré pour le nœud A_i lors de la simulation le comportement observé est décrit dans le Tableau 4.5.

Tableau 4.5 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test de transition LOC du nœud A_i, dans le cas de la suppression d'un module dans le design.

Modèles	A_i	X_i	Y_l	**Détection de l'erreur**
Duv	T	T	$F(X_i=T)$	Oui si $G(A_i= T)\neq T$
Référence	T	$G(A_i=T)$	$F(X_i=G(A_i=T))$	

Si l'erreur n'est pas redondante, il $\exists i$ tel que $G(A_i=T) \neq T$ et la détection de l'erreur est donc garantie, d'après le Tableau 4.5.

En conclusion, la détection de l'erreur MCE est garantie lors de l'application des vecteurs de test de transition.

4.3.7 Le modèle LCE

Ce modèle représente l'erreur d'ajouter ou supprimer des étiquettes d'une structure de *case.* Un *case* est une structure présente dans la plupart des langages de programmation. Elle permet en général de faire un choix entre différentes exécutions possibles suivant la valeur d'un signal donné. Dans l'exemple présenté dans le Tableau 4.6, les structures *case* du DUV erroné et du modèle de référence dépendent respectivement des signaux *S* et *S'*. Dans le DUV l'étiquette correspondant à la valeur *v* de S est manquante.

Tableau 4.6 Exemple d'erreur de type LCE.

Modèle de reference : SystemC	DUV erroné : Case VHDL

Switch (S)
{
Cond1 : case value1: statement_1; break;
Cond2 :case value 2: statement_2; break;
Condv :case value v: statement_v; break;
}
	case (S')
	case value1 : statement1;
	case value2 : statement2;
	----------------------------------- 	
	default : case_item_statement;
	end case;

Soit *s* le nombre de bits de *S* et *s'* le nombre de bits de *S'*.On distingue les deux cas suivants :

- $s \neq s'$. Dans ce cas, l'ensemble des valeurs de tests de *S'* sont générées sur *s'* bits. Lors de la simulation RTL, l'application de ces valeurs sur le signal *S* du modèle de référence génère une erreur qui sera détectée par le simulateur. L'erreur est causée par la différence de nombre de bits entre la valeur forcée (*s'*) et le signal forcé (*s*). Donc, dans ce cas, la détection de l'erreur est garantie par le simulateur;
- $s = s'$. Les vecteurs de test de transition LOC (V1, V2) générés pour détecter les pannes de transition au niveau des différents bits de *S'*, créent des transitions sur tous les bits de *S'*, et donc n'importe quelle valeur contenue dans l'intervalle de définition de *S'* peut être forcée, y compris la valeur de *v*. Donc en simulant la valeur *v* au niveau des signaux *S* et *S'*, le modèle de référence présentera le comportement défini pour cette valeur dans sa structure de *case,* alors que le DUV dans lequel cette étiquette est manquante présentera le comportement défini par défaut et ainsi une erreur est détectée due à la différence des comportements des deux modèles.

Donc la détection de ce type d'erreur est possible si la valeur *v* de S est couverte. Afin de garantir cette détection, on peut s'assurer lors de la simulation que toutes les valeurs possibles des signaux faisant partie d'un *case* sont simulées. Les valeurs possibles des signaux peuvent être calculées basée sur leurs types respectifs. Ce processus est automatisé par un outil développé dans le cadre de cette thèse et qui sera présenté dans le chapitre suivant.

4.3.8 Le modèle ESE

Cette erreur inclut toutes les déviations possibles d'une expression logique correcte, tel l'ajout ou la suppression d'opérandes et d'opérateurs. Soient *F* et *F'* les fonctions modélisant les expressions dans le modèle de référence et dans le DUV respectivement. Tel que décrit à la Figure 4.7.

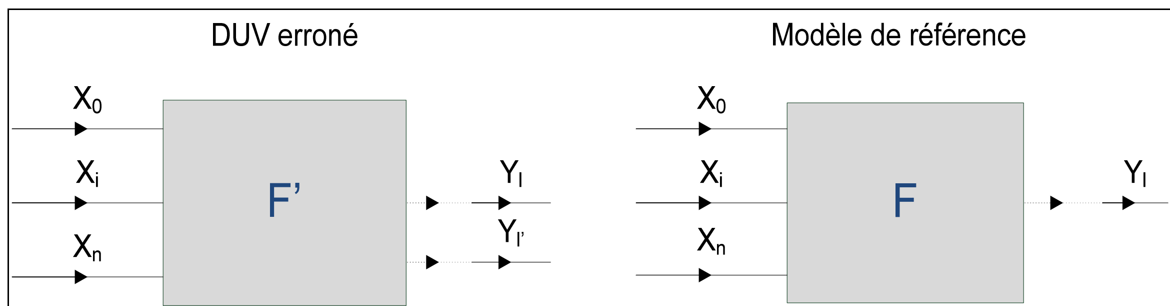


Figure 4.7 Exemple d'erreur de type ESE.

Les vecteurs de test de transition STF (STR) LOC (V1, V2) générés pour détecter les pannes de transition au niveau des différents bits X_i des entrées de F' créent des transitions au niveau des nœuds X_i et les propagent vers des nœuds observables Y_l.

En appliquant ces vecteurs lors de la simulation, des transitions sont créées sur les différents nœuds X_i des deux modèles mais se propageront différemment. En effet, si $F \neq F'$ alors il existe au moins un *i* tel que la transition T sur X_i propage vers Y_l pour F et $Y_{l'}$ pour F' avec $Y_l \neq Y_{l'}$.

Alors la détection des erreurs d'expression est garantie par les vecteurs de test de transition des différents nœuds d'entrées de l'expression en question.

4.3.9 Les erreurs FSM

Ce modèle représente les différentes erreurs rencontrées dans une machine à états. Elles se divisent en deux catégories :

- les erreurs d'états, SCE,
- les erreurs de transitions, NSE.

4.3.9.1 Le modèle SCE

Ce modèle représente le nombre incorrect d'états dans une FSM. Le concepteur peut insérer ou supprimer par erreur des états de la FSM.

Soient :

- M et M' les FSM respectives du modèle de référence et du DUV;
- N et N' le nombre d'états respectifs de M et de M';
- Les signaux *state* et *state'* modélisant respectivement les différents états de M et M';
- *b* et *b*' le nombre de bits respectifs de *state* et *state'*.

Dans le cas d'une erreur de type SCE, $N \neq N'$. La génération de vecteurs de test se fait en se basant sur le DUV et donc des vecteurs de transition sont générés pour couvrir les différents *b'* bits de *state'*.

Nous disposons de 2 types d'erreurs possibles:

- **états supplémentaires**: $N' > N$ et $b' \geq b$. Ce type d'erreur est représenté dans la Figure 4.8. Dans ce cas nous avons $state \subseteq state'$;

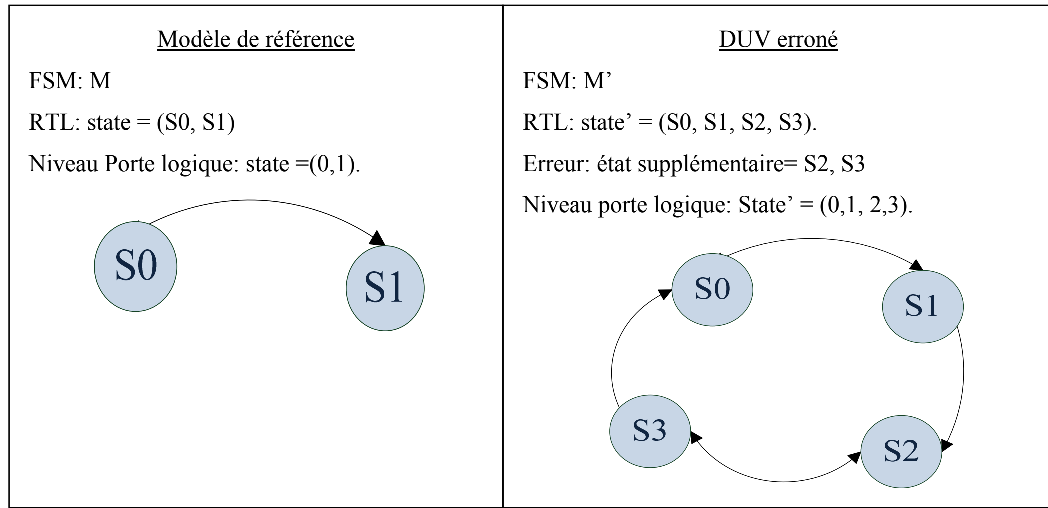


Figure 4.8 Exemple d'erreur de type NSE, états supplémentaires.

- **états manquants** : N'< N et b'≤b. Ce type d'erreur est représenté dans la Figure 4.9. Dans ce cas nous avons *state'* ⊆*state*.

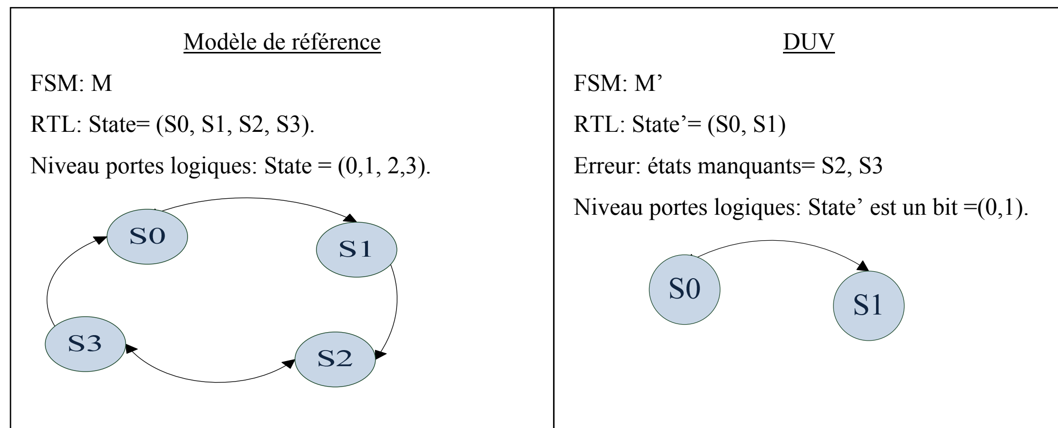


Figure 4.9 Exemple d'erreur de type NSE, états manquants.

Pour ces deux types d'erreur on distingue :

- le cas où b'≠ b. Dans ce cas, l'ensemble des valeurs de tests correspondants à *state'* sont générées sur *b'* bits. Lors de la simulation RTL, l'application de ces valeurs sur le signal *state* du modèle de référence génère une erreur qui sera détectée par le simulateur. L'erreur est causée par la différence de nombre de bits entre la valeur forcée (*b'*) et le signal forcé (*b*). Donc, dans ce cas, la détection de l'erreur est garantie par le simulateur;

- le cas où b'= b. Dans ce cas, les vecteurs de test de transition générés pour détecter les pannes de transition sur les différents bits du signal *state'*, forcent le signal à une des valeurs contenues dans son intervalle de définition. L'intervalle de définition de *state'* est le même que celui de *state*. Et donc, il est possible de couvrir les états supplémentaires et manquants avec les vecteurs de test générés. Si ces états sont couverts l'erreur est détectée à cause de la différence de comportements du DUV et du modèle de référence. Cependant, l'ensemble de vecteurs de test généré pour couvrir l'ensemble des transitions sur les différents bits *b'* de *state'* ne couvrent pas nécessairement tous les états du module. Dans l'exemple ci-dessus les vecteurs de tests contenant les états S0, S1 et S2 (00, 01, 10) sont suffisants pour couvrir les transitions sur les deux bits de *state'* et dans ce cas l'état *S3* ne sera jamais couvert et l'erreur correspondante ne sera pas détectée.
  Afin de garantir la détection de l'erreur il faut garantir la couverture de tous les états. Lors de la simulation, la couverture d'états est donnée par le simulateur. Basée sur cette information et sur l'ensemble des états légaux extrait par notre outil qui sera présenté dans le chapitre suivant, nous pouvons déduire l'ensemble des états non couverts. Ensuite, il s'agirait de générer les vecteurs de test permettant de couvrir ces états en appliquant sur l'outil ATPG des contraintes modélisant l'ensemble de ces états. Ainsi, nous disposons d'un ensemble des vecteurs de test permettant la couverture de tous les états de l'espace d'état. Dans ce cas, la détection de l'erreur est garantie. La mise en œuvre de cette méthodologie basée sur l'utilisation des contraintes fonctionnelles pour la couverture de l'ensemble des états fait partie des travaux futurs.

4.3.9.2 Le modèle NSE

Ce modèle représente les erreurs de transition dans une FSM. Soient :

- T= (S_i,S_j, I), une transition de l'état S_i à l'état S_j avec *I* l'ensemble des valeurs des entrées de la FSM;
- T'=(S_i,S'_j, I), la transition erronée correspondante avec $S'_j \neq S_j$;

- $Q = \{q; S_{iq} \neq S'_{jq}\}$ et $c = |Q|$, avec $q \in \{0...b\}$ b étant le nombre de bits modélisant l'ensemble des états et S_{iq}, S'_{jq} les qème bits respectifs de S_i et S'_j. Donc c représente le nombre de bits différés dans S_i et S'_j.

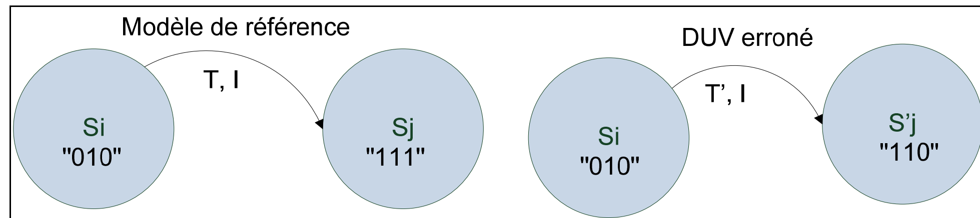


Figure 4.10 Exemple d'erreur de type NSE.

La Figure 4.10 présente un exemple d'une transition erronée où les états S_i et S'_j diffère d'un seul bit, donc $c=1$.

Durant la simulation, si le DUV est dans l'état S_i et les entrées sont forcées à I, il transite à l'état S'_j alors que dans les mêmes conditions le modèle de référence transitera à l'état S_j et l'erreur de transition sera détectée. Donc, afin qu'un vecteur de transition constitué d'une paire de vecteurs (V_1, V_2) puisse détecter ce type d'erreur, il va falloir dans un premier temps qu'il force le design à l'état S_i et ensuite le faire bousculer vers l'état S'_j en créant une transition sur l'un ou plusieurs des bits du signal d'état. En d'autre termes, afin de détecter l'erreur il faut couvrir la transition T'.

Étant donné que $S_i \neq S'_j$ alors $c > 0$, donc il existe au moins un bit tel que $S_{iq} \neq S'_{jq}$.
Le vecteur de test visant une panne de transition au niveau des bits S_{iq} avec $q \in Q$ crée une transition au niveau de S_{iq}, et ceci en forçant le design dans S_i et créant les conditions nécessaires pour le bousculer dans S'_j. Dans l'exemple de la Figure 4.10, un des vecteurs de test de transition visant le nœud S_{i2}, force le DUV à l'état S_i et les entrées à I et le DUV transitera vers l'état S'_j alors que le modèle de référence sous les mêmes conditions transitera vers S_j résultant en deux comportements différents aux 2 niveaux.

La détection des erreurs de transitions est donc possible par les vecteurs de test de transition couvrant les erreurs de transition sur les bits S_{iq} avec $q \in Q$. Plus le nombre de bits *c* différés entre les deux états S_i et S'_j est grand, plus la probabilité de couvrir la transition *T'* entre ces deux états est grande et plus la probabilité de détecter l'erreur est grande. Afin de garantir la détection de toute erreur de transition, il faut garantir la couverture de toutes les transitions possibles d'un design. Une possibilité serait de générer les vecteurs de test avec la méthode *N-detect* (Wang, Wu et Wen, 2006). En effet avec le *N*-detect, différents vecteurs permettant la détection d'une même panne de transition sont générés. Ainsi, en se basant sur la couverture de transitions générée par l'outil Modelsim suite à la simulation du design, on peut identifier les transitions non couvertes et utiliser la méthode *N-detect* pour générer un ensemble de vecteurs de test permettant la couverture de ces transitions. Dans ce cas, la détection de l'erreur est garantie. La mise en œuvre de cette méthodologie basée sur l'utilisation du *N-detect* fait partie des travaux futurs.

4.4 Test de transition versus *stuck-at*

Dans (Abadir, 1988), le modèle de panne « *stuck-at* » a été utilisé pour couvrir quelques types d'erreurs RTL. Les erreurs couvertes sont des erreurs de structure telles que l'ajout ou la suppression de portes logiques et la méthodologie a été développée pour les circuits combinatoires seulement.

Dans notre travail, nous avons opté pour le modèle de panne de transition qui s'avère plus efficace pour la détection de l'ensemble des modèles d'erreurs présentés (structurelles et comportementales). Par rapport aux vecteurs de test « *stuck-at* », les vecteurs de test de transition :

- permettent **une meilleure probabilité de détection de certains modèles d'erreurs** (BOE, BSE, LCE, BCE : bus supplémentaire). En effet, un test de transition permet de forcer sur le nœud en question les valeurs '0' et '1' et ceci durant le même test, augmentant ainsi l'ensemble des combinaisons de valeurs possibles appliqué sur les différents nœuds du design. Ceci induit une augmentation de la probabilité de détection

de certaines erreurs. Prenons le cas du modèle d'erreur BOE. L'ensemble des types de vecteurs de test « *stuck-at* » est décrit au Tableau 4.7, et le comportement correspondant est décrit au Tableau 4.8.

Tableau 4.7 Les différents tests possibles s@1 (s@0) du nœud X_i, basés sur les comportements de X_i et X_j.

Test	**X_i**	**X_j**	**Description**
Test 1	0 (1)	0 (1)	Le test crée les mêmes valeurs sur X_i et X_j
Test 2	0 (1)	1 (0)	Le test crée une valeur sur X_i et une valeur inverse sur X_j

Tableau 4.8 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test s@1 du nœud X_i, dans le cas d'une erreur de type BOE.

Test	**Modèle**	**X_i**	**X_j**	**Y_l**	**$Y_{l'}$**	**Détection de l'erreur**
Test 1	DUV	0	0	$F(X_i = 0)$	$F(X_j = 0)$	non car $F(X_i = 0) = F'(X_i = 0)$ et $F(X_j = 0) = F'(X_j = 0)$
	Référence	0	0	$F(X_i = 0)$	$F(X_j = 0)$	
Test 2	DUV	0	1	$F(X_i = 0)$	$F(X_j = 1)$	Oui car $F(X_i = 0) \neq F(X_i = 1)$ et $F(X_j = 0) \neq F(X_j = 1)$
	Référence	1	0	$F(X_i = 1)$	$F(X_j = 0)$	

Si on suppose que chaque type de test possède une probabilité égale pour être généré, la probabilité de couverture d'une erreur de type BOE par des tests s@ est de 50%. Alors que la probabilité de couverture de ce même type d'erreur avec les tests de transitions est de 67%. En effet, d'après le Tableau 4.2, nous voyons qu'en plus de ces deux cas de test (test 1 et test 2), les tests de transitions présentent un troisième cas (test 3) non couvert par les tests « *stuck-at* » et pour lequel la détection de l'erreur est garantie. Le test et le comportement correspondant sont présentés dans les tableaux Tableau 4.9 et Tableau 4.10.

Tableau 4.9 Le test de transition test 3.

Test	**X_i**	**X_j**	**Description**
Test 3	T	Constante (0 ou1)	Le test crée une transition T sur X_i et garde une valeur constante sur X_j

Tableau 4.10 Comportement des modèles de référence et du DUV lors de l'application des vecteurs de test de transition test 3, dans le cas d'une erreur de type BOE.

Test	**Modèle**	X_i	X_j	Y_l	$Y_{l'}$	**Détection de l'erreur**
Test 3	DUV	T	Cte	$F(X_i= T)$	$F(X_i=cte)$	Oui car $F(X_i= T) \neq F(X_i= cte)$ et $F(X_j= T) \neq F(X_j= cte)$

Le même raisonnement s'applique pour les modèles d'erreur BSE, LCE, et BCE. Leur probabilité de détection augmente avec les vecteurs de transition par rapport aux vecteurs « *stuck-at* » tel que décrit dans le Tableau 4.11. Dans le cas du modèle d'erreur BSE, la probabilité de détection passe également de 50% avec les tests « *stuck-at* » à 67% avec les tests de transition. Pour ce qui est du modèle d'erreur BCE, sa probabilité de détection est garantie avec les vecteurs de tests de transition alors qu'elle ne l'est pas avec les vecteurs de test « *stuck-at* » (i.e. égale à 50%);

Tableau 4.11 Différence entre les probabilités de détection d'erreurs des tests de transition et celle des tests « *stuck-at* ».

Modèle d'erreur	**Probabilité de détection**	
	Stuck-at	**transition**
BOE	50%	67%
BSE	50%	100%
BCE	50%	100%

- sont **plus adaptés aux comportements séquentiels** des circuits. Ils permettent la détection des erreurs de comportement séquentiel qui ne peuvent être détectées par le modèle « *stuck-at* ». En effet, bien que l'insertion des registres à balayage permette la transformation du comportement séquentiel du circuit en un comportement combinatoire, le fait de créer une transition sur un nœud donné et de la propager à travers le design permet à ce dernier de transiter d'un état à un autre, et ceci pour un même vecteur de test. Ce comportement permet de couvrir un modèle d'erreur que les tests « *stuck-at* » ne peuvent pas couvrir qui est le modèle d'erreur de transition;

- permettent de **couvrir plus d'erreurs avec moins de vecteurs de test**. Un vecteur de test de transition peut être considéré comme deux vecteurs de « *stuck-at* » opposé (s@0 et s@1). Par exemple, un vecteur de test de transition couvre deux états alors qu'un vecteur « *stuck-at* » ne couvre qu'un seul.

Ainsi les vecteurs de test de transition se distinguent vraiment des vecteurs de tests « *stuck-at* » dans la simulation au niveau RTL.

4.5 Les états illégaux

Nous avons montré que l'utilisation des tests de transition générés à partir des outils ATPG avait le potentiel de couvrir la plupart des modèles d'erreurs au niveau RT. Les tests utilisés sont basés sur la technique d'insertion des registres à balayage permettant de forcer une valeur quelconque au niveau d'un nœud interne du design. Malgré que cette technique permette une amélioration de la contrôlabilité, elle risque de conduire le design dans des états ne se produisant jamais en mode fonctionnel, appelé états illégaux. En effet, dans la majorité des cas, le design n'utilise pas son espace d'état complet créant ainsi des états illégaux.

Étant donné que nous utilisons le modèle LOC, les vecteurs générés peuvent inclure des valeurs initiales illégales pour lancer la transition. Les états de destination sont légaux par définition étant donné que les états de destination correspondent à la réponse du circuit aux vecteurs initiaux.

Dans ce qui suit, nous présentons en détails l'effet produit par l'utilisation des vecteurs de test contenant des états illégaux au niveau de la simulation.

4.5.1 Effet des états illégaux

Forcer un modèle à un état illégal peut provoquer des réactions différentes suivant le style de codage de ce dernier. On distingue les deux cas suivants :

- cas 1 : le comportement du module est spécifié pour tous les états possibles, légaux et illégaux. En forçant un état illégal, le module réagira en accord avec la description spécifiée dans le code correspondant à l'état en question;
- cas 2 : le comportement du module n'est spécifié que pour les états légaux. Alors, en forçant un état illégal, le module ne contenant aucune description relative à cet état met à jour le signal d'état en question mais garde les valeurs de l'état précédent pour le reste des signaux.

Lors de la vérification, deux modèles sont simulés : le DUV et le modèle de référence. Si le vecteur de test contient un état illégal, les deux modèles sont forcés dans cet état et réagiront suivant le style de code de chacun.

En effet, si les deux modèles ont le même style de code (cas 1 ou cas 2), ils réagiront de la même façon à l'application de l'état illégal. Présentant le même comportement aux deux niveaux, alors aucun problème ne se pose dans ce cas.

Mais si les deux modèles n'ont pas le même style de code, ils peuvent réagir différemment à l'application de l'état illégal. Le comportement non équivalent des deux modèles engendrera la détection d'une erreur qui n'existe pas en réalité. D'où le problème de la détection de fausses erreurs.

4.5.2 Exemple : compteur

Soit un compteur de deux bits, Il s'incrémente de 0 à 2 et possède un état illégal, celui qui correspond à la valeur 3. Il peut être codé d'au moins deux manières différentes :

- cas a : Le comportement du compteur est bien spécifié pour tous les cas possibles incluant les états légaux et illégaux;
- cas b : Le comportement du compteur est spécifié juste pour les états légaux.

Les codes VHDL correspondants aux deux cas sont présentés dans la Figure 4.11.

Cas a : **<u>Modèle DUV (VHDL)</u>**	**Cas b:** **<u>Modèle DUV (VHDL)</u>**

if reset='1' then
count_s<="00";
elsif (clk='1' and clk'event) then
--l'état 2 : remise à zéro
if (**count_s="10"**) then
count_s<="00";
--Tous les autres états incrémentation
else
count_s<=count_s+"01";
end if;
end if;
	if (reset='1') then
	count_s<="00";
	elsif (clk='1' and clk'event) then
	--l'état 2 : remise à zéro
	if(**count_s="10"**) then
	count_s<="00";
	-- les états 0 et 1 : incrémentation
	elsif (**count_s<"10"**) then
	count_s<= count_s+"01";
	end if;
	end if;

Figure 4.11 Exemple de deux codes VHDL différents d'un compteur.

Les machines à états correspondants aux deux cas a et b sont décrites dans la Figure 4.12.

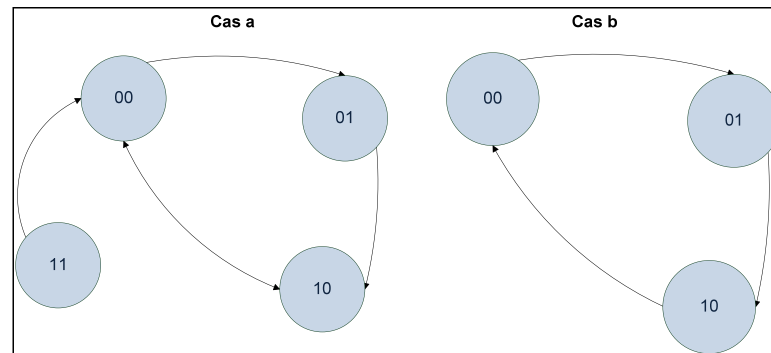


Figure 4.12 Les machines à états correspondants aux 2 styles de code cas a et cas b du compteur 2 bits présenté dans l'exemple ci-dessus.

Dans les deux cas, l'état 3 dans lequel count= ''11'' est un état inaccessible. Mais si pour une raison le signal count prend la valeur 11, dans le cas a le compteur s'incrémente et *count* prend la valeur « 00 » alors que dans le cas b le comportement n'est pas défini et donc ne peut être prédit. Les résultats de la simulation de ces modèles à l'aide des vecteurs de test de transition seront décrits dans ce qui suit.

En effet, après la synthèse du compteur, l'insertion des registres à balayage, la génération des vecteurs de test, à l'aide des outils ATPG, crée des vecteurs de test dont certains contiennent des valeurs illégales, tel que représenté dans la Figure 4.13.

ATPG patterns:

pattern = 0 -- chain "chain1" = "11"; **(État illégal)**
pattern = 1 -- chain "chain1" = "10"; **(État légal)**
pattern = 2 -- chain "chain1" = "11"; (**État illégal**)
pattern = 3 -- chain "chain1" = "01"; (**État légal**)
pattern = 4 -- chain "chain1" = "10"; (**État légal**)
pattern = 5 -- chain "chain1" = "01"; (**État légal**)
pattern = 6 -- chain "chain1" = "11"; (**État illégal**)
pattern = 7 -- chain "chain1" = "10"; (**État légal**)
pattern = 8 -- chain "chain1" = "01"; (**État légal**)

Figure 4.13 Vecteurs de test ATPG
correspondant à l'exemple de la Figure 4.11.

En appliquant ces vecteurs de test au niveau des deux modèles lors de la simulation, les valeurs illégales sont forcées et les modèles réagissent différemment suivant leur style de code :

- dans le premier cas (cas a) où le comportement est bien spécifié pour tous les cas valides et invalides, quand l'état illégal est forcé, le compteur s'incrémente pour prendre la valeur « 00 », comme le montre la Figure 4.14;

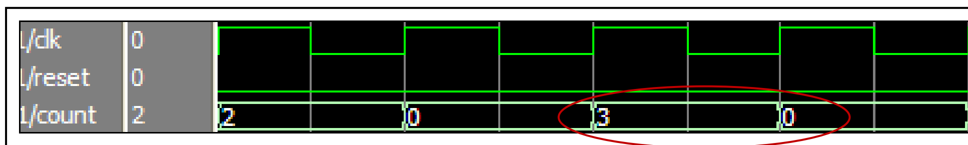


Figure 4.14 Résultat de la simulation du compteur (cas a) avec
les vecteurs de test de transition.

- dans le second cas (cas b) où le comportement est spécifié juste dans les cas valides, quand l'état illégal est forcé, le design ignore la valeur et le compteur s'incrémente par

rapport à la valeur qu'il avait dans le cycle précédent l'état illégal, comme le montre la Figure 4.15.

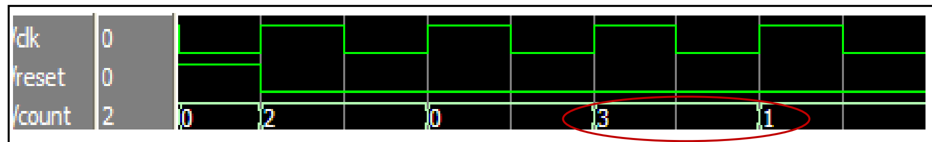


Figure 4.15 Résultat de la simulation du compteur (cas b) avec les vecteurs de test de transition.

Si nous considérons que les codes des cas a et b correspondent respectivement aux modèles DUV et de référence, nous pouvons voir que dans le cycle suivant l'application de l'état illégal, la valeur du signal *count* est différente dans les deux modèles et provoquera la détection d'une erreur qui n'existe pas en réalité.

4.5.3 Les fausses erreurs

D'après l'exemple présenté ci-dessus, nous pouvons conclure qu'en présence d'états illégaux, de fausses erreurs peuvent être détectées lors de la simulation. Ce problème nuit à la productivité de la méthodologie de vérification, et malgré le gain de couverture et de temps qu'apporte l'utilisation des vecteurs de test structurels lors de la simulation, la possibilité de détecter de fausses erreurs engendre une augmentation du temps et des efforts alloués à la vérification provoquant une diminution de la qualité de cette méthodologie. Donc, il sera essentiel de résoudre ce problème et d'éviter la génération des vecteurs de test illégaux.

Une solution à ce problème, basée sur l'extraction de contraintes fonctionnelles du design, est développée et sera présentée en détails dans le chapitre suivant. Cette méthodologie permet une génération automatique des contraintes fonctionnelles du DUV. Ces contraintes sont appliquées au générateur de vecteurs de test qui sera forcé à générer que des vecteurs légaux, appelés des vecteurs pseudo-fonctionnels.

4.6 Conclusion

Pour conclure, l'étude théorique effectuée dans ce chapitre montre que les vecteurs de test de transition LOC basés sur les registres à balayage et utilisés lors de la simulation au niveau RT, ont le potentiel de détecter de la plupart des erreurs RTL de conception. Le seul cas pour lequel la détection de l'erreur n'est pas possible est celui des états manquants d'une machine à états où le signal d'état du code erroné contient moins de bits que celui correspondant au circuit idéal. Par conséquent, nous en déduisons que l'utilisation de ces vecteurs au niveau de la vérification pourra produire de bonnes couvertures. De plus, ces vecteurs structurels sont générés automatiquement par les outils ATPG, donc aucun effort n'est requis pour leur production. En outre, l'émulation des registres à balayage traduite par le forçage des signaux internes à des valeurs données permet d'acquérir une meilleure couverture en peu de temps. Tous ces critères et avantages jumelés ensemble permettront une amélioration de la qualité de la vérification et une diminution du temps et de l'effort liés à cette dernière. Par contre, les vecteurs de test structurels peuvent contenir des états illégaux causant lors de la simulation des comportements aléatoires au niveau des deux modèles DUV et de référence, ce qui peut engendrer une détection des fausses erreurs. Ce type d'erreurs nuit énormément à la méthodologie proposée ajoutant du temps et des efforts non nécessaires à la vérification. Le chapitre suivant présentera une solution à ce problème et décrira en détails la méthodologie proposée pour éviter la création des vecteurs de test illégaux.

CHAPITRE 5

EXTRACTION AUTOMATIQUE DES CONTRAINTES FONCTIONNELLES À PARTIR D'UN MODÈLE RTL

5.1 Introduction

Le chapitre précédent a introduit un problème potentiel à l'utilisation des tests structurels dans la simulation d'un design au niveau RT. Rappelons que ces vecteurs de test sont générés par les outils ATPG au niveau porte logique, et leur utilisation durant la vérification permet d'émuler la présence des registres à balayage, en forçant les signaux internes du design. Ceci présente plusieurs avantages, toutefois un problème apparait lorsque les valeurs forcées font partie des états illégaux et ne peuvent être activées et propagées en mode fonctionnel. Le chapitre 4 souligne le problème causé par les états illégaux au niveau de la vérification, ce problème étant la détection des fausses erreurs. En effet, lorsqu'un état illégal, ne pouvant se produire en mode fonctionnel, est forcé durant la simulation, le DUV et le modèle de référence peuvent se comporter différemment, dépendamment de leurs styles de code respectifs, et donc induire une détection de fausses erreurs.

De plus, les états illégaux présentent une problématique au niveau du test. Dans plusieurs méthodes de génération de test, l'étape de justification arrière est critique et très souvent nécessite beaucoup de temps. Les valeurs sur les différents nœuds sont justifiées pour activer et/ou propager une faute. En présence des états illégaux, la justification arrière consomme encore plus de temps, étant donné que l'espace de recherche est plus large et contient des valeurs injustifiables. En conséquence, l'efficacité du générateur se dégrade et la complexité du test augmente (Konijnenburg, 1999). Les états illégaux ont également un impact sur la détection des fautes: en présence des états illégaux, plusieurs fautes ST-FU sont détectées, causant ce qu'on appelle un surtest et donc une perte de rendement potentiellement importante (Lin, 2005).

Ainsi, en contraignant les outils ATPG à générer des tests ne contenant que des états légaux, appelé des vecteurs de test pseudo-fonctionnels, on peut améliorer la qualité de la vérification et aussi l'efficacité du test.

Dans ce chapitre, nous présentons une nouvelle méthodologie d'extraction des contraintes fonctionnelles. Ces contraintes peuvent être appliquées directement sur les outils ATPG pour éviter la génération des états illégaux. Notre méthodologie est basée sur une nouvelle technique d'identification des états légaux au niveau RT. Le résultat est un outil automatique basé sur une analyse syntaxique et lexical du code VHDL. Afin de surmonter le problème de détection des fausses erreurs lors de la simulation, cet outil a été développé et utilisé dans l'environnement de vérification proposé dans cette thèse, et présenté en détails dans le chapitre suivant. De plus, les résultats expérimentaux montrent que l'outil peut être utilisé pour améliorer l'efficacité du test et réduire la perte de rendement. Notons que l'outil présenté peut être adapté facilement à n'importe quel autre langage matériel que le VHDL.

Le chapitre est divisé comme suit. La section 5.2 introduit la grammaire d'analyse d'expression développée pour le langage VHDL. La section 5.3 présente ensuite une vue globale sur la méthodologie proposée alors que la section 5.4 décrit les détails de l'implémentation de l'outil. Finalement, la section 5.5 présente les résultats expérimentaux montrant l'efficacité de l'approche proposée.

5.2 Grammaire d'analyse d'expression VHDL

La méthodologie proposée dans ce chapitre se base essentiellement sur l'analyse du code VHDL afin d'extraire les états légaux et les contraintes fonctionnelles du design. Cette analyse est basée sur une grammaire d'analyse d'expression (*PEG, parsing expression grammar*) que nous avons développée spécifiquement pour le langage VHDL, et qui sera présentée en détails dans cette section.

5.2.1 VHDL

Le VHDL est un langage matériel utilisé pour décrire les systèmes électroniques numériques (Bhaskar, 1998). Il permet une description de la structure d'un système, comment ce dernier est décomposé en sous-systèmes et comment ces sous-systèmes sont reliés entre eux (description structurelle). Il permet également de décrire la fonctionnalité d'un système en utilisant des formes familières de langage de programmation (description comportementale). Dans l'architecture, les déclarations concurrentes sont réservées aux processus et aux affectations des signaux. Les processus sont en outre constitués d'instructions séquentielles telles que les évaluations des expressions, les exécutions conditionnelles (*if-else, case,* etc.), les exécutions répétées (boucles). Un exemple de code VHDL est donné à la Figure 5.1.

5.2.2 Grammaire d'analyse d'expression

Une PEG est un formalisme permettant de définir la syntaxe d'un langage de programmation sous forme de règles. Tel que défini dans (Ford, 2004), une PEG est un 4-uplet G = (Σ, N, P; ED), où Σ est un ensemble fini de symboles terminaux (tels que les opérateurs d'un langage par exemple), N est un ensemble fini de symboles non-terminaux disjoint de Σ, les symboles non terminaux étant formés à partir de symboles terminaux et d'autres non-terminaux (tels que les instructions d'un langage par exemple), P est un ensemble de règles d'analyse et ED est une expression d'analyse appelée l'expression de départ. Chaque règle d'analyse de P a la forme $A \leftarrow e$, où A est un symbole non terminal et e est une expression d'analyse. Une expression d'analyse peut être un des symboles terminaux, un des symboles non-terminaux ou une combinaison d'expressions d'analyse. Les opérateurs de construction d'expressions d'analyse sont résumés au Tableau 5.1 (Ford, 2004) où e, $e1$ et $e2$ sont des expressions d'analyse.

Les expressions d'analyse sont définies dans (Ford, 2004) comme suit. Si e, $e1$ et $e2$ sont des expressions d'analyse, alors les expressions suivantes le sont aussi: 1) la chaîne vide; 2) tout symbole terminal; 3) tout symbole non terminal; 4) la séquence $e1e2$; 5) le choix e1 | e2 ; 6)

zéro ou plusieurs répétitions de *e*, *e**, 7) une ou plusieurs répétitions de *e*, *e*+; 8) la négation de *e, e!*.

Tableau 5.1 Opérateurs de construction d'expression d'analyse.
Tiré de (Ford, 2004)

Opérateur	**Description**
' '	Chaine de caractère
[ ]	Classe de caractère
.	Caractère quelconque
(*e*)	Groupement
e?	Optionnel
*e**	Zéro ou plusieurs répétitions de *e*
e+	Une ou plusieurs répétitions de *e*
!*e*	Négation de *e*
*e*1 *e*2	Séquence *e1e2*
*e*1|*e*2	Choix priorisé

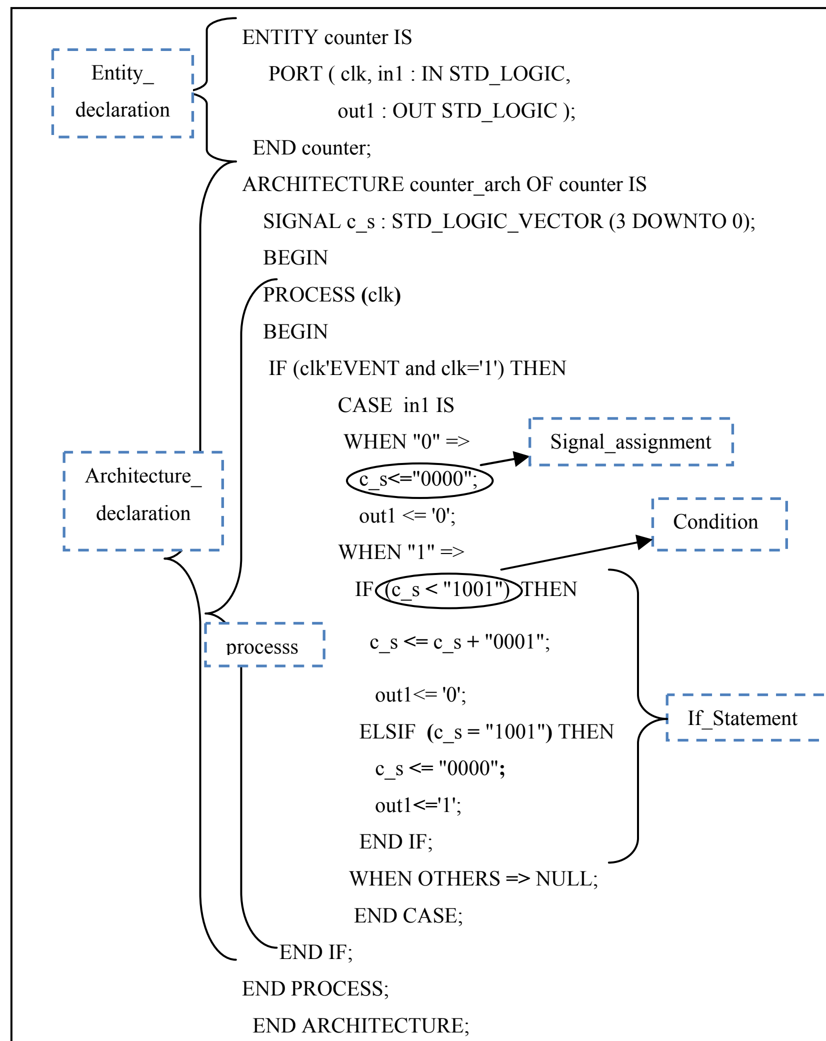


Figure 5.1 Exemple de code VHDL.

5.2.3 La PEG VHDL

Afin de permettre une analyse syntaxique et lexicale du code VHDL, une PEG adaptée au langage VHDL est nécessaire. Dans cette section, nous proposons une PEG VHDL qui couvre la plupart des structures VHDL. Sa définition se base sur la définition standard d’une PEG (Ford, 2004).

La PEG VHDL :

- Σ = [Mots-clés, symboles VHDL (<=, (,), etc.), les opérateurs (booléens et arithmétiques)]. À la figure 5.1, certains symboles non terminaux sont identifiés (les mots clés en caractères majuscules et les symboles et opérateurs en caractères gras);
- N = [module, entity_declaration, architecture_declaration, port_declaration, component_declaration, signal_type, component_instanciation, process, if_statement, case_statement, case_component, condition, signal_assignment, operation, VHDL_type, etiquette, valeurs];
- Es= module;
- L'ensemble des règles d'analyse *P* est défini au Tableau 5.2..

Tableau 5.2 L'ensemble des règles d'analyse VHDL.

Règle d'analyse	***E***	***A***
P1	entity_declaration architecture_declaration	Module
P2	'ENTITY IS' port_declaration 'END ENTITY;'	entity_ declaration
P3	'PORT (' (label ':' ('IN'| 'OUT') VHDL_type)+');'	port_declaration
P4	component_declaration* 'BEGIN' (component_instanciation*|Process*| signal_assignment*) 'END ARCHITECTURE;'	architecture_ declaration
P5	'COMPONENT IS' port_declaration 'END COMPONENT;'	component_ declaration
P6	'SIGNAL' label ':' VHDL_type';'	Signal_type
P7	label: label 'PORT MAP (' (label=> label)+ ');'	component_ instanciation.
P8	'BEGIN' (if_statement*| case_statement*| signal_assignment+) 'END PROCESS;'	Process
P9	(numerical value| label) (operator operation)?	Operation
P10	'IF(' condition ') THEN' (case_statement *| if_statement*| signal_assignment+) ('ELSE case_statement*| if_statement*| signal_assignment+)? 'END IF;'	if_statement

<table>
<tr><th>Règle d’analyse</th><th>E</th><th>A</th></tr>
<tr><td>P11</td><td>‘CASE’ label ‘IS’ (case_component)+ ‘END CASE;’</td><td>case_statement</td></tr>
<tr><td>P12</td><td>‘WHEN’ numerical_value => case_statement*| if_statement*| signal_assignment+‘;‘</td><td>case_component</td></tr>
<tr><td>P13</td><td>label (‘<’|’>’|‘≤’|’≥’|’=’) operation</td><td>condition</td></tr>
<tr><td>P14</td><td>label <= (numerical value| operation) ‘;’</td><td>signal_ assignment</td></tr>
<tr><td>P15</td><td>KEYWORDS+</td><td>VHDL_type</td></tr>
<tr><td>P16</td><td>[a-z]+</td><td>label</td></tr>
<tr><td>P17</td><td>[0-9]+</td><td>values</td></tr>
</table>

Cette grammaire est utilisée dans la méthodologie proposée afin d'analyser le code en fonction de ses structures. Ainsi, l’outil peut identifier les composants, les connexions des ports, l'affectation des signaux, les structures conditionnelles *case* et *if*, les opérations, les conditions ainsi que les chevauchements de conditions.

5.3 Approche proposée

Dans cette section, nous présentons la méthodologie proposée pour l'extraction des contraintes fonctionnelles et la génération des vecteurs de test pseudo-fonctionnels basés sur l'identification des états légaux. Ici, il est important de spécifier que nous considérons deux types d’états : l’état de haut niveau (HLS, « *High Level State* ») et l’état de bas niveau ou RTL (appelé simplement état dans le reste de la thèse) lié aux signaux d’état. Nous définissons un état de haut niveau comme l’ensemble des assignations des signaux d’état (RTL) liées à une condition particulière à l’intérieur d’un *process*. Ainsi, si pour une condition donnée, la valeur d’un compteur (signal d’état) est incrémentée, alors l’état de haut niveau contient toutes les valeurs possibles pouvant prendre ce compteur lors de ces assignations, alors que la valeur même de ce même compteur est considérée comme un état (RTL). De la même manière, les valeurs légales d’un signal d’état peuvent varier d’un état de haut niveau à l’autre.

Notons que les contraintes fonctionnelles peuvent être classées en 2 catégories : les contraintes temporelles et les contraintes logiques. Les contraintes temporelles sont utiles pour la synthèse et l'optimisation, tandis que les contraintes logiques sont utilisées dans la vérification et le test. Les contraintes logiques peuvent être temporelles en spécifiant les conditions sur les nœuds à travers plusieurs cycles d'horloge, ou spatiales en précisant les conditions non-temporelles qui doivent être satisfaites dans chacun des cycles d'horloge. Dans notre travail, nous nous intéressons à la vérification et au test, plus précisément l'identification des HLS légaux du design. De ce fait, nous nous limitons aux contraintes logiques spatiales.

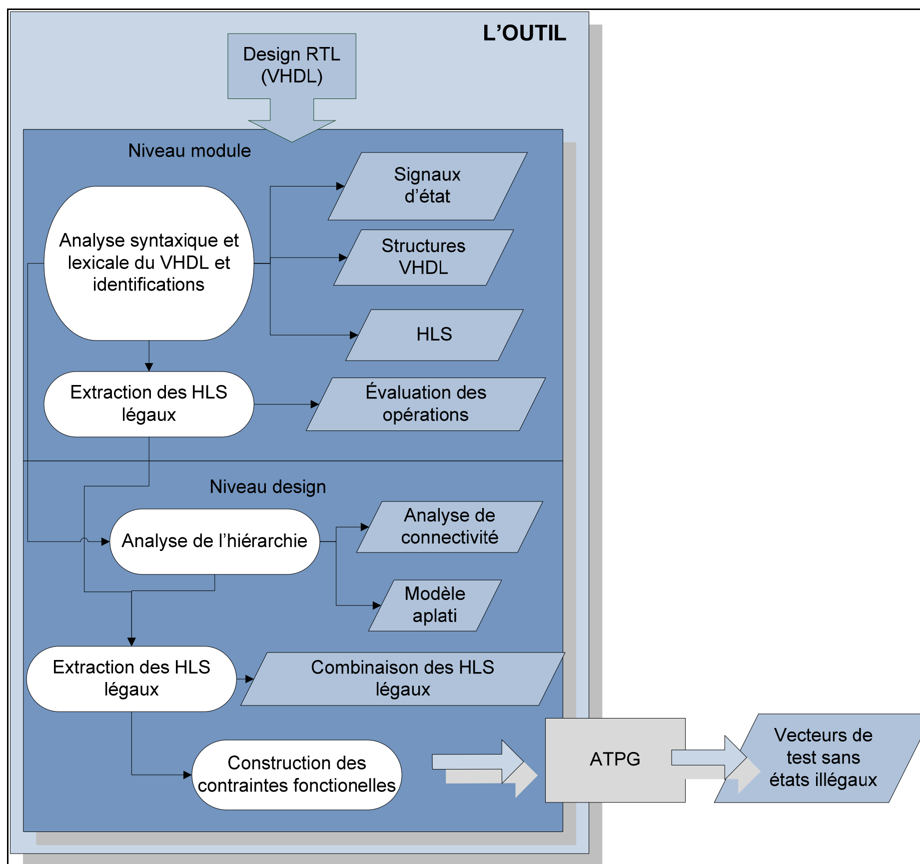


Figure 5.2 Extraction des contraintes fonctionnelles d'un design VHDL.

La Figure 5.2 illustre les différentes étapes impliquées dans cette démarche. Tout le travail s'effectue au niveau RT, basé sur une description VHDL du design. L'outil effectue plusieurs

étapes. D'abord, il commence par une analyse lexicale et syntaxique du code VHDL de chacun des modules du design. Cette étape permet d'extraire les informations utiles à différents niveaux de l'approche proposée. Ensuite, toujours au niveau du module, toutes les opérations identifiées sont évaluées et les HLS légaux sont construits. Au niveau design, une analyse de connectivité est effectuée afin de créer un modèle aplati du design et déduire toutes les interconnections et dépendances des modules. L'outil construit les HLS légaux du design en combinant tous les HLS légaux de chacun des modules tout en respectant les connexions inter-modulaires.

Enfin, à partir de l'ensemble des HLS légaux du design, la liste des contraintes fonctionnelles est créée et sera appliquée aux outils ATPG pour générer des tests pseudo-fonctionnels.
L'outil est automatisé et le code est écrit en Perl. L'implémentation de ce dernier est décrite en détails dans la section suivante.

5.4 Implémentation détaillée

La procédure d'extraction des contraintes fonctionnelles peut être divisée en 5 étapes de base:

- analyse du VHDL et identification des différentes structures ;
- extraction des HLS légaux des modules constituant le design ;
- analyse hiérarchique du design ;
- extraction des HLS légaux du design ;
- extraction des contraintes fonctionnelles.

Nous décrirons en détails chaque étape de la procédure en l'appliquant sur l'exemple VHDL présenté dans la Figure 5.1. De plus, nous présenterons les différents algorithmes et fonctions utilisées dans l'implémentation.

Dans ce qui suit, on considère :

- un design composé de M différents modules et I différentes instances $I_{id_instance}$ de ces modules avec $id_module \in \{1..M\}$ et $id_instance \in \{1..I\}$;
- un module avec N différents signaux d'état $x_{\text{id_signal}}$, et S différents HLS $S_{id_hlstate}$. $id_signal \in \{1...N\}$ et $id_hlstate \in \{1..S\}$.

5.4.1 Analyse du VHDL et identification des différentes structures

La première étape de la méthodologie consiste en une analyse syntaxique et lexicale du code, basée sur laquelle des informations utiles à l'extraction des HLS légaux (telles que les affectations de signaux, les processus et les dépendances de données, les dépendances inter-modulaires ainsi que la hiérarchie du design) sont identifiées et stockées. Toutes les analyses et la collecte de données se font directement sur le code VHDL : aucun modèle intermédiaire n'est nécessaire pour extraire les informations.

Ainsi, sur la base de la PEG du VHDL proposée dans la section 5.2.3, l'outil effectue une analyse lexicale et syntaxique des différentes structures du code pour chaque module du design. Il analyse le code VHDL ligne par ligne. Chaque ligne VHDL est transformée en un ensemble de jetons. Chaque jeton est une séquence de caractères représentant un symbole terminal ou non-terminal, tel qu'un identificateur, un opérateur, etc. Ainsi, basé sur l'ensemble des jetons et l'ensemble des règles d'analyse P de la PEG VHDL, chaque structure est identifiée dans son contexte et les données correspondantes ainsi que les dépendances sont extraites et sont stockées dans la représentation correspondante à chaque structure. Ces données seront utilisées à différents niveaux de la méthodologie.

En plus de l'identification des structures, l'ensemble des valeurs initiales des signaux d'états ainsi que l'ensemble des différents HLS du module sont identifiés durant cette étape. Les procédures respectives utilisées pour identifier et construire ces deux ensembles sont : *initial_compute* et *high_level_state_identification*.

La Figure 5.3 décrit en détails la première étape d'analyse du code VHDL en présentant l'identification des différentes structures VHDL ainsi que l'appel des procédures durant le

processus. Notre outil couvre la majorité des structures de base VHDL (tel que mentionné précédemment, il peut également être adapté à d'autres langages). Nous décrirons dans ce qui suit, l'identification des différentes structures VHDL, ainsi que les deux procédures *initial_compute* et *high_level_state_identification* permettant l'identification des valeurs initiales des signaux d'états et les différents HLS du module.

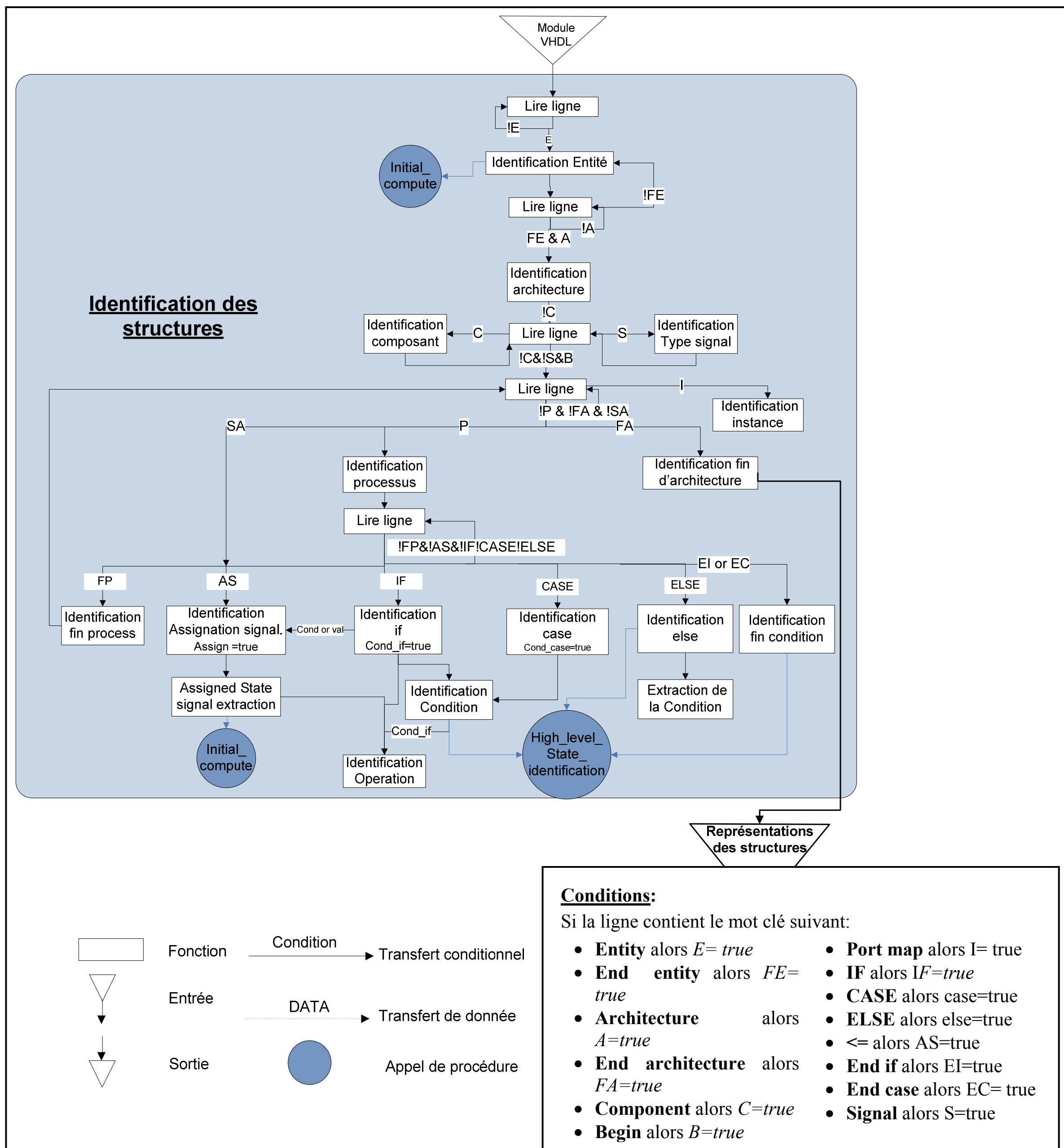


Figure 5.3 Identification des structures VHDL d'un module.

5.4.1.1 Identification des structures VHDL

Cette partie décrit l'identification des différentes structures VHDL ainsi que leurs représentations correspondantes. Le Tableau 5.3 décrit ce processus. Pour chaque structure identifiable, le tableau indique les règles utilisées pour l'identification et présente un lien vers la représentation correspondante construite suite à l'identification ainsi que le type d'information modélisée dans cette dernière. Les formats résultants sont décrits à la Figure 5.4.

Tableau 5.3 Identification des différentes structures VHDL.

Structure identifiée	**Règles d'analyse**	**Représen-tation**	**Information modélisée dans la représentation**
Entité	P1, P2, P3	Figure 5.4.a	Interface et ports du module
Composant	P4, P5	Figure 5.4.b	Interface et ports du composant
Type signal	P6	Figure 5.4.c	Types des signaux internes du composant
Instance	P4, P7	Figure 5.4.d	*Mapping* du composant instancié
Affectation de signal	P4, P14	Figure 5.4.e	Le signal sous contraintes, les opérations, et le *if* ou *case* correspondants si il existe (sinon *id=0*)
Structure case	P8, P11, P12	Figure 5.4.f	Le signal sous contraintes et les valeurs correspondantes
Structure if	P8, P10	Figure 5.4.g	condition
Condition	P13	Figure 5.4.h	Le signal contraint et l'opération modélisant la condition
Opération	(P13 ou P14), P9	Figure 5.4.i	Opérandes et opérateurs

D'après le Tableau 5.3, l'outil permet d'identifier, basé sur différentes règles d'analyse, les structures VHDL suivantes :

- l'**entité :** dans la représentation de l'entité (Figure 5.4.a), le champ signal est assigné si une assignation du signal à l'un des ports est détectée;
- les C**omposants**;
- les **types de signaux**;

- les **instances**;
- les structures ***if***;
- les **conditions (COND)**;
- les structures ***else***. Elles sont extraites en inversant la condition du *if* correspondant;
- les structures ***case***;
- l'**affectation d'un signal (SA)**. Dans ce cas, les signaux connectés aux ports de l'entité sont déduits, basé sur cette représentation, et donc la représentation de l'entité est mise à jour en ajoutant l'identificateur du signal connecté aux différents ports, s'il y a lieu;
- l'**opération**. Sa représentation contient l'identificateur de l'opération, un attribut (COND / SA) indiquant si l'opération correspond à une affectation de signal ou à une condition, les opérateurs et opérandes correspondants et enfin un identificateur de HLS *hlstate_id* qui relie les résultats de l'opération au HLS correspondant du module; les opérateurs peuvent être arithmétiques (+, -, *, /) ou logique (AND, OR, NOT, NAND, NOR, XOR, XNOR); le calcul est décrit en détails un peu plus loin.

En outre, l'identification de l'**architecture**, **process**, **fin de l'architecture**, **fin du process**, **fin de la condition,** consiste à identifier les mots clés correspondants : *architecture, process, end architecture, end process, end if* et *end case*. Aucune transformation n'est faite dans ce cas.

a)

Entity	Id_port	Port_name	direction	Signal
counter	1	Clk	in	n/a
counter	2	In1	In	n/a
counter	3	Out1	Out	n/a

b)

Id_component	Id_port	direction

c)

Signal	Type
In1	Std_logic
Out1	Std_logic
C_s	Std_logic _vector (3 downto 0)

d)

Id_instance	Id_component	Id_port	Connected to

e)

Id_sa	Signal	Id_cond	Id_op	Expression
1	C_s	1, 2	3	C_s<=0000
2	Out1	1, 2	4	Out1<=0
3	C_s	3, 4	7	C_s<=c_s+1
4	Out1	3, 4	8	Out1<=0
5	C_s	3, 5, 6	11	C_s<=0000
6	Out1	3, 5, 6	12	Out1<=1

f)

Id_case	Id_cond	Condition
1	2	In1= 0
1	3	In1= 1
1	7	others

g)

Id_if	Id_cond	Condition
1	1	Clk= 1 and clk'event
2	4	C_s<1001
2	5	C_s≥1001
3	6	C_s=1001

h)

Id_cond	Id_op	signal	operator
1	1	clk	=
2	2	In1	=
3	5	In1	=
4	6	C_s	<
5	9	C_s	≥
6	10	C_s	=
7	n/a	n/a	null

i)

Id_op	cond | Sa	operation	operand	Id_hlstate
1	Cond	Clk= 1 and clk'event	clk	1,2,3,4,5
2	Cond	0	In1	1
3	Sa	0000	C_s	1
4	Sa	0	Out1	1
5	Cond	1	In1	2, 3, 4, 5
6	Cond	1001	C_s	3
7	Sa	C_s + 1	C_s	3
8	Sa	0	Out1	3
9	Cond	1001	C_s	4
10	Cond	1001	C_s	5
11	Sa	0000	C_s	5
12	Sa	1	Out1	5

Figure 5.4 Les représentations de a) entité b) composant c) signal_type d) instance e) affectation de signal f) structure if g) structure case g) condition i) opération.

5.4.1.2 Identification des valeurs initiales des différents signaux d'états

Nous avons besoin de construire la liste des différents signaux d'état du module incluant les ports d'entrées et de sorties avec les intervalles des valeurs initiales correspondantes. Tel que montré à la Figure 5.3, pendant l'analyse du code VHDL, nous faisons appel à la procédure *initial_compute*, lorsque les ports de l'entité ou les différents signaux internes sont identifiés. La procédure calcule l'intervalle de valeurs (valeurs min et max), associé à chaque port ou signal, et ceci en se basant sur leurs types respectifs (Figure 5.4.c). Ensuite, elle stocke l'ensemble des valeurs calculées tel qu'indiqué au Tableau 5.4.

L'ensemble des valeurs initiales des signaux d'états LV_0 est ainsi calculé et peut être modélisé comme suit :

$$LV_0=\left\{\bigcup_{j=0}^{N}\left[x_{j\min}\ ,x_{j\max}\right]\right\}\text{; avec } x_j\ j\in\{0..N\}. \qquad (5.1)$$

Basé sur l'exemple de la Figure 5.1:

$$LV_0=\{[0,1],[0,15],[0,1]\}\text{ ;}$$

Tableau 5.4 Valeurs initiales des signaux d'état basées sur l'exemple de la Figure 5.1.

Signal	Min	Max
In1	0	1
C_s	0	15
Out1	0	1

5.4.1.3 Identification des HLS

Tel que mentionné précédemment, un HLS d'un module est défini comme l'ensemble des valeurs des signaux d'état pouvant apparaitre en même temps dans un module. Ceci correspond aux valeurs extraites des assignations ayant lieu sous les mêmes contraintes, en plus des contraintes elles-mêmes. En d'autres termes, toutes les assignations soumises aux

mêmes conditions correspondent au même HLS ainsi que les conditions en question. Alors, du moment où une nouvelle condition apparaît ou une condition existante est inactivée, nous faisons appel à la procédure *high_level_state_identification* pour mettre à jour l'identificateur du HLS courant, tel que montré à la Figure 5.3. Chaque HLS possède un identificateur (*id_hlstate*) qui apparait dans la représentation des opérations (Figure 5.4.i).

Notons que la condition responsable de la synchronisation des assignations avec l'horloge (if clk'event and clk=1) ne définit pas un état et donc n'est pas considérée dans l'identification des nouveaux HLS.

L'identification des états HLS est effectuée par la procédure *high_level_state_identification* décrite à la Figure 5.5.

```
Procédure: high_level_state_identification
Id_hlstate=0;
State_count= 0;
Active_cond= {};

Loop: If (Condition identification) then
//incrémenter le nombre de HLS
          hlState_count ++
//ajouter l'identificateur de la condition nouvellement identifiée à la liste
//des conditions actives
          Active_cond = Active_cond ∪ id_cond;
          Id_hlstate= hlstate_count;
//insérer le nouvel HLS dans la table des HLS avec son identificateur
// et ses conditions correspondantes
          Insert (HLS_table, id_hlstate, active_cond);
          Elsif (end condition identification) then
//supprimer la condition correspondante de la liste des conditions actives
          Active_cond = Active_cond - id_cond;
//mettre à jour le HLS à celui ayant des conditions //correspondants à la liste
des conditions active.
          Id_hlstate= id_hlstate(HLS_table(active_cond = active_cond))
          Jump loop;
End if;
```

Figure 5.5 Détails de l'implémentation de la procédure *high_level_state_identification.*

Chaque assignation de signal est affectée à un HLS bien spécifique. En effet, lorsqu'une assignation de signal est identifiée, l'*id_hlstate* courant lui est affecté et est donc stocké dans sa représentation correspondante. Dans le cas d'une condition, elle peut être affectée à plusieurs HLS différents. Ainsi, afin d'identifier la liste de conditions pour chacun des HLS, nous gardons une trace de l'ensemble des conditions activées et lorsqu'un nouvel HLS est identifié la liste des conditions correspondantes est identifiée aussi, et l'ensemble est stocké comme le montre le Tableau 5.5.

Tableau 5.5 HLS correspondants à l'exemple de la Figure 5.1.

Id_hlstate	**Active_cond**
1	1, 2
2	1, 3
3	1, 3, 4
4	1, 3, 5
5	1, 3, 5, 6

5.4.2 Extraction des HLS des module

La deuxième étape consiste à extraire l'ensemble des HLS de chacun des modules composant le design, et ceci en se basant sur les informations extraites dans l'étape précédente. En effet, les valeurs des différents signaux correspondants aux différents HLS du module sont calculées suite à une évaluation de l'ensemble des opérations identifiées dans le module et stockées (Figure 5.4.i). Dans ce qui suit, nous allons décrire le processus d'évaluation des opérations ainsi que les implémentations des procédures utilisées pour extraire l'ensemble des HLS d'un module.

5.4.2.1 Évaluation des opérations

Une opération VHDL consiste en un ensemble d'opérateurs arithmétiques ou logiques dont les entrées peuvent être des signaux, variables ou des constantes.

Une opération sur les signaux x_j peut être modélisée sous la forme d'une fonction *f* comme suit:

$$f\,(a_0\,x_0,\ldots,a_j x_j,\ldots,a_N\,x_N)\ \text{avec}\ j\in\{0..N\}\ \text{et}\ a_j\in\{0,1\};$$

- la fonction *f* correspond à l'ensemble des opérateurs modélisant l'opération;
- les entrées de la fonction *f* correspondent aux opérandes;
- le domaine *D* de la fonction correspond aux intervalles de valeurs de l'ensemble des opérandes;
- l'intervalle *R* des valeurs de *f* correspond à l'ensemble des résultats possibles de l'opération.

Un exemple d'une opération se trouvant dans une affectation de signal : *c_s <= c_s + 1;* avec :

- la fonction $f = +$;
- les entrées sont: x_0= c_s and x_1= 1;
- le Domaine (*D*) est
  D= $\left\{\left[c_s_{\min}\ ,c_s_{\max}\right],[1,1]\right\}$;
- l'intervalle *R* de *f*, est le résultat de l'évaluation de l'opération:
  R(c_s)= $\left[c_s_{\min}\ +1, c_s_{\max}\ +1\right]$.

Ainsi, afin de calculer les résultats d'une opération, il faut d'abord identifier la fonction *f*, ses entrées et son domaine *D*, puis calculer les résultats avec les valeurs correspondantes des opérandes. La fonction *f* ainsi que ses entrées peuvent être identifiées à partir de la représentation des opérations (Figure 5.4.i), alors que le domaine *D* doit être calculé et construit progressivement avec les différentes identifications des différentes structures dans le code. En effet, *D* prend initialement les valeurs de l'ensemble LV_0, calculé selon le type de chacun des signaux d'états. Ensuite, *D* est mis à jour à chaque ajout ou suppression d'une contrainte et donc à chaque évaluation d'une condition.

La fonction EVAL permettant l'évaluation d'une opération donnée a les caractéristiques suivantes :

- entrées: opération et son domaine *D* (l'ensemble des valeurs valides de x_j) ;
- sortie: l'intervalle R des résultats de l'évaluation.

Évaluation des opérations se trouvant dans des conditions ou des affectations de signaux

Le processus d'évaluation d'opération que nous venons juste de décrire peut se faire dans le contexte d'une opération ou d'une affectation de signal. Dans les deux cas, l'évaluation se fait en se basant sur le même concept mais les résultats de l'évaluation sont utilisés différemment.

En effet, chaque HLS $S_{id_hlstate}$ identifié dans l'étape précédente possède un ensemble d'opérations qui lui sont assignées. Ces opérations peuvent être regroupées en 2 catégories :

- **opérations identifiées dans des conditions** : Pour un HLS donné, l'ensemble de ses conditions représente les contraintes des différents signaux correspondants à ce HLS. Et donc, le résultat de l'évaluation des opérations contenues dans ses conditions définit le domaine du HLS (D_{id_hlsate}).

  Une opération dans une condition peut être décrite de la manière suivante :

  *If (signal1 cond_opérateur **operation1**)*

  L'opérateur *cond_opérateur* représente n'importe quel opérateur de comparaison (=, < , ≥, ≤, >). Basé sur l'opérateur *cond_opérateur* et le résultat de l'évaluation de l'opération de la condition *EVAL(opération1),* nous déduisons l'intervalle des valeurs du signal sous contrainte *signal1*. Ce processus est réalisé à l'aide la fonction EVAL_COND dont les caractéristiques sont les suivantes :

  - entrées: l'opérateur de comparaison *cond_operator* et le résultat de l'évaluation des opérations *EVAL (opération)* ;
  - sortie: l'intervalle des valeurs résultant *R*.

  Ainsi, les valeurs minimum et maximum de l'intervalle *R* calculées par EVAL_COND, représentent une partie des contraintes des signaux d'états du HLS correspondant. Ces

valeurs sont alors stockées dans un tableau, comme le montre le Tableau 5.6. Ce tableau est utilisé pour identifier les domaines des différents HLS $D_{id_hlstate}$.

$$D_{id_hlstate}=\left\{\bigcup_{j=0}^{k}\left[x_{j\min}\ (id_state), x_{j\max}(id_state)\right]\right\}$$

Avec k $\in [0, N]$, où k est le nombre de signaux avec contraintes correspondants au HLS $S_{id_hlstate}$.

Tableau 5.6 Contraintes des signaux d'états.

Id_op	Id_hlstate	Signal identifier	Min	max	I|IS|O

- **opérations identifiées dans les affectations de signaux** : Dans un HLS donné, ces opérations doivent être évaluées en fonction des contraintes des signaux d'état correspondants et donc en fonction du domaine du HLS correspondant $D_{id_hlstate}$. L'évaluation de ces opérations est effectuée à l'aide de la fonction EVAL avec D = $D_{id_hlState}$ et les résultats obtenus correspondent à l'ensemble des valeurs valides des différents signaux correspondant au HLS $S_{id_hlstate}$. Les valeurs sont stockées dans un tableau, comme le montre le Tableau 5.7.

Tableau 5.7 Affectation des signaux d'états.

Id_op	Id_hlstate	Signal identifier	Min	Max	I|IS|O

Ainsi, pour chaque HLS S_{id_hlsate}, les conditions sont évaluées en premier afin de calculer le domaine du HLS $D_{id_hlstate}$, ceci en évaluant les opérations correspondants aux conditions assignés au HLS S_{id_hlsate}. Ensuite, le domaine calculé est utilisé pour évaluer les opérations qui se trouvent dans les différentes affectations des signaux assignés au HLS S_{id_hlsate}. Les valeurs obtenues sont stockées dans des tableaux (ex. tableaux 5.6 et 5.7), et seront utilisées pour extraire l'ensemble des HLS des différentes instances des modules composants le design. L'ensemble du processus est automatisé et l'implémentation détaillée est décrite dans ce qui suit.

5.4.2.2 Implémentation détaillée

L'extraction des HLS des différents modules est automatisée à l'aide de la procédure COMPUTE. La procédure évalue chacune des opérations identifiée lors de l'analyse syntaxique du module et effectue les mises à jour des tableaux de contraintes des signaux d'état (Tableau 5.6) et d'affectation des signaux d'état (Tableau 5.7). Et le résultat de la procédure est l'ensemble des HLS du module Q_{id_module} ainsi que la table de ces HLS (Tableau 5.8).

Notons que l'implémentation proposée est limitée aux designs VHDL à un seul domaine d'horloge. De plus les structures VHDL couvertes sont limitées à l'ensemble des structures supporté par la PEG VHDL présentée dans la section 5.2.3.

Les caractéristiques de la procédure principale COMPUTE sont les suivantes :

- entrées :
  - les représentations des affectations des signaux, conditions, opérations du module. (Figure 5.4.e, 5.4.h, 5.4.i) ;
  - la liste des valeurs initiales des signaux d'états LV_0.

- sorties :

  - l'ensemble des HLS du module Q_{id_module}

    $Q_{id_module} = \{S_0, .., S_{id_hlstate}, .., S_S\}$ avec

    $S_{id_hlstate} = \bigcup_{j=o}^{N}$ intervalle des valeurs de x_j, x_j étant un signal d'état

    Et $x_j \in \{In_{id_module}, IS_{id_module}, Out_{id_module}\}$

    - In_{id_module} est l'ensemble des entrées du module;
    - O_{id_module} est l'ensemble des sorties du module;
    - IS_{id_module} est l'ensemble des signaux internes du module.

- la table des HLS. À chacun des identificateurs de HLS *id_hlstate* sont assignés les intervalles de valeurs correspondants aux différents signaux d'états.

Tableau 5.8 HLS.

Id_hlstate	**I**		**IS**		**O**	
	Min	**max**	**Min**	**Max**	**Min**	**Max**

Les algorithmes de la procédure COMPUTE et sous routines sont décrits au Tableau 5.9.

Tableau 5.9 Procédures, paramètres et code.

Nom de la procédure	**Code**	

Compute	Entrées	//Représentations des opérations, conditions et affectations des signaux du module
		Table OP, Cond, SA
		//Liste des intervalles de valeurs des signaux d'états
		Set LV_0

	Tables Locales	SC: state signal constraint table SA : state signals assignments table LS: high level legal states table

	Code	Loop: Id_hlstate: 0..s
		$D_{id_hlstate}$= LV_0
		//On évalue toutes les opérations de condition d'un HLS pour calculer son domaine de définition
		For each id_op in OP(id_hlstate= id_hlstate, cond/SA= **cond**) do
		X= signal identifier (Cond(id_op))
		Cond_operator = operator (Cond(id_op))
		$R'(X)_{id_op}$= EVAL (f_{id_op}, $D_{id_hlstate}$)
		$R(X)_{id_op}$= COND_EVAL (cond_operator, $R'(X)_{id_op}$)
		$D_{id_hlstate}$= **UpdateDomain** ($D_{id_hlstate}$, R_{id_op}, X)
		SC= **UpdateTable**(SC, id_hlstate, X, R_{id_op})
		End for each;
		//On évalue les opérations d'affectation des signaux correspondants au HLS.
		For each id_op in OP(id_hlstate= id_hlstate, cond/SA= **SA**) do
		X=signal identifier (SA(id_op))
		$R(X)_{id_op}$= EVAL (f_{id_op}, $D_{id_hlstate}$)
		//Mise a jour de la table des affectations des signaux
		SA=**UpdateTable** (SA, id_hlstate, X, R_{id_op})
		End for each;
		End loop;
		(LS, Q)= **ExtractHLStable** (SA, SC)

	Sorties	Set Q Table LS
Update_ Domain	Entrées	Set D Range R Signal X

	Code	If X $\in$ D then
		If Min (R) > X_{min} (D (x_j= X)) then
		X_{min}(D(x_j= X))= Min(R)
		End if;
		If Max (R) < X_{max} (D(x_j= X)) then
		X_{max}(D(x_j= X))= Max(R)
		Else
		D= D$\bigcup R$
		End if;
		End if;

	Sortie	D

Nom de la procédure		Code
Extract_ HLS	Entrées	//les tables des affectations des signaux et des contraintes Table SA, SC

	Code	Loop Id_hlstate: 0..s do
		//initialisation de l'ensemble des HLS
		$Q_{id_hlstate}$= $\{\ \}$
		//Pour chaque HLS on extrait l'ensemble des intervalles de valeurs des différents signaux d'état
		For each (X=signal identifier(id_hlstate) in SA)
		I_IS_O= I|IS|O (SA(X, id_hlstate)
		Min= min(SA(X, id_hlstate)
		Max= max (SA(X, id_hlstate)
		$R = [\min, \max]$
		//On construit le HLS avec ses valeurs correspondantes
		$S_{id_hlstate}$= $\{Q_{id_hlstate} \cup R\}$
		//On ajoute les velaurs d'états à la table d'états.
		LS= LS $\cup$ $\{id_hlstate, I_IS_O, X, \min, \max\}$
		End for each;
		End loop;
		For Id_hlstate: 0..s do
		//Pour chaque HLS si certains signaux n'ont pas été assignés dans la table des affectations des //signaux SA on extrait leurs valeurs de la table des contraintes CT.
		For each (X=signal identifier(id_hlstate) in CT)
		If (X $\notin$ LS(id_hlstate))
		I_IS_O= I|IS|O (CT(X, id_hlstate)
		Min= min(CT(X, id_hlstate)
		Max= max (CT(X, id_hlstate)
		$R = [\min, \max]$
		$Q_{id_hlstate}$= $\{Q_{id_hlstate} \cup R\}$
		LS= LS $\cup$ $\{id_hlstate, I_IS_O, X, \min, \max\}$
		End If
		End for each
		End loop;

	Sorties	Q, LS

Les tableaux résultants de l'application de la procédure COMPUTE sur le design de la Figure 5.1 sont présentés ci-dessous.

Tableau 5.10 Contraintes des signaux d'états.

Id_op	**Id_hlstate**	**Signal identifier**	**Min**	**max**	**I|O|IS**
2	1	In1	0	0	I
5	2,3,4,5	In1	1	1	I
6	3	C_s	0	8	IS
9	4	C_s	9	15	IS
10	5	C_s	9	9	IS

Tableau 5.11 Affectation des signaux d'états.

Id_op	Id_hlstate	Signal identifier	Min	I|O|IS	Max
3	1	C_s	0	IS	0
4	1	Out1	0	O	0
7	3	C_s	1	IS	9
8	3	Out1	0	O	0
11	5	C_s	0	IS	0
12	5	Out1	1	O	1

Tableau 5.12 HLS.

Id_hlstate	I		IS		O	
	In1		C_s		Out1	
	Min	Max	Min	max	Min	max
1	0	0	0	0	0	0
2	1	1	1	9	0	0
3	1	1	0	0	1	1

5.4.3 Analyse hiérarchique du design

La troisième étape réside en une analyse hiérarchique du design basée sur l'information extraite lors de l'analyse syntaxique du code de chacun des modules. Un design VHDL est généralement décomposé en plusieurs modules et sous modules reliés entre eux. Ainsi l'espace d'états atteignables d'un module n'est pas seulement défini par son comportement, mais aussi par son interface qui le relie au reste du design. Et donc, afin de calculer l'ensemble des HLS de tout le design, les dépendances entre les modules doivent être prises en considération. Cette section décrit en détails l'analyse de la connectivité des modules et la construction du modèle aplati correspondant.

5.4.3.1 Analyse de connectivité des modules

La méthodologie proposée considère la division des designs en composants ou modules. Au cours de l'analyse syntaxique et lexicale, si le module sélectionné est composé d'instances d'autres modules, les sous-composants avec leurs connexions sont identifiés et les informations sont stockées comme indiqué à la Figure 5.4. Considérons le design de la Figure

5.6. La représentation correspondante de l'identification des modules est décrite dans les tableaux 5.13-5.17.

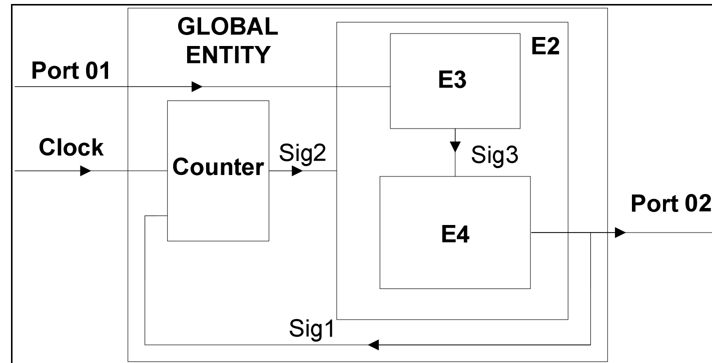


Figure 5.6 Exemple d'un design hiérarchique.

Tableau 5.13 Les représentations des entités du design de la Figure 5.6.

Entity_name	Id_port	Port_name	Direction	Sig.
Global entity	1	Clock	In	n/a
Global entity	2	Port01	in	n/a
Global entity	3	Port02	Out	Sig1
Counter	1	Clock	In	n/a
Counter	2	In1	In	n/a
Counter	3	Out1	Out	n/a
E2	1	Port_In1_1	In	n/a
E2	2	Port_In2_2	In	n/a
E2	3	Port_Out1_2	Out	n/a
E3	1	Port_In1_3	In	n/a
E3	2	Port_Out1_3	Out	n/a
E4	1	Port_In1_4	In	n/a
E4	2	Port_Out1_4	Out	n/a

Tableau 5.14 Les composants de l'entité Globale du design de la Figure 5.6.

Component_name	Id_port	Direction
Counter	1	In
Counter	2	In
Counter	3	Out
E2	1	In
E2	2	In
E2	3	Out

Tableau 5.15 Les instances de l'entité globale
du design de la Figure 5.6.

Id_instance	Component_name	Id_port	Signal
1	Counter	1	Clock
1	Counter	2	Sig1
1	Counter	3	Sig2
2	E2	1	Port01
2	E2	2	Sig2
2	E2	3	Sig1

Tableau 5.16 Les composants de E2
du design de la Figure 5.6.

Component_name	Id_port	Direction
E3	1	In
E3	2	Out
E4	1	In
E4	2	Out

Tableau 5.17 Les instances de E2
du design de la Figure 5.6.

Id_instance	Component_name	Id_port	Signal
3	E3	1	Port_In1_2
3	E3	2	Sig3
4	E4	1	Sig3
4	E4	2	Port_Out1_2

Une analyse de connectivité est réalisée basée sur les données des tables présentées ci-dessus. Pour chaque instance, le mappage des ports est examiné et les connexions avec d'autres instances sont détectées. Les frontières de la hiérarchie du design sont traversées tel que les instances du niveau supérieur et inférieur sont prises en compte.

Une structure de données présentée au Tableau 5.18 est construite. Chaque ligne représente une instance, et pour chaque instance, le niveau d'hiérarchie, ainsi que les dépendances des ports d'entrée et de sortie (si les sorties sont connectées aux sorties du module parent), sont bien définis. Cette structure sera utile pour construire le modèle aplati du design.

Tableau 5.18 Structure des données représentant les connexions du design de la Figure 5.6.

Inst	Module	niveau	In_dependencies	Out_dependencies
1	Counter	1	In1= Clock In2= Out1(2)	
2	E2	1	In1= port01 In2= Out1(1)	Out1= Port02
3	E3	2	In1= In1(2)	
4	E4	2	In1=out1(3)	Out1= out1(2)

5.4.3.2 Modèle aplati

Une fois l'information sur la connectivité et l'hiérarchie du design est extraite, un modèle aplati du design est construit. Ce modèle représente les relations directes entre les instances. Pour commencer les dépendances inter-modulaires du Tableau 5.18 sont mises à jour de la manière suivante:

Pour chaque instance I_i avec $i \in \{0..I\}$

Si ($in(I_i)= out(I_j)$) et I_i a un sous-module I_k et/ou I_j a un sous-module I_l avec

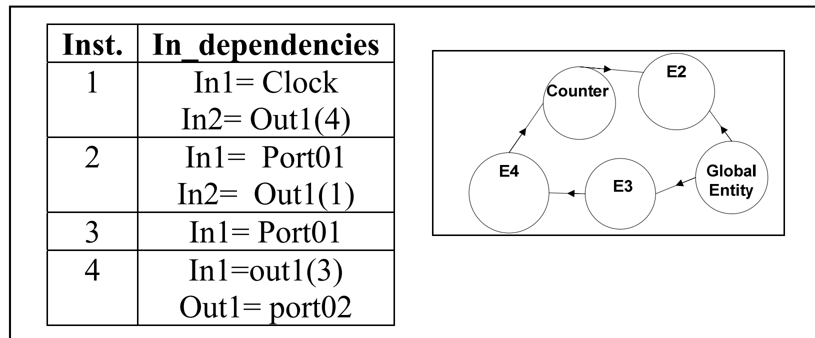
- *($in(I_k)= in(I_i)$) alors $in(I_k)= out(I_j)$*
- ($out(I_l)= out(I_j)$) then $in(I_i)= out(I_l)$
- ($in(I_k)= in(I_i)$) and ($out(I_l)= out(I_j)$)

Si($out(I_i)= out(I_j)$) et I_i a un sous-module I_k avec

- ($out(I_k)= out(I_i)$) alors $Out(I_k)= out(I_j)$

Figure 5.7 Pseudo-code de la création d'un modèle aplati.

Basée sur les données de la table transformée, on construit le graphe du modèle aplati. Les résultats sont présentés ci-dessous.



Inst.	In_dependencies
1	In1= Clock In2= Out1(4)
2	In1= Port01 In2= Out1(1)
3	In1= Port01
4	In1=out1(3) Out1= port02

Figure 5.8 Modèle aplati du design de la Figure 5.6.

5.4.4 Extraction des HLS du design

Afin d'extraire les HLS d'un design, une combinaison de l'ensemble des HLS des différents modules composant le design doit être effectué tout en respectant les dépendances inter-modulaire. Ainsi pour construire l'ensemble des HLS Q de tout le design, une combinaison des ensembles des HLS de chacune des I instances constituant le design est effectuée et ensuite l'ensemble des HLS final est minimisé et ajusté pour respecter les dépendances.

Basé sur l'algorithme présenté dans (Holzmann, 1991), nous définissons l'ensemble des HLS combinés comme le produit des différents ensembles de HLS. Étant donné I différents ensembles de HLS, Q_1, Q_2.. et Q_I avec $|Q_1| = n$, $|Q_2| = m$ et $|Q_I| = p$, l'ensemble des HLS combiné Q est défini comme suit:

$$Q = Q_1 \otimes Q_2 ... \otimes Q_I \tag{5.2}$$

Pour chaque $S \in Q, S = S_i(I_1) \otimes S_j(I_2) ... \otimes S_k(I_I)$

Avec $S_i\,(I_1) \in Q_1, i \in \{1...n\}$, et $S_j\,(I_2) \in Q_2, j \in \{1...m\}$,

et $S_k\,(I_I) \in Q_I, k \in \{1..p\}$.

Finalement, étant donné le modèle aplati, l'ensemble des HLS est mis à jour de façon à respecter les dépendances inter modulaire.

5.4.5 Extraction des contraintes fonctionnelles

Après le calcul de l'ensemble des HLS du design, on procède à l'extraction des contraintes. Afin de respecter les exigences des outils ATPG, les contraintes fonctionnelles doivent être présentées sous forme d'une formule booléenne.

Tout d'abord, une contrainte est extraite de chaque HLS de l'ensemble *Q*, appelé contrainte de HLS. Cette contrainte est une conjonction de contraintes individuelles dont les littéraux sont les bits des signaux d'états. Un exemple de contrainte individuelle correspondant au HLS S_0 de l'exemple est :

$$C=\overline{c_s_0} \text{ and } \overline{c_s_1} \text{ and } \overline{c_s_2} \text{ and } \overline{c_s_3} \text{ and } \overline{in1} \quad (5.3)$$

Ensuite, la contrainte globale, qui est une disjonction de toutes les contraintes des HLS, est formée. Le comportement du circuit doit satisfaire cette contrainte à tout moment. Le pseudo-code pour extraire les contraintes fonctionnelles est décrit ci-dessous.

```
1. K= 1;//nombre de HLS
2. While (k<= S ){
3. //Les contraintes individuelles sont crées à partir
4. //des valeurs des signaux dans chaque HLS
5. List(c0,...cnk)=Create_individual_constraints(Gk);
6. //les contrainte HLS sont des conjonctions des contraintes individuelles
7. Ck = c0 and ..and cnk;
8. K=k+1;
9. }//end while
10. //La contrainte globale est une disjonction des contraintes de HLS
11. C= C0 or .... or CNG
```

Figure 5.9 Pseudo-code de l'extraction des contraintes fonctionnelles du design.

Cette formule booléenne modélisant les contraintes du design est appliquée sur les outils ATPG dans le but de générer des vecteurs de test pseudo-fonctionnels et donc légaux.

5.5 Résultats expérimentaux

La méthode proposée a été implémentée et exécutée sur un Pentium, processeur Duo 2,2 GHz, et 1,99 Go de RAM. Dans cette section, nous présentons les résultats expérimentaux basés sur les benchmarks ITC'99 (Davidson). Le Tableau 5.19 décrit certaines caractéristiques des benchmarks utilisés, indiquant la taille et la complexité de chacun. L'outil présenté dans ce chapitre permet d'établir la liste des HLS légaux et des contraintes fonctionnelles à partir de la description VHDL du circuit. Les contraintes sont utilisées par la suite par des outils ATPG commerciaux afin de générer des tests pseudo-fonctionnels.
Cette section décrit l'efficacité de cette méthodologie au niveau vérification et test. En effet, ces vecteurs de tests générés peuvent être réutilisés dans la vérification des circuits sans risquer de forcer le circuit dans des états illégaux, ce qui empêche la détection de fausses erreurs. De plus, ces vecteurs de test permettent de réduire le nombre de fautes STFU détectées, ce qui permet l'amélioration de l'efficacité du test ainsi que la diminution des pertes de rendement.

Pour démontrer l'efficacité de l'approche proposée, nous montrons les résultats à deux niveaux différents. Le premier paragraphe décrit les résultats à un niveau RT dans le processus de vérification, se basant sur le nombre de fausses erreurs détectées lors de la simulation. Le second paragraphe décrit les résultats au niveau des portes logiques dans le processus de test, se basant sur la couverture de test atteinte et le nombre de fautes STFU détectées. Pour ce qui concerne le test, nous avons utilisé les mêmes paramètres utilisés dans (Liang, 1997; Lin, 2006; Lin, 2005; Wu, 2007) pour évaluer l'efficacité de notre méthode.

Tableau 5.19 Caractéristiques des circuits ITC'99 utilisés.

Circuit			code		Niveau porte logique	
Nom	**Entrées Primaires**	**Sorties Primaires**	**Nb de lignes**	**Nb de process**	**Portes logiques**	**FF**
B02	1	1	70	1	28	4
B03	4	4	141	1	149	30
B05	1	36	332	2	935	34
B07	1	8	92	1	420	49
B13	10	10	296	5	339	53
B14	32	54	509	1	4775	299

5.5.1 La vérification

Afin de montrer que les contraintes extraites permettent d'éviter la détection de fausses erreurs dans l'approche de vérification proposée dans cette thèse, nous effectuons des simulations des modèles VHDL et de leur modèles de références SystemC respectifs, en utilisant les vecteurs de test générés. L'outil de vérification, développé dans cette thèse, détecte une erreur lorsque le DUV et le modèle de référence correspondant présentent des comportements différents pour les mêmes entrées. De plus il assure le suivi en sauvegardant les valeurs des signaux au moment où l'erreur est détectée. Malgré la différence du style de code des deux modèles, aucune fausse erreur n'a été détectée lors de la simulation des vecteurs de test générés avec les contraintes fonctionnelles données par notre outil. Notons que lors de la simulation des vecteurs de test générés sans contraintes, de nombreuses erreurs ont été détectées. Toutes ces erreurs correspondent à des états illégaux et sont donc considérées comme des fausses erreurs. Le Tableau 5.20 décrit le nombre de fausses erreurs détectées dans le cas où aucune contrainte n'a été appliquée à l'outil ATPG.

Tableau 5.20 Nombre de fausses erreurs détectées sans contraintes.

Circuit	Fausses erreurs
B02	5
B03	11
B05	20
B07	21
B13	26
B14	68

5.5.2 Le test

Afin de montrer l'efficacité de notre méthodologie à régler le problème de sur testage, nous avons effectué des expériences de génération de vecteurs de test avec des outils ATPG commerciaux et comparons les résultats obtenus en appliquant les contraintes à ceux obtenus sans appliquer les contraintes. En présence des contraintes, l'outil ATPG ne génère que les vecteurs qui satisfont les conditions; et donc plusieurs fautes originairement détectées avec les ATPG sans contraintes ne sont plus détectées en présence des contraintes. Par conséquent, la couverture de test diminue et le nombre de fautes fonctionnellement non testable AU (*ATPG Untestable*) augmente, dépendamment de l'efficacité des contraintes imposées.

Le Tableau 5.21 permet une comparaison entre les résultats obtenus à l'aide des outils ATPG auxquels les contraintes générées sont appliquées et ceux obtenus à l'aide des outils ATPG sans aucune contrainte. Dans les deux cas, la génération des vecteurs de test se fait pour des modèles de fautes de transitions LOC (LCTF, « *Launch On Capture Transition Fault* »). Les couvertures de test correspondantes aux deux cas se trouvent respectivement dans les colonnes 2 et 3. Les colonnes 4 à 9 présentent le nombre de fautes détectées, le nombre de fautes AU et le nombre de fautes abandonnées correspondants aux 2 cas.

On peut déduire d'après les données dans le Tableau 5.21 que tous les circuits présentent une baisse de couverture dans le cas où des contraintes sont appliquées à l'outil ATPG. En effet,

l'outil ATPG avec contraintes détecte moins de fautes intestables (AU, « *ATPG Untestable* »). Par exemple, pour le circuit B14, 4,188 fautes AU ont été identifiées dans le cas où les contraintes sont appliqués alors que 1251 fautes AU sont identifiées dans le cas sans contraintes. Et donc, 9,55% des fautes détectées sans contraintes s'avèrent AU avec contraintes, et ce pourcentage est plus grand que celui trouvé dans (Wu, 2007) qui est 8.66 %, d'où l'efficacité des contraintes générées avec notre outil.

De même le nombre de fautes abandonnées est réduit en présence de contraintes.

Tableau 5.21 ATPG avec contraintes versus sans contraintes.

	Couverture LCTF (%)		**Fautes détectées**		**Fautes AU**		**Fautes abandonnées**	
	SC	**AC**	**SC**	**AC**	**SC**	**AC**	**SC**	**AC**
B02	95.29	92.93	162	158	8	12	0	0
B03	96.35	93.42	977	950	37	65	1	0
B05	97.39	95.63	1984	1952	47	81	4	2
B07	96.86	95.40	2269	2239	70	103	5	2
B13	96.15	94.22	1695	1562	63	197	1	0
B14	94.57	85.02	30758	27836	1251	4188	45	30

Notre approche est complètement automatisée, et comme l'analyse se fait au niveau RT la complexité du calcul est limitée. Quelques secondes sont suffisantes pour exécuter toute la procédure et générer des vecteurs de test pseudo-fonctionnels, à partir des modèles VHDL.

5.6 Conclusion

Les états illégaux d'un design causent plusieurs problèmes à différents niveaux : test et vérification. Dans la méthodologie de vérification qu'on propose dans cette thèse, les états illégaux causent une détection de fausses erreurs et affecte ainsi la qualité de la vérification. De plus, le processus de test est aussi affecté par les états illégaux, qui ajoutent des valeurs injustifiables à l'espace d'état du design, causant une augmentation de la complexité de la génération de vecteurs de test et une diminution de l'efficacité du test. D'ailleurs, plusieurs fautes FU-ST sont détectées, causant une perte de rendement importante et créant ce qu'on appelle un sur testage (Lin, 2005). Dans ce chapitre, nous avons présenté une nouvelle

méthode pour l'extraction des HLS et des contraintes fonctionnelles d'un design, utilisés pour éviter la génération de vecteurs de test illégaux. Notre méthodologie est basée au niveau RT et est entièrement automatisée. Le résultat est un outil qui prend en entrée la description VHDL de la conception et procède à une analyse lexicale et syntaxique pour extraire l'ensemble des HLS et des contraintes fonctionnelles de la conception. L'outil considère la division naturelle des designs en composants, il effectue une analyse de la connectivité et des dépendances inter-modulaire dont le résultat est pris en considération lors du calcul des états du design. Enfin, les résultats expérimentaux montrent l'efficacité de l'approche proposée à la fois au niveau test et vérification. En fait, les vecteurs de test générés à l'aide de l'ATPG avec contraintes permettent une amélioration de la qualité de la vérification et du test. Au niveau vérification aucune fausse erreur n'a été détectée et au niveau test la couverture de test (LCTF) ainsi que le nombre de fautes ST-FU détectées ont été réduits.

L'ensemble du processus est entièrement automatisé et est exécutée en quelques secondes pour les circuits considérées. De plus, l'outil peut être facilement adapté à n'importe quel autre langage matériel.

CHAPITRE 6

ENVIRONNEMENT DE VÉRIFICATION COMPLET PROPOSÉ

6.1 Introduction

En se basant sur les différents concepts et méthodologies décrits dans les chapitres précédents, nous proposons dans ce chapitre un environnement de vérification complet basé sur l'utilisation des vecteurs de test de transition LOC avec insertion des registres à balayage. L'environnement de vérification proposé est automatisé et contient trois outils complémentaires: en plus de l'extracteur de contraintes fonctionnelles présenté au chapitre précédent, on y retrouve également un générateur de banc d'essai, qui rend l'application des tests structurels au niveau RT possible avec un minimum d'effort, et un détecteur d'erreurs.

L'environnement de vérification regroupe deux niveaux d'abstraction, RTL et portes logiques, et peut être intégré dans le flux de conception standard sans nécessiter de modification.

Le chapitre est organisé comme suit. La section 6.2 décrit l'environnement de vérification automatisé dans son ensemble. Enfin, la section 6.3 donne une autre série de résultats expérimentaux présentant l'efficacité de l'environnement proposé en termes de vérification, comparée à celle de deux autres approches usuelles.

6.2 Environnement de vérification proposé

L'environnement de vérification proposé est décrit à la Figure 6.1. Les 3 outils développés dans le cadre de cette thèse apparaissent dans les ellipses ombragées. Nous présentons dans ce qui suit les deux outils qui n'ont pas encore été décrits, à savoir :

- Le générateur automatique de bancs d'essai, et;
- le détecteur d'erreurs.

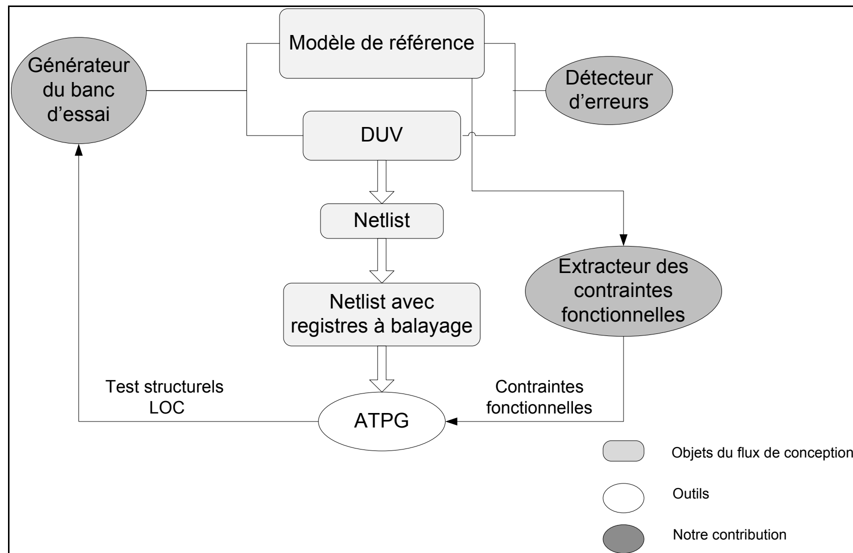


Figure 6.1 L'environnement de vérification proposé.

6.2.1 Le générateur automatique du banc d'essai

Un générateur automatique de bancs d'essais a été implémenté afin de minimiser le temps et l'effort requis pour l'adaptation et l'application des tests structurels au niveau de la simulation RTL.

L'outil prend en entrée le fichier de sortie de l'outil ATPG regroupant l'ensemble des tests structurels LOC ainsi que le DUV, et produit en sortie le banc d'essai VHDL correspondant. L'outil se base sur les concepts présentés dans le chapitre 3 sur les étapes et transformations requises pour l'application des tests structurels dans la simulation RTL. Il exécute les étapes suivantes :

- une analyse syntaxique du DUV. Cette analyse permet d'identifier :
  - l'entité à simuler;
  - la hiérarchie du design et les différents composants internes afin d'identifier le chemin hiérarchique de chacun des ports d'entrée et signaux internes à simuler;
  - Les types correspondants aux ports et signaux internes à simuler.
- une analyse syntaxique du fichier de sortie des outils ATPG contenant les tests structurels. Cette analyse permet d'identifier :

- les différents signaux internes et entrées primaires à simuler;
- les différents vecteurs de test correspondants à chacun des signaux.

- la construction d'une liste des vecteurs de test. Chaque vecteur de test est stocké avec les données suivantes :
  - l'identificateur du signal interne/entrée primaire correspondant;
  - le chemin hiérarchique allant de l'entité globale du signal/port correspondant;
  - le type du signal/port correspondant;
  - le vecteur de test correspondant en format série de bits.
- une conversion des vecteurs de test en différentes valeurs et formats correspondants aux types des signaux/ports à simuler. Ceci permet la transformation des séries de bit des vecteurs structurels afin qu'elles soient adaptées aux signaux/ports lors de la simulation RTL;
- la construction du banc d'essai suivant l'architecture suivante:
  - une entité vide;
  - une architecture contenant :
    - un composant correspondant à l'entité à simuler;
    - un processus pour la génération d'une horloge périodique;
    - un processus pour l'application de l'ensemble des vecteurs de test. Chaque vecteur de test correspond à un test structurel LOC basé sur l'insertion des registres à balayage et donc s'applique en deux phases de la manière suivante:
      - forcer les entrées primaires;
      - forcer les signaux internes (à l'aide de la fonction *force_signal* de Modelsim);
      - simuler une période;
      - forcer les entrées primaires;
      - simuler une période.

Le banc d'essai généré par cet outil est conçu en VHDL et permet de simuler le DUV et le modèle de référence en appliquant les vecteurs de tests structurels LOC basés sur l'insertion des registres à balayage générés par les outils ATPG. On rappelle que pour l'environnement

proposé actuel, le modèle de référence simulé est au niveau RT. Il peut être conçu en VHDL ou en SystemC.

6.2.2 Le détecteur d'erreurs

La dernière étape du processus suivant la simulation est la détection et l'identification des erreurs du design. Cette étape est automatisée, et le résultat est un outil qui prend en entrée les réponses du DUV et du modèle de référence, et produit en sortie la liste des erreurs détectées du design. L'observation n'est pas limitée juste aux sorties primaires du circuit mais elle couvre aussi tous les signaux d'états internes, ce qui permet une meilleure détection et localisation des erreurs. Le processus se déroule comme suit : après la simulation, le banc d'essai génère un fichier de sortie contenant toutes les valeurs des ports de sortie et des signaux d'état internes correspondant à chaque cycle de simulation du design simulé. Une comparaison des fichiers correspondants au DUV et au modèle de référence est effectuée afin de détecter les erreurs. Finalement, chaque erreur est identifiée avec les stimuli et les réponses du circuit correspondants.

6.3 Résultats expérimentaux

Afin de démontrer la fonctionnalité de l'environnement ainsi que l'efficacité de la méthodologie de vérification proposée, une comparaison avec d'autres méthodologies de vérification est effectuée, basée sur un ensemble des circuits benchmarks ITC'99. Les caractéristiques des circuits utilisés sont présentées dans le Tableau 6.1 (S. Davidson, 1999).

Les circuits sont synthétisés à l'aide d'un outil de synthèse commercial (Design Vision de Synopsys). Après synthèse, l'insertion des registres à balayage est effectuée à l'aide d'un outil DFT commercial (DFTAdvisor de Mentor Graphics). Les vecteurs de tests structurels de transition LOC sont générés par un outil ATPG commercial (Fastscan de Mentor Graphics). Des contraintes fonctionnelles, générées par l'outil d'extraction de contraintes proposée dans cette thèse, sont envoyés à l'ATPG afin de générer de tests pseudo-fonctionnels. Enfin, le générateur automatique de bancs d'essais génère le testbench basé sur

l'ensemble des tests structurels générés et visant le simulateur Modelsim de Mentor. La méthodologie proposée peut être adaptée à d'autres simulateurs tant que ces simulateurs permettent de forcer les signaux internes.

La simulation des deux modèles : DUV et de référence, se fait en parallèle à l'aide du banc d'essai. Les circuits ITC'99 ne contenant pas d'erreurs, ont été utilisé comme modèle de référence dans la simulation.

Tableau 6.1 Caractéristiques des circuits expérimentaux.

Circuits	Lignes VHDL	Entrées/Sorties Primaires	Bascules	Niveau hiérarchique	Portes logiques
B03	141	11/8	30	1	149
B05	332	1/6	34	1	935
B06	128	2/6	9	1	60
B07	92	1/8	49	1	420
B08	89	9/4	21	1	167
B09	103	1/1	28	1	159
B10	167	11/6	17	1	189
B13	296	10/10	53	1	339
B14	509	32/54	245	1	4,775
B18	1,424	36/23	3,320	3	68,752

Dans cette section, nous cherchons à évaluer l'efficacité de l'approche proposée (AP) en la comparant à deux techniques de vérification largement utilisées : l'approche pseudo-aléatoire (APA) et l'approche aléatoire basée sur les contraintes (AAC).

Le Tableau 6.2 montre une comparaison des couvertures atteintes lors de l'application des 3 approches. Les couvertures sont basées sur les métriques (états, transitions) des FSM, les résultats étant fournis par Modelsim. Ces métriques permettent de refléter essentiellement la rigueur de la simulation de la machine à états. Cependant, la corrélation avec la détection des erreurs RTL de conception n'est pas bien claire. Pour cette raison, nous avons procédé à une simulation des erreurs pour obtenir les résultats de couverture d'erreurs présentés au Tableau 6.3. Basé sur les modèles d'erreurs RTL (Campenhout, 1998) présentés au chapitre 4, différentes types d'erreurs ont été injectées dans chacun des modèles. Les modèles ont été ensuite simulés et la couverture d'erreurs calculée pour chaque approche. L'injection

d'erreurs a été réalisée manuellement, ce qui a limité le nombre d'erreurs injectées. Notre détecteur d'erreur proposé dans cette thèse a été utilisé pour détecter les erreurs.

Tableau 6.2 Comparaison des couvertures d'états et de transitions.

	États (%)			**Transition (%)**		
	APA	**AAC**	**AP**	**APA**	**AAC**	**AP**
B03	100	100	100	100	100	100
B05	85.4	100	100	64.7	88.2	90.3
B06	98.0	100	100	70.0	94.0	100
B07	100	100	100	82.4	95.4	98.2
B08	100	100	100	87.5	92.3	100
B09	75.0	96.0	100	63.6	92.3	100
B10	54.5	78.5	98	37.5	75.6	95.4
B13	63.3	86.5	93.4	45.8	75.4	91.3
B14	100	100	100	100	100	100
B18	56.0	-	98.0	47.0	-	96.8

Tableau 6.3 Comparaison des couvertures d'erreurs.

	Nb. d'erreurs injectées	**% d'erreurs détectées**		
		APA	**AAC**	**AP**
B03	20	100	100	100
B05	25	48	72	88
B06	20	75	85	100
B07	20	80	85	95
B08	20	75	90	100
B09	20	55	80	95
B10	20	30	65	90
B13	25	40	68	88
B14	25	92	100	100
B18	30	43	-	87

Sur la base de ces résultats expérimentaux, nous pouvons voir que notre approche surpasse clairement l'approche pseudo-aléatoire, et qu'elle donne des résultats en général meilleurs que ceux de l'approche AAC. Rappelons que l'AAC peut aussi générer automatiquement un grand nombre de vecteurs de tests respectant un nombre de contraintes spécifiées par

l'ingénieur de vérification. Par conséquent, elle couvre les coins sombres induisant une bonne couverture. Cependant, l'interaction humaine est nécessaire avec des efforts considérables pour construire une infrastructure de vérification, comprendre le code et définir des contraintes efficaces. Avec l'approche proposée, aucun effort n'est nécessaire ni pour générer les bancs d'essai et les vecteurs de test, ni pour comprendre le code.

Notons que le circuit B18 est un modèle complexe composé de milliers de ligne de codes, de nombreux niveaux hiérarchiques et contenant divers types de données. Lire et comprendre le code et sa fonctionnalité prendrait plusieurs semaines sinon des mois. En raison de contraintes de temps et du fait qu'il n'y a pas de modèle SystemC disponible pour ce circuit, nous avons seulement utilisé notre approche automatisée pour appliquer les modèles de test structurels, estimer la couverture, et la comparer à l'approche pseudo-aléatoire. Dans ce cas, le modèle de référence utilisé est le circuit VHDL B18 avant insertion des erreurs. Cela montre clairement que notre méthode peut être appliquée sur des circuits complexes, sans aucune connaissance de la fonctionnalité du circuit. Enfin, il convient de mentionner qu'il a fallu moins de 3 heures pour appliquer notre méthodologie complète sur le circuit B18. Ceci englobe la synthèse, l'analyse d'insertions, la génération des vecteurs structurels et les étapes de la simulation (le tout exécuté sur un Sun Blade 900 MHz).

6.4 Conclusion

Nous avons présenté l'environnement de vérification complet et automatisé basé sur l'application de modèles de test structurels LOC dans avec registres à balayage. En plus des bonnes couvertures atteintes grâce à l'efficacité des tests de transition LOC appliqués et de l'émulation des registres à balayage, l'automatisation de l'environnement permet une réduction conséquente du temps et des efforts requis pour accomplir la vérification. Les longues phases complexes de lecture et de compréhension du DUV, et de conception des bancs d'essai et des différents scénarios de vérification ne sont plus nécessaires.

Les résultats ont montré que l’approche proposée permet d’atteindre rapidement des couvertures égales et la plupart des fois plus élevées que celles obtenues avec d'autres approches de vérification usuelles.

CONCLUSION

Dans cette thèse, nous avons élaboré une nouvelle méthodologie permettant d'améliorer la productivité et en général la qualité de la vérification fonctionnelle des circuits séquentiels complexes, plus précisément au niveau d'abstraction appelé transfert de registres. Cette méthodologie explore une combinaison entre le test et la vérification. L'idée est de tirer profit du développement et de l'évolution des techniques et outils de test (DFT, ATPG) pour améliorer la vérification. Ainsi, le résultat de nos travaux est un environnement de vérification automatisé permettant de surmonter les principaux défis de la vérification fonctionnelle, d'améliorer sa couverture et de diminuer le temps et les efforts investis pour la compréhension du design, la génération des bancs d'essai et la simulation des systèmes.

La contribution majeure de cette thèse est la combinaison des deux domaines de test et de vérification, considérés généralement comme deux phases indépendantes. L'approche proposée est bien différente des approches présentes dans la littérature, que ce soit au niveau de la manière d'aborder la vérification ou même de l'ordre des étapes dans lequel la vérification est effectuée. Nous suggérons d'effectuer la synthèse, l'insertion des registres à balayage ainsi que la génération des vecteurs de test durant l'étape même de la vérification. Ceci peut paraitre difficile à considérer étant donné le flux normal de conception où le test survient après que l'étape de vérification soit accomplie. Mais, les gains obtenus au niveau du temps, des efforts et de la couverture de la vérification justifient l'approche proposée et montrent son importance dans l'amélioration de la qualité de la vérification.

Tout d'abord, nous avons défini la relation existante entre ce qui est difficile à tester et ce qui est difficile à vérifier, montrant que ce qui est difficile à tester est difficile à vérifier alors que le contraire n'est pas nécessairement vrai. Cette corrélation a permis de justifier l'orientation de notre recherche vers l'utilisation des vecteurs de test structurels dans la vérification, afin de couvrir les coins sombres et ainsi améliorer la qualité de la vérification dynamique. Ensuite, nous avons élaboré une stratégie pour l'utilisation des tests structurels dans la simulation fonctionnelle RTL. Pour cela, une technique de passage du niveau des portes

logiques au niveau RTL a été développée tout en émulant l'insertion des registres à balayage dans la simulation RT. Ainsi, l'ensemble des tests structurels est généré au niveau porte logique et transformé en un banc d'essai permettant la simulation d'un modèle RTL. En plus de forcer ses entrées primaires, le banc d'essais résultant force également les signaux d'état. Ceci permet de réduire les longues séquences de vecteurs de vérification nécessaires pour couvrir les différents états des circuits séquentiels. Ainsi le comportement séquentiel est transformé en un comportement combinatoire et une bonne couverture d'états peut être atteinte en un temps réduit.

Le choix du type des tests structurels s'est porté sur le test de transition LOC avec insertion des registres à balayage. Les tests de transition LOC permettent une meilleure simulation de la fonctionnalité. En effet, une étude théorique a été réalisée afin de montrer qu'un ensemble réaliste d'erreurs RTL peuvent être détectées par ce type de tests. De plus, l'étude a montré l'efficacité des tests de transition par rapport au test basé sur les pannes de type stuck-at.

Toutefois, l'utilisation du test au niveau de la vérification a apporté une problématique secondaire : l'existence des états illégaux dans les vecteurs de test. L'un des objectifs des travaux de recherche a donc été de générer des tests structurels ne contenant que des états légaux. C'est pourquoi nous avons développé une nouvelle méthode pour l'extraction des états légaux de haut niveau et des contraintes fonctionnelles d'un design, afin de les utiliser pour éviter la génération de vecteurs de test illégaux. Cette méthodologie s'applique au niveau RT et est entièrement automatisée. Le résultat est un outil qui prend en entrée la description VHDL de la conception et procède à une analyse lexicale et syntaxique pour extraire l'ensemble des états légaux de haut niveau et des contraintes fonctionnelles de la conception. Les contraintes fonctionnelles sont ensuite utilisées par les outils ATPG afin de générer des tests légaux. La méthodologie proposée s'est avérée utile non seulement pour la vérification mais aussi pour améliorer la qualité du test. Les résultats montrent une diminution du surtest et de la complexité de ce dernier.

Finalement, un environnement de vérification fonctionnelle automatisé basé sur l'application de modèles de test structurels LOC avec insertion des registres à balayage a été proposé. L’environnement est composé de trois outils de base: 1) un extracteur de contraintes qui identifie les contraintes fonctionnelles de conception, 2) un outil générateur de banc d'essai, et 3) un détecteur d’erreurs. Il permet la génération automatique des bancs d’essai tout en évitant les états illégaux. L’efficacité des tests de transitions LOC ainsi que l’émulation des registres à balayage permet une amélioration des couvertures atteintes. De plus, l’aspect automatisé de l’environnement permet une réduction conséquente du temps et des efforts requis pour accomplir la vérification. Les phases de lecture et de compréhension des DUV, et de conception des bancs d’essai et des différents scénarios de vérification ne sont plus nécessaires.

RECOMMANDATIONS

L'environnement de vérification proposé dans cette thèse englobe deux niveaux d'abstraction, le niveau porte logique et le niveau RT. Il permet une simulation du DUV au niveau RTL. Le modèle de référence SystemC utilisé a le même degré de détails que le modèle RTL. Il serait possible d'élaborer l'environnement de vérification afin de pouvoir viser des niveaux d'abstraction plus haut que celui du RT. Ceci est possible en développant des adaptateurs qui permettront d'ajuster les stimuli générés par le banc d'essai RTL au niveau d'abstraction du DUV et du modèle de référence. En effet, dans (Jindal, 2003) les auteurs décrivent comment simuler un modèle SystemC transactionnel à l'aide d'un banc d'essai RTL. Le même principe peut être utilisé et appliqué à l'environnement proposé dans cette thèse afin d'exploiter la réutilisation verticale du banc d'essai RTL pour simuler le DUV à différents niveaux d'abstraction. De plus, ceci permet l'utilisation des modèles de référence à des niveaux d'abstraction plus hauts que celui du DUV.

En outre, l'étude théorique présentée dans le chapitre 4 a démontré l'efficacité des tests structurels de transition LOC dans la détection des différents modèles de fautes RTL. Pour certaines fautes la détection est garantie mais pour d'autres elle ne l'est pas. Des travaux futurs pourront être faits pour implanter les méthodes proposées dans le chapitre 4 afin de garantir la détection de certaines fautes. Et par la suite automatiser le processus et l'inclure dans l'environnement de vérification.

Pour les erreurs LCE et SCE, leur détection est garantie si toutes les valeurs des signaux d'état en question sont couverts. Ainsi, on peut développer un module permettant de déduire l'ensemble des valeurs des signaux d'état non couverts tout en se basant sur l'ensemble des valeurs des états légaux générés par l'extracteur de contraintes. Et après, extraire les contraintes fonctionnelles correspondantes afin de générer des vecteurs de test permettant de couvrir les valeurs manquantes et garantir ainsi la détection des erreurs de type LCE et SCE. Pour l'erreur NSE, sa détection est garantie si toutes les transitions de la FSM sont couvertes. Ainsi, on peut appliquer et implanter la technique *N-detect* dans la génération des vecteurs de

test de transition pour les nœuds non couverts identifiés dans le rapport de simulation de Modelsim, afin de montrer son efficacité dans la détection des erreurs de type NSE.

De plus, l'outil d'extraction des contraintes est basé sur la PEG VHDL proposée dans le chapitre 5. Cette PEG couvre la plupart des structures de base de VHDL. Par contre elle pourrait être développée pour supporter des structures VHDL plus sophistiquées, comme les packages et les appels de fonction, ou même un langage matériel différent comme le Verilog par exemple.

Finalement l'environnement de vérification complet pourrait être adapté à différents langages de programmation comme le Verilog, le SystemVerilog, le SystemC etc.

ANNEXE I

LES MODÈLES D'UN SYSTÈME

Une modélisation d'un système est une représentation abstraite de sa fonctionnalité et de ses caractéristiques. Elle s'avère très utile durant la phase de vérification. Ces modèles peuvent être classés en plusieurs catégories:

- les modèles formels
  - **automates finis**: Un exemple de ce modèle est la machine à états (FSM*, Finite State Machine*). Il s'agit d'une machine abstraite utilisée en théorie de la calculabilité et dans l'étude des langages formels. Un automate est constitué d'états et de transitions. Son comportement est dirigé par un mot fourni en entrée : l'automate passe d'un état à un autre, suivant les transitions, à la lecture de chaque lettre de l'entrée. On procède à l'extension de ces modèles pour introduire la notion de temps, pour former les **Automates finis temporisés**. Ce type d'automates permet la modélisation du temps pour les systèmes temporels. Il est pour ceci muni d'un ensemble d'horloges et de contraintes temporelles au niveau des états et des transitions ;
  - **BDD**: Chaque problème peut être représenté sous forme d'expression booléenne. Ce graphe est un graphe acyclique direct qui permet la représentation de telles expressions. On a de même les BDD ordonnés, *Ordered Binary Decision Diagram OBDD* où les variables respectent un ordre linéaire quelque soit le chemin pris. Et les OBDD réduit, *Reduced Ordered binary Decision Diagram ROBDD* où le nombre d'états est réduit.

Rappelons que la vérification statique, plus précisément la vérification des propriétés basée sur la satisfiabilité et la vérification de l'équivalence, se base sur ce genre de modèle.

- les graphes de flots
  - graphes de flots de données : ce graphe est dédié aux systèmes dominés par le flot de données. Ce genre de systèmes présente un flux de données à très haut débit pour un même contexte, et très peu de changements de contextes sont nécessaires ;

- graphes de flot de contrôle et de données *CDFG, Control Data Flow Graph* (Namballa, 2004) : ce type de graphe est dédié aux applications mixtes, autrement dit les applications dominées par le contrôle et les flots de données. Un exemple d'un CDFG est donné dans la Figure-A I-1.

Ces graphes peuvent être exploités dans la vérification dynamique pour mieux visualiser la couverture de la simulation et les parties du design non couvertes.

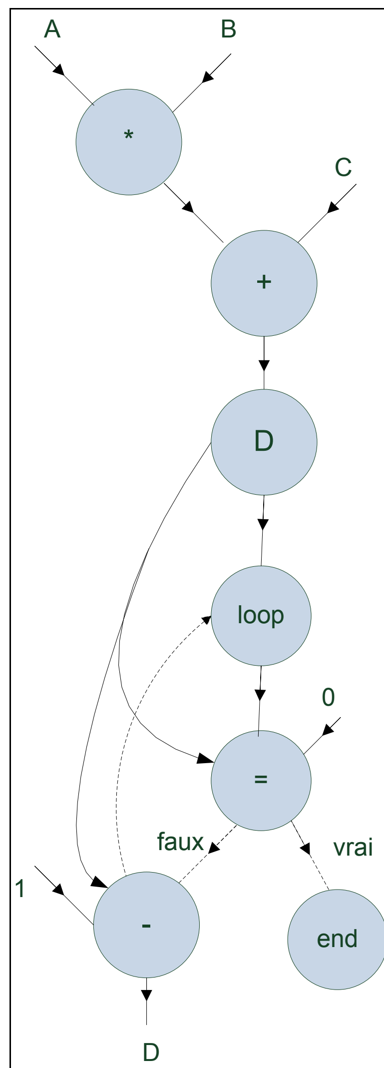


Figure-A I-1 Exemple d'un CDFG.
Adaptée de Namballa, Ranganathan et Ejnioui (2004)

ANNEXE II

CODE PERL DU GÉNÉRATEUR DU BANC D'ESSAI

```
############################################################
# @ copyright Christelle Hobeika                           #
############################################################

#!/usr/bin/perl
use POSIX qw(ceil floor);
use list::util qw(min max);
# definition du fichier de configuration
$config="config.txt";
open (config) or die "could not opwn the $config file\n";
@conf=<config>;
foreach $line (@conf)
{
      $line =~ s/#.*//;
      $line =~ s/\n//;
      if ($line =~ /Module_File_Name/)
      {
            @arg=split(/\t/,$line);
            $module_file_name=@arg[1];
            $module_file_name =~ s/\s//;
      }
      if ($line =~ /Pattern_File_Name/)
      {
            @arg=split(/\t/,$line);
            $pattern_file_name=@arg[1];
            $pattern_file_name =~ s/\s//;
      }
      if ($line =~ /Module_Name/)
      {
            @arg=split(/\t/,$line);
            $module_name=@arg[1];
            $module_name =~ s/\s//;
      }
      if ($line =~ /Period/)
      {
            @arg=split(/\t/,$line);
            $period=@arg[1];
      }
}
close (config);

# INITIALISATION DE L"ENSEMBLE DES  TABLEAUX
#le paragraphe décrivant les ports
```

```
@port_par="";
$tmp=pop(@port_par);
#le tableau contenant le nom des sous_modules
@component_inst_name="";
$tmp=pop(@component_inst_name);
# le tableau des cellules scan
@scan_cell="";
$tmp=pop(@scan_cell);
@scan_cell_indice="";
$tmp=pop(@scan_cell_indice);
# le tableau temporaire contenant les cellules de scan
@scan_cell_tamp="";
$tmp=pop(@scan_cell_tamp);
# le tableau des ports
@port="";
$tmp=pop(@port);
#le tableau contenant les ports des composants
@component_port="";
$tmp=pop(@component_port);
# le tableau des ports de sortie
@port_output="";
$tmp=pop(@port_output);
#caracteristique des cellules scan nom indice type
@scan_carac="";
$tmp=pop(@scan_carac);
# tableau contenant les indices des cellules de scan de sortie
@indice_output="";
$tmp=pop(@indice_output);
@foo="";
$tmp=pop(@foo);
# tableau contenant les ports d'entree a simuler
@PI="";
$tmp=pop(@PI);
#tableau contenant les pattern de test des cellules de scan
@scan_cell_pattern="";
$tmp=pop(@scan_cell_pattern);
#tableau contenant les pattern de test des PI1
@PI1_pattern="";
$tmp=pop(@PI1_pattern);
#tableau contenant les pattern de test desPI2
@PI2_pattern="";
$tmp=pop(@PI2_pattern);

@signals="";
$tmp=pop(@signals);

@global="";
$tmp=pop(@global);
```

```
# PREMIERE ETAPE TRAITEMENT DU FICHIER VHDL
# read the module file
# traitement du fichier vhdl decrivant l'entite a simuler
# on lit le nom de l'entite ou du module: module_name
# le paragraphe decrivant les ports
# et les ports de sortie k on stocke dans un tableau
# ouverture du fichier et stockage dans un tableau

open (module_file_name) or die " ERROR could not open
$module_file_name\n";
@module_sim = <module_file_name>;
#Lecture du nom du module et du paragraphe décrivant les ports
$i=0;
$bool1="false";
$bool2="false";
$k=@module_sim;
while ($i < $k)
{
        @module_sim[$i] =~ s/--.*//g;
        @module_sim[$i] =~ s/^\s*\n//g;
        if (@module_sim[$i]=~ /entity.*is/i)
        {
           $entity_name= &getEntityName(@module_sim[$i]);
           $entity_name=~ s/\s//;
           $bool1="true";
         }
         if (@module_sim[$i]=~ /signal/i)
         {
           &pushGlobal($entity_name,"SIGNAL",@module_sim[$i])
         }
         if (@module_sim[$i]=~ /variable/i)
        {
           &pushGlobal($entity_name,"VARIABLE",@module_sim[$i])
         }

         if ((@module_sim[$i]=~ /port/i) and (@module_sim[$i]!~
           /port  map/i) and ($bool1 eq "true"))
         {
           @module_sim[$i] =~ s/.*port.*\(//i;
           while ((@module_sim[$i]!~ /end.*$entity_name/i) )
           {
                &pushGlobal($entity_name,"PORT",@module_sim[$i]);
                $i++;
           }
           $bool1="false";
         }
         if (@module_sim[$i]=~ /port.*map/i)
        {
           @module_sim[$i] =~ s/--.*//g;
           @module_sim[$i] =~ s/^\s*\n//g;
```

```
            if (@module_sim[$i] =~ /:/)
            {
                  &pushGlobal_port_map($entity_name,"COMPONENT",
                  @module_sim[$i])
            }
      }
      $i++;
}
close(module_file_name);

# DEUXIEME ETAPE TRAITEMENT DU FICHIER CONTENANT LES PATTERN DE TEST
# read the PATTERN file
# traitement du fichier test pattern decrivant l'ensemble des test a
# simuler
# on lit l'ensemble des ports d'entree simules: PI
# l'ensemble des cellules de scan: scan_cell
# les pattern appliques

# ouverture du fichier de pattern et stockage dans un tableau
open (pattern_file_name)  or die "could not open
$pattern_file_name\n";
@pattern=<pattern_file_name>;
@array_pattern_file_name=@pattern;
$i=0;

# 1-creation du tableau contenant les entrees primaires en excluant
# la clock
# et memorisant l<indice de l'horloge dans l<ensemble des entrees,
# Création du tableau de scan_cell_tamp contenant toutes les
# cellules de scan temporaire

@component_inst_name = grep (/COMPONENT/,@global);
while ($i < @pattern)
{
#1- creation du tableau d'entree pirimaire en excluant la clock
#   declare input bus "PI" = "/clock", "/reset", "/C", "/scan_in1",
#   "/scan_en";
      $indice_virgule =$i;
      if (@pattern[$i] =~ /input\s*bus/i)
      {
                  while(  @pattern[$i] !~ /.*;.*\n/)
                  {
                        @pattern[$i] =~ s/\n//g;
                        $indice_virgule++;
                        @pattern[$i]="@pattern[$i]
                        @pattern[$indice_virgule]";
                  }
            # traitement 1 enlever les "/ et espace
            @pattern[$i] =~ s/^.*=//g;
            @pattern[$i] =~ s/"//g;
```

```
            @pattern[$i] =~ s/\///g;
            @pattern[$i] =~ s/\s//g;
            # traitement 2 split sur la virgule,PI_temp est
            # l'ensemble des entrees#
            @PI_temp = split(/,/, @pattern[$i]);
            $j=0;
            while (@PI_temp[$j] !~ /scan_in/)
            {
                  if (@PI_temp[$j] =~ /clock/)
                  {
#2- stocker l'indice de la clock dans les entrees primaire dans
# indice_clock
                        $indice_clock = $j;
                  }
                  elsif (@PI_temp[$j] !~ /clock/)
                  {
                        @PI_global=grep (/$module_name.*@PI_temp[$j]
                                    .*PORT.*/,@global);
                        if(@PI_global[0] =~ /\w/)
                        {
                              $indice_PI=" -";
                              $type_PI=@PI_global[0];
                              $type_PI =~ s/.*\t//;
                        }
                        else
                        {
                              @PI_temp[$j] =~ s/\[//;
                              @PI_temp[$j] =~ s/\]//;
                              if (@PI_temp[$j] =~ /\d\d$/)
                              {
                                    $indice_PI=substr @PI_temp[$j],
                                          length (@PI_temp[$j])-2,2;
                              }
                              else
                              {
                                    $indice_PI=substr @PI_temp[$j],
                                    length (@PI_temp[$j])-1,1;
                              }
                              @PI_temp[$j] =~ s/\d+$//;
                              @PI_global1=(/$module_name.*@PI_temp[$j]
                                    .*PORT.*/,@global);
                              $type_PI=@PI_global1[0];
                              $type_PI =~ s/.*\t//;
                        }

                        @PI_temp[$j] =~ s/\s//g;
                        @PI_temp[$j] =~ s/\n//g;
                        $type_PI =~ s/\n//g;
                        $indice_PI =~ s/\n//g;
# creation du tableau contenant les entrees primaires en excluant
```

```
# la clock: PI
                    push (@PI,"@PI_temp[$j]\t
                         $indice_PI\t$type_PI\n");
               }
               $j++;
          }
     }

#3- creation du tableau de cellules de scan temporaire


#scan_cell =    0  MASTER  FFFT  "/cnt_Sout_reg"  "_i2/mlc_dff2"
#               "SD"  "qb";

     elsif (@pattern[$i] =~ /scan_cell\s*=/)
     {

          @pattern[$i] =~ s/^.*"\///;
          @arg = split(/"/, @pattern[$i]);
          @arg[0]=~ s/\s//g;
          @arg[0]=~ s/\[//g;
          @arg[0]=~ s/\]//g;
          if ( (@arg[0] =~ /\d\d$/))
          {
               $indice=substr @arg[0], length (@arg[0])-2,2;
               @arg[0] =~ s/\d\d$//;
          }
          elsif (@arg[0] =~ /\d$/)
          {
               $indice = substr @arg[0], length (@arg[0])-1,1;
          }
          if (@arg[0] =~ /_reg.*_reg/)
          {
               @arg[0] =~ s/(.*)(_reg.*)(_reg.*)/\1\2/;
          }
          else
          {
               @arg[0] =~ s/_reg.*//;
          }

          @tamp = split(/_/,@arg[0]);
          foreach $line (@tamp)
          {
               if (grep (/$line/,@component_inst_name)
                    and $line =~ /\w/)
               {
                    @arg[0]=~ s/(^.*?)_/\1\//;
               }

          }
```

```
            push (@scan_cell_indice,"@arg[0]\t$indice");
            push (@scan_cell_tamp,"@arg[0]\n");

      }
      $i++;

}
close (pattern_file_name) ;

# Imprimer le tableau d'entree primaire PI, l'indice de l'horloge et
# le tableau des cellules de scan
# print "PI  indice  type ######:\n@PI\n";
# creation du tableau contenant les cellules de scan excluant les
# cellules de scan de sortie::: scan_cell
# creation du tableau contenant les indices des cellules de scan de
# sortie::: indice_output
@port_output=grep (/$module_name.*out/,@global);
$i=0;
while ($i < @scan_cell_indice)
{
      $scan_name = &getScanName(@scan_cell_indice[$i]);
      @foo = grep (/$scan_name/,@port_output);
      if (@foo)
      {
# création du tableau contenant les indices des cellules de scan de
# sortie::: scan_cell
            push (@indice_output,$i);
      }
      else
      {
# creation du tableau contenant les cellules de scan excluant les
# cellules de scan de sortie::: scan_cell
            $type = &getScanType(@scan_cell_indice[$i]);
            chomp($type);
            chomp (@scan_cell_indice[$i]);
            push ( @scan_cell,"@scan_cell_indice[$i]\t$type\n");
      }
      $i++;
}


# création des tableaux de pattern de test (PI1, SC, PI2)
open (pattern_file_name)  or die "could not open
      $pattern_file_name\n";
$pat="false";
$debut="true";
$PI2="false";
$chaine ="false";
$PI1 = "false";
$BOOL_PI2="false";
```

```
$line_pat=0;
$test=0;
$pattern_turn = "false";
$chain_turn = "false";
$PI1_turn = "false";
$PI2_turn = "false";
while ($line_pat <= @array_pattern_file_name)
{

      $line = @array_pattern_file_name[$line_pat];
      if ($line =~ /SCAN_TEST/ )
         {
            $pattern_turn="true";
         }
      if (($line =~ /pattern/) and ($pattern_turn eq "true"))
      {
            #print"pattern true\n";
            $pattern_turn = "false";
            $chain_turn="true";
      }
      if (($line =~ /chain/) and ($chain_turn eq "true"))
      {

            @tamp_i = split(/=/, $line);
            $line = @tamp_i[1];
            while ($line !~ /;/)
            {
                  $line_pat++;

                  $line="$line"."@array_pattern_file_name[$line_pat]"
                  ;
                  $line=~ s/\/.*\n//;
            }
            $line =~ s/\D//g;
# lecture de la valeur en excluant ceux correspondants aux registres
# de sortie
            $i=0;
            $chaine_pat ="";
            while ($i < length ($line)-1)
            {
                  $inv_i= length($line)-$i-2;
                  @foo = grep (/$i/,@indice_output);
                  if (@foo)
                  {
                  }
                  else
                  {
                  #creation de la variable chain_pat
                        $tamp = substr $line, $inv_i,1;
                        $chaine_pat= "$tamp"."$chaine_pat";
```

```
                }
                $i++;
            }
# Tableau contenant les pattern de test des cellules de scan
# excluant les sorties: scan_cell_pattern
            push (@scan_cell_pattern, "$chaine_pat");
            $chain_turn="false";
            $PI1_turn="true";
            $test++;
        }
        if (($line =~ /PI/) and ($PI1_turn eq "true"))
        {
            $line =~ s/\D//g;
            $i=0;
            $PI1_pat ="";
            while ($i <( length ($line)-1))
            {
                $inv_i= length($line)-$i-2;
                if (($i==$indice_clock) or ($inv_i== 0  ) or
                    ($inv_i== 1))
                {
                }
                else
                {
                    #creation de la variable chain_pat
                    $tamp = substr $line, $inv_i,1;
                    $PI1_pat= "$tamp"."$PI1_pat";
                }
                $i++;
            }
            # Tableau contenant les pattern de test des cellules de
            # scan excluant les sorties: scan_cell_pattern
            push (@PI1_pattern, "$PI1_pat");
            $PI1_turn="false";
            $PI2_turn="true";
        }
        if (($line =~ /PI/) and ($PI2_turn eq "true"))
        {
            $BOOL_PI2="true";
            $PI2_turn ="false";
            $pattern_turn ="true";
            $line =~ s/\D//g;
            $j=0;
            $PI2_pat ="";
            $tamp="";
            while ($j < (length ($line)-1))
            {
                $inv_j= length($line)-$j-2;
                if (($j==$indice_clock) or ($inv_j== 0) or
                    ($inv_j== 1))
```

```
				{
				}
				else
				{
					#creation de la variable chain_pat
					$tamp = substr $line, $inv_j,1;
					$PI2_pat= "$tamp"."$PI2_pat";
				}
				$j++;
			}
			# Tableau contenant les pattern de test des cellules de
			# scan excluant les sorties: scan_cell_pattern
			push (@PI2_pattern, "$PI2_pat");
		}
		$line_pat++;
}
$L_scp= @scan_cell_pattern;
$L_pi1p= @PI1_pattern;
$L_pi2p= @PI2_pattern;

## 3 ETAPE: CREATION DU FICHIER DE SORTIE
$taille_PI1= @PI1_pattern;
$testbench_nb= floor($taille_PI1/500);
for $counter (0..$testbench_nb)
{
		$out="out_$counter.vhd";
		$data_in="data_in.txt";
		open OUT, "> $out" or die "Can't open $out : $!";
		open data_in, "> $data_in" or die "Can't open $data_in : $!";

		print OUT "-- This testbench has been automatically generated
		-- using types std_logic and
		-- std_logic_vector for the ports of the unit under test.  ---
		-- Xilinx recommends
		-- that these types always be used for the top-level I/O of a
		-- design in order
		-- to guarantee that the testbench will bind correctly to the
		-- post-implementation
		-- simulation model.
		-------------------------------------------------------------
		------------------


		LIBRARY ieee;
		Library modelsim_lib;
		LIBRARY std;

		USE ieee.std_logic_1164.ALL;-- for type conversions
		USE std.textio.all;-- to manipulate files
		use modelsim_lib.util.all;
```

```
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_textio.all;
use work.txt_util.all;

-- Entity definition
-- Entity definition
ENTITY tb_fastscan IS
END tb_fastscan;


--Architecture definition
ARCHITECTURE behavior OF tb_fastscan IS \n";
chomp(@port_par);
chomp ($period);
@port_par=grep(/$module_name.*PORT/,@global);
foreach $value (@port_par)
{
      $value =~ s/\s/\t/;
      $value =~ s/$module_name\t//;
      $value =~ s/PORT/:/;
}
@module_signals=@port_par;
foreach $signal (@module_signals)
{
      $signal =~ s/(.*)/SIGNAL\t\1/;
      $signal =~ s/\sin\s|\sout\s|\sinout\s//;
}


@port = @port_par;
foreach $port_p (@port)
{
      $port_p =~ s/^\s+//;
      $port_p =~ s/\s.*//;
      $port_p =~ s/\n//;
}
print OUT "

      -- Component Declaration for the Unit Under Test (UUT)
      -- Traffic_Light_Controller
      COMPONENT $module_name port (@port_par
      END COMPONENT;
      --Clock period $period
      constant period : time :=$period;

      --Inputs/outputs
      @module_signals
```

```
            -- spy signals\n";

      @scan_element_spy= "";
      foreach $cell (@scan_cell)
      {
            $scan_name = $cell;
            chomp ($scan_name);
            $scan_name =~ s/\s.*//;
            @outi =grep (/$scan_name/,@scan_element_spy);
            if (@outi[0] !~/\w/)
            {
                  push(@scan_element_spy, $scan_name);
                  @cell = split(/\t/ , $cell);
                  @scan_name = split(/\// , $scan_name);
                  $last=@scan_name;
                  print OUT "SIGNAL @scan_name[$last-1] :
                              @cell[2]\n";
            }
      }

print OUT "
      BEGIN
       -- Instantiate the Unit Under Test (UUT)
             uut: $module_name PORT MAP(";
            $i=0;
            while ($i < @port)
            {
                  chomp ($port);

                  if ($i < @port-1)
                  {
                        print OUT "@port[$i] => @port[$i],\n";
                  }
                  else
                  {
                        print OUT "@port[$i] => @port[$i] );\n";
                  }
                  $i++;
            }
            print OUT "
             --clock definition
              PROCESS --horloge
                  BEGIN
                  clock<='1';
                  WAIT FOR PERIOD/2;
                  clock<='0';
                  WAIT FOR PERIOD/2;
            END PROCESS;
```

```
     --the test bench process
    tb : PROCESS
    FILE data_out: TEXT OPEN WRITE_MODE IS \"data_out.txt\";

    VARIABLE ligne_texte: line;

    BEGIN\n";

    foreach $cell (@scan_element_spy)
    {
         @cell = split(/\t/ , $cell);
         print OUT "init_signal_spy(\"/tb_fastscan/uut
         /@cell[0]\", \"@cell[0]\", 1, -1);\n ";
    }

print OUT "
 --initialise the inputs
            reset<='1';
            wait for ($period);\n";
            print data_in "---l'ensemble des stimulis---\n ";
$i=0;
$v=0;
$val=0;
if(@PI1_pattern< 500)
    {
         $i_max= @PI1_pattern;
    }
else
    {
         $i_max= 500;
    }
while ($i < $i_max)
{
    $nb_pattern= ($i+$counter)*2;
    print data_in "\n pattern\t=\t$nb_pattern\t: ";
    $j=0;
    $k=0;
    $l=0;
    $p=0;
    while ($p < @PI)
    {
         chomp (@PI[$p]);
         if(@PI[$p] !~ /^\s*\n/ )
         {
              $PI_name = &getScanName(@PI[$p]);
              chomp($PI_name);
              @foo=grep (/$PI_name.*integer.*/,@PI);
              $match=@foo;
              $match--;
              if(@foo)
```

```
                {
                        $val=0;
                        $v=0;
                        while($v <= $match)
                        {

                                $PI1_val = substr
                                @PI1_pattern[$i+$counter],
                                length (@PI1_pattern
                                [$i+$counter])- $j-1, 1;
                                $indice = @foo[$v];
                                @ind = split(/\t/ , $indice);
                                $indice = @ind[1];
                                $indice =~ s/\s//;
                                if($indice==31)
                                {
                                        $val= ((-1)**$indice) * $val;
                                }
                                else
                                {
                                        $val= ((2**$indice) *
                                        $PI1_val)+ $val;
                                }
                                $v++;
                                $j++;
                                $p++;

                        }
                        $PI1_val = $val;
                        print OUT "$PI_name<=$PI1_val;\n";
                        print data_in "$PI_name<=$PI1_val;";

                }
                else
                {
                        $PI1_val = substr
                        @PI1_pattern[$i+$counter],
                        length (@PI1_pattern[$i+$counter])
                        - $j-1 , 1;
                        print OUT "$PI_name<='$PI1_val';\n";
                        print data_in "$PI_name<=$PI1_val;";
                        $j++;
                        $p++;
                }
        }
}

$k = 0;
$l=0;
@scan_element="";
```

```
while ($k < @scan_cell)
{
      chomp (@scan_cell[$k]);
      if(@scan_cell[$k] !~ /^\s*\n/ )
      {
            $scan_name =@scan_cell[$k];
            $scan_name =~ s/\s.*//;
            @outi =grep (/$scan_name/,@scan_element);
            if (@outi[0] !~/\w/)
            {
                  push(@scan_element, $scan_name);
                  @int_type=grep
                  (/$scan_name\s.*integer.*/i,@scan_cell);
                  @bv_type=grep
                  (/$scan_name\s.*bit_vector.*/i,
                  @scan_cell);
                  @bit_type=grep (/$scan_name\s
                  .*bit;/i,@scan_cell);
                  print"integer_signal: indice type:
                  @int_type\n";
                  if(@int_type)
                  {
                        $val=0;
                        $v=0;
                        while($v <= @int_type-1)
                        {
                              $scancell_val = substr
                              @scan_cell_pattern
                              [$i+$counter],
                              length(@scan_cell_pattern[$i+
                              $counter])-$l -1, 1;
                              $indice = @int_type[$v];
                              @ind = split(/\t/ , $indice);
                              $indice = @ind[1];
                              $indice =~ s/\s//;

                              if($indice==31)
                              {
                                    $val= ((-1)**$indice) *
                                          $val;

                              }
                              else
                              {
                                    $val= ((2**$indice) *
                                    $scancell_val)+ $val;

                              }
                              $v++;
                              $l++;
```

```
                }
                $scancell_val = $val;
                print OUT
                "signal_force(\"/tb_fastscan
                /uut/$scan_name\",\"$scancell_val\"
                , 0 ns, freeze, $period, 1);\n";
                print data_in
                "$scan_name<=$scancell_val;";

        }
        elsif(@bit_type)
        {
                $scancell_val = substr
                @scan_cell_pattern[$i+$counter],
                length(@scan_cell_pattern[$i+$count
                er])-$l -1, 1;
                print OUT
                "signal_force(\"/tb_fastscan/uut/$s
                can_name\",\"$scancell_val\", 0 ns,
                freeze, $period, 1);\n";
                print data_in
                "$scan_name<=$scancell_val;";
                $l++;
        }
        elsif(@bv_type)
        {
                $v=0;
                while($v <= @bv_type-1)
                {

                        $scancell_val = substr
                        @scan_cell_pattern[$i+$counte
                        r],
                        length(@scan_cell_pattern[$i+
                        $counter])-$l -1, 1;
                        $indice = @bv_type[$v];
                        @ind = split(/\t/ , $indice);
                        $indice = @ind[1];
                        $indice =~ s/\s//;
                        print OUT
                        "signal_force(\"/tb_fastscan/
                        uut/$scan_name\[$indice\]\",\
                        "$scancell_val\", 0 ns,
                        freeze, $period, 1);\n";
                        print data_in
                        "$scan_name<=$scancell_val;";
                        $v++;
                        $l++;

                }
```

```
                    }
                    $k++;
                }
                else
                {
                    $k++;
                }
            }
        }

    if ($BOOL_PI2 eq "true")
    {
        print OUT "--Pulse clk
        wait for ($period);
        write(ligne_texte, \"output $nb_pattern :\"";
        foreach $output (@port_output)
        {
            chomp ($output);
            @v = split(/\t/,$output);
            if(@v[2] eq "std_logic_vector")
            {
                print OUT " &\"@v[1] = \" &hstr(@v[1])";
            }
            else
            {
                print OUT " &\"@v[1] = \" &str(@v[1])";
            }
        }

        @scan_element= "";
        @scan_name ="";
        foreach $cell (@scan_cell)
        {
            chomp ($cell);
            $temp=$cell;
            $temp =~ s/\s.*//;
            @outi =grep (/$temp/,@scan_element);
            if (@outi[0] !~/\w/)
            {
                push(@scan_element, $temp);

                @int_type=grep
                (/$temp\s.*integer.*/i,@scan_cell);
                @bv_type=grep
                (/$temp\s.*bit_vector.*/i,@scan_cell);
                @bit_type=grep (/$temp\s.*bit;/i,@scan_cell);
                @scan_name = split(/\// , $temp);
                $last=@scan_name;
                if(@bv_type)
```

```
                {
                      print OUT " &\" $temp = \"
                      &hstr(@scan_name[$last-1])";
                }
                else
                {
                      print OUT " &\" $temp = \"
                      &str(@scan_name[$last-1])";
                }
          }
    }

    print OUT ");\n";
    print OUT "WRITEline (data_out, ligne_texte);";
    $nb_pattern= ($i+$counter)*2+1;
    print data_in "\n pattern\t=\t$nb_pattern\t: ";
    $j=0;
    $a=0;
    while ($a < @PI)
    {
          chomp (@PI[$a]);
          if(@PI[$a] !~ /^\s*\n/ )
          {
                $PI_name = &getScanName(@PI[$a]);
                chomp($PI_name);
                @foo=grep (/$PI_name.*integer.*/,@PI);
                $match=@foo;
                $match--;
                if(@foo)
                {
                      $val=0;
                      $v=0;
                      while($v <= $match)
                      {

                            $PI2_val = substr
                            @PI2_pattern[$i+$counter], length
                            (@PI2_pattern[$i+$counter])-$j-1,
                            1;
                            $indice = @foo[$v];
                            @ind = split(/\t/ , $indice);
                            $indice = @ind[1];
                            $indice =~ s/\s//;
                            if($indice==31)
                            {
                                  $val= ((-1)**$indice) * $val;
                            }
                            else
                            {
```

```
                                        $val= ((2**$indice) *
                                        $PI2_val)+
                                        $val;
                                    }
                                    $v++;
                                    $j++;
                                    $a++;
                                }
                                $PI2_val = $val;
                                print OUT "$PI_name<=$PI2_val;\n";
                                print data_in "$PI_name<=$PI2_val;";
                            }
                            else
                            {
                                $PI2_val = substr
                                @PI2_pattern[$i+$counter], length
                                (@PI2_pattern[$i+$counter])-$j-1, 1;
                                print OUT "$PI_name<='$PI2_val';\n";
                                print data_in "$PI_name<=$PI2_val;";
                                $j++;
                                $a++;
                            }
                        }
                    }
                }
                print OUT "--Pulse clk
                wait for ($period);
                write(ligne_texte, \"output $nb_pattern :\"";
                foreach $output (@port_output)
                {
                    chomp ($output);
                    @v = split(/\t/,$output);
                    if(@v[2] eq "std_logic_vector")
                    {
                        print OUT " &\"@v[1] = \" &hstr(@v[1])";
                    }
                    else
                    {
                        print OUT " &\"@v[1] = \" &str(@v[1])";
                    }
                }
                @scan_element= "";
                @scan_name ="";
                foreach $cell (@scan_cell)
                {
                    chomp ($cell);
                    $temp=$cell;
                    $temp =~ s/\s.*//;
                    @outi =grep (/$temp/,@scan_element);
                    if (@outi[0] !~/\w/)
```

```
            {
                  push(@scan_element, $temp);

                  @int_type=grep (/$temp\s.*integer.*/i,@scan_cell);
                  @bv_type=grep
                  (/$temp\s.*bit_vector.*/i,@scan_cell);
                  @bit_type=grep (/$temp\s.*bit;/i,@scan_cell);
                  @scan_name = split(/\// , $temp);
                  $last=@scan_name;
                  if(@bv_type)
                  {
                        print OUT " &\" $temp = \"
                        &hstr(@scan_name[$last-1])";
                  }
                  else
                  {
                        print OUT " &\" $temp = \"
                        &str(@scan_name[$last-1])";
                  }
            }
      }

      print OUT ");\n";
      print OUT "WRITEline (data_out, ligne_texte);\n";
      print OUT "  \n \n";
      $i++;

}
print OUT "
      END PROCESS;

      END;";
}

sub getEntityName
{
      $line=@_[0];
      #print "match entity $line\n";
      $line =~ s/(entity)(.*)(is)/\2/i;
      $entity_name = $line;
      $entity_name =~ s/\n//;
      $entity_name =~ s/\r//;
      $entity_name =~ s/\W//;
      $entity_name =~ s/\s//;
      return ($entity_name);
}

sub pushGlobal
{
```

```
	$port_dir="-";
	$tamp = @_[2];
	$tamp =~ s/\n//;
	$tamp =~ s/--.*//;
	$type = $tamp;
	$type =~ s/.*://;
	$direction = $type;
	if ($direction =~ /(\sin\s)|(\sout\s)|(\sinout\s)/i)
	{
		$port_dir=$direction;
		$port_dir =~
		s/(.*)((\sin\s)|(\sout\s)|(\sinout\s))(.*)/\2/;
		$port_dir =~ s/\s//;
		$type =~ s/(\sin\s)|(\sout\s)|(\sinout\s)//i;
	}
	$tamp =~ s/:.*//;
	$tamp =~ s/(.*signal)(\s+)?(\w)/\3/i;
	$tamp =~ s/(.*variable)(\s+)?(\w)/\3/i;
	@arg=split(/,/, $tamp);
	foreach $signal (@arg)
	{
		if ($signal =~ /\w/)
		{
			chomp($signal);
			$signal =~ s/\s//;
			push(@global,
			"@_[0]\t$signal\t@_[1]\t$port_dir\t$type\n");
		}
	}

}
sub pushGlobal_port_map
{
	$line = @_[2];
	$line =~ s/\n//;
	$object_name=$line;
	$object_name=~ s/:.*//;
	$object_name=~ s/\s//;
	$type=$line;
	$type =~ s/^.*://;
	$type =~ s/port\s*map.*//i;
	push (@global,"@_[0]\t$object_name\t@_[1]\t-\t$type\n");
}
sub getScanName
{
	$scan_name =@_[0];
	$scan_name =~ s/.*\///;
	$scan_name =~ s/\[.*//;
	$scan_name =~ s/\t.*//;
	$scan_name =~ s/\W//;
```

```
	return ($scan_name);
}
sub getScanType
{
	$line =@_[0];
	$line =~ s/\t.*//;
	$path=$line;
	$name = &getScanName($line);
	#verifie sil faut envoyer @_[0] a la place de $line.
	@arg=split(/\//, $path);
	$path =@arg[length(@arg)-2];
	if ($path =~ /\w/)
	{
		@array= grep (/$path.*COMPONENT/,@global);
		$entity_n = @array[0];
		$entity_n =~ s/.*\t//;
		$entity_n =~ s/\n//;
		$entity_n =~ s/\W//;
		@array_2 = grep (/$entity_n.*$name/,@global);
		$type =@array_2[0];
		$type =~ s/.*\t//;
		$type =~ s/\W//;

	}
	else
	{
		@array_2 = grep (/$module_name.*$name/,@global);
		$type =@array_2[0];
		$type =~ s/.*\t//;
		#$type =~ s/\W//;

	}
	return ($type);
}

sub getModuleSignals
{
	@local=@_;
	foreach $value (@local)
	{
		$value =~ s/^\s*\n//;
		$value =~ s/\sin\s//i;
		$value =~ s/\sout\s//i;
		$value =~ s/\sinout\s//i;
		$value =~ s/\s//g;
		$value =~ s/(.*)/SIGNAL\t\1/i;
		$value =~ s/;//;
		$value =~ s/:/:\t/;
		$value =~ s/(.*)/\1;\n/;
	}
```

```
    return (@local);

}
exit;
```

ANNEXE III

CODE PERL DU DÉTECTEUR D'ERREURS

```
############################################################
# @ copyright Christelle Hobeika                          #
############################################################
#!/usr/bin/perl
open TXT2, "data_out_golden.txt" or die "$!";
open TXT1, "data_out_duv.txt" or die "$!";
$out="erreur.txt";
open OUT, "> $out" or die "Can't open $out : $!";

my %diff;
$diff{ $_ }=1 while (<TXT2>);

while(<TXT1>)
{
  print OUT unless $diff{$_};
}

close TXT1;
close OUT;
close TXT2;
open OUT, " $out" or die "Can't open $out : $!";
open ERROR, "> detected_errors.txt" or die "Can't open
detected_errors.txt : $!";
open IN, "data_in.txt" or die "Can't open data_in.txt : $!";
open TXT2, "data_out_golden.txt" or die "$!";

@output= <OUT>;
@IN= <IN>;
@output_golden= <TXT2>;
print "---------------@output_golden------------------";
$p= @output;
$j=0;
while ($j < $p)
{
      @erreur_in="";
      #print"j= $j";
      @arg = split(/\:/, @output[$j]);
      $indice_erreur= @arg[0];
      @golden=grep (/$indice_erreur/,@output_golden);
      print ERROR "output_golden\t=\t @golden \n";
      $indice_erreur =~ s/output/pattern/g;
      @erreur_in=grep (/$indice_erreur/,@IN);
      print ERROR "output_erroné\t=\t @output[$j] \n
```

```
     pattern_responsable\t=\t @erreur_in \n -------------------\n";
          $j++;
}
```

ANNEXE IV

CODE PERL DE L'EXTRACTEUR DES CONTRAINTES FONCTIONNELLES

```
############################################################
# @ copyright Christelle Hobeika                           #
############################################################
# Script perl de l'outil de gÉnÉration automatique des contraintes
# fonctionelles d'un design VHDL
# ENTRÉE: les fichiers VHDL constituant le design et un fichier de
# configuration indiquant le nom de la clock le nom du fichier et de
# l'entitÉ ^ traiter
# SORTIE: Un fichier contenant l'Ensemkble des contraintes
# fonctionnelles du design sous forme d'Équations boolÉennes et pr t
# pour l'ATPG
# TRAITEMENT: l'outil effectue les Étapes suivantes: 1) anaylse
# syntaxique et lexicale du code pour extraire et stocker les
# informations utiles 2) extraction des high level state
# HLS de chaque module 3) analyse de connectivitÉ du design 4)
# extraction des HLS du design et 5) gÉnÉration des contraintes
# fonctionnelles.

#!/usr/bin/perl
use POSIX qw(ceil floor);
use list::util qw(min max);
use Switch;
use Bit::Vector;


# lecture du fichier de configuration
$config="config.txt";

open (config) or die "could not opwn the $config file\n";
@conf=<config>;
foreach $line (@conf)
{
      $line =~ s/#.*//;
      $line =~ s/\n//;
      if ($line =~ /Module_File_Name/)
      {
            @arg=split(/\t/,$line);
            $module_file_name=@arg[1];
            $module_file_name =~ s/\s//;
      }

      if ($line =~ /Module_Name/)
      {
            @arg=split(/\t/,$line);
```

```
          $module_name=@arg[1];
          $module_name =~ s/\s//;
     }
     if ($line =~ /Clock/)
     {
          @arg=split(/\t/,$line);
          $clock=@arg[1];
          $clock =~ s/\s//;
     }
}
close (config);
print "module_file_name $module_file_name\n";
print "module_name $module_name\n";

# INITIALISATION DE QUELQUES  TABLEAUX
### TABLEAU DES TYPES DES SIGNAUX ET DES PORTS: SIGNAL      TYPE
@signal_type="";
$tmp=pop(@signal_type);
### TABLEAU DES VALEURS INITIALES DES SIGNAUX ET DES PORTS: SIGNAL
###   MAX   MIN   NB_BITS
@init_val_sig= "";
$tmp=pop(@init_val_sig);
### TABLEAU DE L'ENTITÉ: NAME    PORT  DIRECTION
@entity= "";
$tmp=pop(@entity);
### TABLEAU DES COMPOSANTS: ID_COMP   COMP_NAME  ID_PORT
     PORT_NAME  DIRECTION
@component="";
$tmp=pop(@component);
$id_component=0;
$id_port=0;
### TABLEAU DES INSTANCES
@instance="";
$id_instance=0;
$tmp=pop(@component_inst_name);
### TABLEAU des SIGNAL ASSIGNMENT : ID_SA STATE_SIGNAL
ID_ACTIVE_COND ID_OP EXPR_D
@sig_ass="";
#id assignment
$id_ass= 0;
$assig="false";
### TABLEAU DES VALEURS DES ASSIGNATIONS DES SIGNAUX :  ID_OP
     ID_HLSTATE SIGNAL     MIN   MAX IISO
@state_sig_ass="";
### TABLEAU DES IF : ID_IF ID_COND COND
@if_else="";
$id_if=0;
### TABLEAU DES CONDITION : ID_COND ID_OP SIGNAL OPERATOR
$id_cond= 0;
@cond="";
```

```
### TABLEAU DES CONTRAINTES : ID_OP ID_HLSTATE SIGNAL MAX MIN I_IS_O
@sig_constr="";
###Tableau des OPERATION de la forme : ID_OP MAX MIN
@oper="";
pop(@oper);
$id_operation= 0;
### TABLEAU DE CASE: ID_CASE     ID_COND    CONDITION
@case="";
$id_case=0;
$id_case_value=0;
### TABLEAU DES HLS: ID_HLSTATE  ID_COND
@HLS="";
### Tableau des conditions actives
@active_cond="";
### indice de l'État de haut niveau courant
$id_hlstate=0;
### compteur des États de haut niveau
$hlstate_count=0;
#le paragraphe decrivant les signaux d'états
@state_sig="";
$tmp=pop(@state_sig);
$state_sig_id=0;
### TABLEAU DES États lÉgaux
@LS="";
@global="";
$tmp=pop(@global);


# PREMIERE ETAPE TRAITEMENT DU FICHIER VHDL

# lecture et analyse du fichier VHDL
open (module_file_name) or die " ERROR could not open
$module_file_name\n";
@module_sim = <module_file_name>;
#Lecture du nom du module et du paragraphe decrivant les ports
$i=0;
$bool1="false";
$bool2="false";
$bool3="false";
$proc="false";
$arch="false";
$k=@module_sim;
$indice=0;
while ($i < $k)
{
           #supprimer les commentaires du fichier
        @module_sim[$i] =~ s/--.*//g;
        @module_sim[$i] =~ s/^\s*\n//g;
        #identification de l'entitÉ
        if (@module_sim[$i]=~ /entity.*is/i)
```

```
        {
                $entity_name= &getName(@module_sim[$i]);
                $entity_name=~ s/\s//;
                #ce boolÉen nous indiqu qu'on est dans
l'identification de l'entitÉ
                $bool1="true";
            }
            #identification de la dÉcalaration d'un port
            if ((@module_sim[$i]=~ /port/i) and (@module_sim[$i]!~
/port map/i) and ($bool1 eq "true"))
            {
                @module_sim[$i] =~ s/.*port.*\(//i;
                while ((@module_sim[$i]!~ /end.*$entity_name/i)
)#and (@module_sim[$i] !~ /end.*component/i))
                {

      &pushGlobal($entity_name,"PORT",@module_sim[$i]);
                        if (@module_sim[$i]=~
/(.*)(:\s*IN|out)(.*)\s*/i)
                      {
                                $id_port= 0;
                                @tab0="";
                                $port_type=  $3;
                                $port_name=  $1;
                                $direction = $2;
                                $direction =~ s/://g;
                                $port_type =~ s/\);//g;
                                $port_type =~ s/,//g;
                                $port_type =~ s/\n//g;
                                $port_name=~ s/://g;
                                $max_min= &initial_compute($port_type);

                                if ($port_name =~ /,/i)
                                {
                                     @port_name= split (/,/,$port_name);

                                     foreach (@port_name)
                                     {
                                          if($_=~/\w/)
                                          {
                                               $id_port++;
                                               $grp_port= join("\t",$_,
$port_type);

      push(@signal_type,"$grp_port\n");
                                               push (@init_val_sig,
"$_\t$max_min\n");
                                               push (@entity,
"$entity_name\t$id_port\t$_\t$direction\n");
                                          }
```

```
                                    }
                              }
                              else
                              {
                                    push (@signal_type,
"$port_name\t$port_type\n");
                                    push (@init_val_sig,
"$port_name\t$max_min\n");
                                    push (@entity,
"$entity_name\t$port_name\t$direction\n");
                              }
                        }
                        $i++;
                  }

            #identification de la fin de l'entitÉ
            $bool1="false";
            }
            #identification de la dÉcalaration d'un signal et
sauvegarde du signal
            #et de son type dans le tableau state_sig
            if (@module_sim[$i]=~ /signal\s*(.*)(:)(.*)\s*/i)
         {
                  @tab0="";
                  &pushGlobal($entity_name,"SIGNAL",@module_sim[$i]);
                  $signal_name=  $1;
                  $signal_type=  $3;
                  $signal_type =~ s/;//g;
                  $signal_type =~ s/\n//g;
                  if ($signal_name =~ /,/i)
                  {
                        @signal_name= split (/,/,$port_name);
                        foreach (@signal_name)
                        {
                              if($_=~/\w/)
                              {
                                    $grp_sig= join("\t",$_,
$signal_type);

                                    push(@signal_type,"$grp_sig\n");
                              }
                        }
                  }
                  else
                  {
                        push (@signal_type,
"$signal_name\t$signal_type\n");
                  }
            }
            #identification de la dÉcalaration d'une variable
            if (@module_sim[$i]=~ /variable/i)
```

```
        {
      &pushGlobal($entity_name,"VARIABLE",@module_sim[$i])
            }
            #identification du dÉbut de l'architecture
            if (@module_sim[$i]=~ /architecture/i)
            {
                  $arch= true;
            }
            #identification de la fin de l'architecture
            if ((@module_sim[$i]=~ /end architecture/i) and ($arch=
true))
            {
                  $arch= false;
            }
            #identification d'un composant
            if (@module_sim[$i]=~ /component/i)
        {
                  $id_component= $id_component+1;
                  $id_port=0;
                  $component_name= &getName(@module_sim[$i]);
                  $component_name=~ s/\s//;
                  #ce boolÉen nous indiqu qu'on est dans
l'identification du composant
                  $bool3="true";
            }
            #identification de la dÉcalaration des ports du composant

            if ((@module_sim[$i]=~ /port/i) and (@module_sim[$i]!~
/port map/i) and ($bool3 eq "true"))
            {
                  @module_sim[$i] =~ s/.*port.*\(//i;
                  while ((@module_sim[$i]!~ /end.*component/i) )#and
(@module_sim[$i] !~ /end.*component/i))
                  {
                        if (@module_sim[$i]=~
/(.*)(:\s*IN|out)(.*)\s*/i)
                     {
                              @tab0="";
                              $port_type=  $3;
                              $port_name=  $1;
                              $direction = $2;
                              $direction =~ s/://g;
                              $port_type =~ s/\);//g;
                              $port_type =~ s/,//g;
                              $port_type =~ s/\n//g;
                              $port_name=~ s/://g;
                              if ($port_name =~ /,/i)
                              {
```

```
                                @port_name= split (/,/,$port_name);

                                foreach (@port_name)
                                {
                                      if($_=~/\w/)
                                      {
                                            $id_port= $id_port + 1;
                                            push (@component,
"$id_component\t$component_name\t$id_port\t$_\t$direction\n");
                                      }
                                }
                          }
                          else
                          {
                                $id_port= $id_port + 1;
                                push (@component,
"$id_component\t$component_name\t$id_port\t$port_name\t$direction\n"
);
                          }
                    }
                    $i++;
              }
              $bool3="false";
        }
        #identification d'un process
        if ((@module_sim[$i]=~ /process.*$clock/i) and
(@module_sim[$i]!~ /end process/i))
        {
              $proc="true";
        }
        #identification de fin d'un process
        if ((@module_sim[$i]=~ /end process/i) and ($proc eq
"true"))
        {
              $proc="false";
        }
        #identification d'un if
        if ((@module_sim[$i]=~ /.*if.*\((.*)\)/i) and ($proc eq
"true"))
        {
              $id_cond++;
              push ( @active_cond, "$id_cond,");
              $assig="false";
              $condition="";
              if ($1=~ /$clock/i)
              {
                    $cond= "false";
                    $sync= "true";
              }
              else
```

```
                {
                        $hlstate_count++;
                        $id_hlstate= $hlstate_count;
                        push(@HLS, "$id_hlstate\t@active_cond\n");

                        $cond="true";
                        $imbrication++;
                        $id_if++;
                        $condition= $1;
                        # Identification de la condition
                        if ($condition =~ m/(.*?)(<=|>=|>|<|=)(.*)/)
                        {
                                $op_gauche= $1;
                                $operator = $2;
                                $op_droite= $3;
                                push(@cond,
"$id_cond\t$id_op\t$op_gauche\t$operator\n");
                                push(@if_else,
"$id_if\t$id_cond\t$condition\n");
                                @temp2= grep(/^\s*$id_op/, @operation);
                                @temp3= split(/\t/,@temp2[0]);
                                $min=@temp3[2];
                                $min=~ s/\s//;
                                $max=@temp3[1];
                                $id_sig_cond++;
                                $id_operation++;
                                $I_IS_O= &I_IS_O($op_gauche);
                                $states= &state_cond($id_cond);
                                print"les tates sont: \n";
                                push(@oper,
"$id_operation\tCOND\t$op_droite\t$states\t$op_gauche\n");
                        }
                }
        }

        #identification d'une instance
        if (@module_sim[$i]=~ /port.*map/i)
       {
                @inst0="";
                @inst2="";
                @inst3="";
                $comp="";
                $port_inst= "";
                $id_instance++;
                @module_sim[$i] =~ s/--.*//g;
                @module_sim[$i] =~ s/^\s*\n//g;
                if (@module_sim[$i] =~
/(.*):(.*)(port\s*map\s*\()(.*)\);/i)
                {
                        print"detecte: @module_sim[$i] et 4=$4\n";
```

```
                        $port_inst= $4;
                        print"port_inst=$port_inst\n";
                        $comp= $2;
                        $comp=~ s/\s//;
                        print"component: $comp \n, tableau=
@component\n";
                        @inst0 = grep (/$comp/i,@component);
                        print"trouve: @inst0\n";
                        @inst3= split(/,/,$port_inst);
                        foreach (@inst0)
                        {
                              @inst1= split(/\t/,@inst0[0]);
                              $id_port_inst= @inst1[2];
                              push(@instance,
"$id_instance\t@inst1[0]\t$id_port_inst\t@inst3[$id_port_inst]\n");
                        }
                  }
          }
            ## identification d'une assignation d'un signal
            if ((@module_sim[$i]=~ /<=/i) and ($proc eq "true"))
        {
                  $assig="true";
                  $id_ass++;
                  @expr="";
                  $expr_d="";
                  $state_g="";
                  $max_min_bit="";
                  @glob="";
                  @sig="";
                  @temp0="";
                  @temp1="";
                  $state_sig="";
                  @temp0 = split(/<=/, @module_sim[$i]);
                  ##s'il y a un else avant l'assignation on fait
juste enlever le else
                  if (@temp0[0]=~/else/)
                  {
                        @temp0[0]=~ s/else//;
                  }
                  $state_sig = @temp0[0];
                  $state_sig =~ s/\s//g;
                  @sig =grep (/$state_sig/i,@signal_type);
                  @exist=grep (/$state_sig/i,@init_val_sig);
                  ## calculer la valeur minimale et maximale du
signal ainsi que le nombre de bits correspondant
                  ## en faisant appel a la fonction initial_compute
et en sauvegardant le resultat dans le tableau
                  ## @init_val_sig
                  if (@exist[0] !~ /\w/)
                  {
```

```
                    @type="";
                    $type="";
                    $type= @sig[0];
                    @type= split(/\t/, $type);
                    $state_sig_val=&initial_compute(@type[1]);
                    push (@init_val_sig,
"@type[0]\t$state_sig_val\n");
                }
                $expr_d = @temp0[1];
                $expr_d =~ s/\s//g;
                $expr_d =~ s/;//g;
                $expr_d =~ s/'//g;
                $expr_d =~ s/"//g;
                @temp3= split(/\t/,$max_min_bit);
                $min=@temp3[2];
                $min=~ s/\s//;
                $max=@temp3[1];
                $I_IS_O= &I_IS_O($state_sig);
                $id_operation++;
                push(@sig_ass,
"$id_ass\t$state_sig\t$id_operation\t$expr_d\t@active_cond\n");
                push(@oper,
"$id_operation\tSA\t$expr_d\t$id_hlstate\n");
            }
            #ELSE CONDITION IDENTIFICATION
            if ((@module_sim[$i]=~ /else/i) and ($proc eq "true") and
($cond eq "true"))
            {
                pop(@active_cond);
                $id_cond++;
                $hlstate_count++;
                $id_hlstate= $hlstate_count;
                $assig="false";
                $op_gauche= "";
                $operator = "";
                $op_droite= "";
                @temp="";
                @condition="";
                @condition= grep (/^\s*$id_if.*/,@if_else);
                @temp= split(/\t/,@condition[0]);
                push ( @active_cond, "$id_cond,");
                push(@HLS, "$id_hlstate\t@active_cond\n");
                $condition= @temp[2];
                if ($condition=~ /</)
                {
                    $condition =~ s/</>=/;
                }
                elsif ($condition=~ />/ )
                {
                    $condition =~ s/>/<=/;
```

```
                }
                elsif ($condition=~ /<=/ )
                {
                        $condition =~ s/<=/>/ ;
                }
                elsif ($condition=~ /<=/ )
                {
                        $condition =~ s/<=/</;
                }
                #Condition identification
                if ($condition =~ m/(.*?)(<=|>=|<|>|=)(.*)/)
                {
                        $op_gauche= $1;
                        $operator = $2;
                        $op_droite= $3;
                        $id_operation++;
                        $states= &state_cond($id_cond);
                        push(@oper,
"$id_operation\tCOND\t$op_droite\t$states\t$op_gauche\n");
                        push(@cond,
"$id_cond\t$id_op\t$op_gauche\t$operator\n");
                        push(@if_else,
"$id_if\t$id_cond\t$condition\n");
                }

        }

        #END IF CONDITION IDENTIFICATION
        if ((@module_sim[$i]=~ /.*end if.*/i) and ($proc eq "true"))
        {
                $assig="false";
                $cond= "false";
                $imbrication--;
                pop ( @active_cond);
                @id_hlstate= grep(/.*\t@active_cond/, @HLS);
                @id_hlstate= split(/\t/,@id_hlstate[0]);
                $id_hlstate= @id_hlstate[0];
        }

        #CASE STRUCTURE IDENTIFICATION
        if ((@module_sim[$i]=~ /\s*case\s*(.*)\s*is/i) and ($proc eq
"true"))
        {
                $case="true";
                $condition="";
                $id_case++;
                $case_signal= $1;
                $id_case_value=0;
                $imbrication++;
        }
```

```
	#CASE CONDITION IDENTIFICATION
	if ((@module_sim[$i]=~ /\s*when\s*(.*)\s*=>/i) and ($proc eq
"true") and($case eq "true"))
	{
		$case_value="";
		$case_value= $1;
		$case_value=~ s/"//g;
		$id_case_value++;
		@del="";
		@temp2="";
		@temp2="";
		if ($id_case_value != 1)
		{
			pop ( @active_cond);

		}
		$id_cond++;
		$hlstate_count++;
		$id_hlstate= $hlstate_count;
		$id_grp=$imbrication;
		$id_operation++;
		$states= &state_cond($id_cond);
		push(@oper,
"$id_operation\tCOND\t$case_value\t$states\t$case_signal\n");
		push ( @active_cond, "$id_cond,");
		push(@HLS, "$id_hlstate\t@active_cond\n");

	push(@case,"$id_case\t$id_cond\t$case_signal\t$case_value\n");
		push(@cond, "$id_cond\t$id_op\t$case_signal\t=\n");
		@temp2= grep(/^\s*$id_op/, @operation);
		@temp3= split(/\t/,@temp2[0]);
		$min=@temp3[2];
		$min=~ s/\s//;
		$max=@temp3[1];
		$id_sig_cond++;
		$I_IS_O= &I_IS_O($case_signal);
		push(@sig_constr,
"$id_op\t$id_hlstate\t$case_signal\t$min\t$max\t$I_IS_O\n");

	}
	#END CASE IDENTIFICATION
	if ((@module_sim[$i]=~ /.*end case.*/i) and ($proc eq "true"))
	{
		$assig="false";
		$cond= "false";
		$case= "false";
		pop ( @active_cond);
		@id_hlstate= grep(/.*\t@active_cond/, @HLS);
		@id_hlstate= split(/\t/,@id_hlstate[0]);
		$id_hlstate= @id_hlstate[0];
```

```
      }

      $i++;

}

print "le tableau de l'entitÉ est:@entity\n ";
print "le tableau des composants est:@component\n ";
print "le tableau des instances est: @instance\n";
print "le tableau des types des signaux est:@signal_type\n ";
print "le tableau des signaux d'état valeurs initiales
est:@init_val_sig\n ";
print "le tableau des if est:@if_else\n ";
print "le tableau des case est:@case\n ";
print "le tableau des conditions est:@cond\n ";
print "le tableau des assignations des signaux est:@sig_ass\n ";
print "le tableau des operations est :@oper\n ";
print "le tableau des etats de haut niveau: @HLS\n ";

#ÉTAPE 2: CALCUL DES ÉTATS LÉGAUX DE MODULE
# ÉVALUATION DES OPÉRATIONS ET CONSTRUCTION DE LA TABLE DES ÉTATS
# LÉGAUX
&compute;
#ÉTAPE 3: ANALYSE DE L'HIÉRARCHIE DU DESIGN
&hierarchical_analysis;

print "le tableau des signaux de contraintes: @sig_constr\n ";
close(module_file_name);

# LES DIFFÉRENTES FONCTIONS DE L'OUTIL

### Nom de la fonction: getName
### EntrÉe: la ligne de code contenat le nom de l'EntitÉ ou du
###composant
### Sortie: le nom de l'entitÉ ou du composant
### traitement: Extrait le nom de la ligne de dÉclaration de
l'entitÉ ou du composant
sub getName
{
      $line=@_[0];
      $line =~ s/(entity)(.*)(is)/\2/i;
      $line =~ s/(component)(.*)(is)/\2/i;
      $name = $line;
      $name =~ s/\n//;
      $name =~ s/\r//;
      $name =~ s/\W//;
      $name =~ s/\s//;
      return ($name);
}
```

```
sub pushGlobal
{
      $port_dir="-";
      $tamp = @_[2];
      $tamp =~ s/\n//;
      $tamp =~ s/--.*//;
      $type = $tamp;
      $type =~ s/.*://;
      $direction = $type;
      if ($direction =~ /(\sin\s)|(\sout\s)|(\sinout\s)/i)
      {
            $port_dir=$direction;
            $port_dir =~
s/(.*)((\sin\s)|(\sout\s)|(\sinout\s))(.*)/\2/;
            $port_dir =~ s/\s//;
            $type =~ s/(\sin\s)|(\sout\s)|(\sinout\s)//i;
      }
      $tamp =~ s/:.*//;
      $tamp =~ s/(.*signal)(\s+)?(\w)/\3/i;
      $tamp =~ s/(.*variable)(\s+)?(\w)/\3/i;
      @arg=split(/,/, $tamp);
      foreach $signal (@arg)
      {
            if ($signal =~ /\w/)
            {
                  chomp($signal);
                  $signal =~ s/\s//;
                  push(@global,
"@_[0]\t$signal\t@_[1]\t$port_dir\t$type\n");
            }
      }

}

### Nom de la fonction: bit_nb
### EntrÉe: une valeur donnée
### Sortie: le nombre de bits correspondant
### traitement: Donne le nombre de bits correspondant ^ la valeur

sub bit_nb
{
      $bit_nb="";
      $value=@_[0];
      if($value=0)
      {
            $bit_nb= 1;
      }
      elsif($value=1)
      {
            $bit_nb= 1;
```

```
	}
	else
	{
		$bit_nb= ceil((log($value)/log(2))+0.5);
	}
	return ($bit_nb);
}

#### Nom fonction: val_bit
#### entrÉe: nombre de bits
#### sortie: valeur max correspondant au nombre de bit
#### Traitement: fonction qui retourne la valeur maximale
#### correspondante au nb de bits donnÉ
sub val_bit
{
	$bit_nb=@_[0];
	$val_bit= (2**$bit_nb)-1;
	return ($val_bit);
}


#### Nom fonction: initial_compute
#### entrÉe: type
#### sortie: valeur max et min
#### Traitement: fonction qui retourne la valeur maximale et
####minimale
#### et le nb de bits correspondant a un signal donnÉ

sub initial_compute
{
	$val_max=0;
	$val_min=0;
	$bit=0;
	$type=@_[0];
	if (($type =~ /std_logic_vector\((\d) to (\d)\)/i) or ($type
=~ /bit_vector\((\d) to (\d)\)/i))
	{
		$val_max= &val_bit($2+1);
		$val_min= &val_bit($1);
		$bit= $2+1;
	}
	elsif(($type =~ /std_logic_vector\s*\((\d) downto (\d)\)/i) or
($type =~ /bit_vector\((\d) downto (\d)\)/i))
	{
		$val_max= &val_bit($1+1);
		$val_min= &val_bit($2);
		$bit= $1+1;
	}
	elsif(($type =~ /std_logic/i) or ($type =~ /bit/i) or ($type
=~ /boolean/i))
```

```
        {
              $val_max= 1;
              $val_min= 0;
              $bit= 1;
        }
        elsif($type =~ /integer range (\d) to (\d)\)/i)
        {
              $val_max= &val_bit($2+1);
              $val_min= &val_bit($1);
              $bit= $2+1;
        }
        elsif($type =~ /integer range (\d) downto (\d)\)/i)
        {
              $val_max= &val_bit($1+1);
              $val_min= &val_bit($2);
              $bit= $1+1;
        }
        elsif($type =~ /integer/i)
        {
              $val_max= &val_bit(32);
              $val_min= &val_bit(0);
              $bit= 32;
        }
        $ret=join ('     ',$val_max, $val_min, $bit);
        return($ret)
}
#### Nom fonction: cond_ident
#### entrÉe: la condition d'un if
#### sortie: signal sous condition, expression de la condition
#### Traitement: elle identifie la condition d'un if. Elle
#### retourne la partie gauche de l'expression et la aprtie
###droitefonction qui
#### retourne la valeur maximale et minimale
#### et le nb de bits correspondant a un signal donnÉ

sub cond_ident
{
        $op=@_[0];
        if ($op =~/(.*?)(=|<|>|<=|>=)(.*)/)
        {
              @op_gauche= &op_eval($1);
              @op_droite= &op_eval($2);
              return(@op_gauche,@op_droite);
        }
}

### Nom de la fonction: expr_tab
### EntrÉe: une formule boolÉenne
### Sorte: un tableau dont cahque ligne contient un opÉrande et un
###opÉrateur
```

```
### Traitement: fonction qui retourne le tableau formé par les
###opérandes et l
### les opérations d'une expression
### exemple: expr = a + b * c devient
### @op:      *   c
###           +   b
###          a   -
sub expr_tab
{
      @op="";
      $op=@_[0];
      while ($op =~/(.*)(or|and|xor|\+|\*|\/|\-)(.*)/)
      {
            $op= $1;
            push (@op, "$2 \t $3 \n");
      }
      push (@op, "$op\t - \n");
      return(@op);
      }
#### Nom de la fonction: tab_eval
#### EntrÉe : le tableau contenant les opÉrateurs et opÉrandes d'une
###opÉration
#### Sortie : Les valeurs maximales et minimales du rÉsultat
### correspondant MAX MIN NB_BITS
#### Traitement: prend en entrÉe la sortie de la fonction expr_tab
###et donne les rÉsultats
#### max et min de l'opÉration

sub tab_eval
{
      @op=@_[0];
      @domain=@_[1];
      $list="$id_ass";
      $max_res=0;
      $min_res=0;
      $bit_res=0;
      $max=0;
      $min=0;
      $bit=0;
      #taille du tableau
      $taille=@op - 1;
      @temp0="";
      @temp1="";
      @temp4="";
      pop(@temp4);
      @temp5="";
      @temp_case0="";
      @temp_case1="";
      @temp_case2="";
      @term0= split(/\t/,@op[$taille]);
```

```
@term0[0]=~s/\s//;
if(@term0[0]=~/\d/)
{
      $min_res=@term0[0];
      $max_res=@term0[0];
      $bit_res=&bit_nb($max_res);

}
else
{
      @temp0="";
      $list=$list."\t".@term0[0];
      @temp0= grep(/@term0[0]/, @domain);
      if(@temp0[1]!~/\w/)
      {
            @temp0= grep(/@term0[0]/, @init_val_sig);
            @temp1= split(/\t/,@temp0[0]);
            $min_res=@temp1[3];
            $max_res=@temp1[2];
            $bit_res=@temp1[4];
      }
      else
      {
            @temp0= grep(/@term0[0]/, @init_val_sig);
            @temp1= split(/\t/,@temp0[0]);
            $min_res=@temp1[3];
            $max_res=@temp1[2];
            $bit_res=@temp1[4];
      }
}
 while ($taille>1)
{
       @term1= split(/\t/,@op[$taille-1]);
       @term1[1]=~s/\s//;
       if(@term1[1]=~/\d/)
       {
             $min=@term1[1];
             $max=@term1[1];
             $bit=&bit_nb($max);
       }
       else
       {
            $list=$list."\t".@term1[1];
            @temp2= grep(/@term1[1]/, @domain);
            if(@temp2[0]!~/\w/)
            {
                  @temp1= split(/\t/,@temp2[0]);
                  $min_res=@temp1[1];
                  $max_res=@temp1[2];
            }
```

```
                else
                {
                      @temp2= grep(/@term1[1]/, @init_val_sig);
                      @temp3= split(/\t/,@temp2[0]);
                      $min_res=@temp3[3];
                      $max_res=@temp3[2];
                      $bit_res=@temp3[4];
                }
           }
           if(@term1[0]=~ /\*/)
           {
                 $min_res=$min_res* $min;
                 $max_res=$max_res* $max;
           }
           if(@term1[0]=~ /\+/)
           {
                 $min_res=$min_res+ $min;
                 $max_res=$max_res+ $max;
           }
           $bit_res=&bit_nb($max_res);
          $taille--;
     }
     $state= join ("\t", $max_res, $min_res,$bit_res );
     return($state);
}

### Nom de la fonction: get_type
### EntrÉe: un signal du design
### Sortie: le type correspondant du design
### Traitement: la fonction vÉrifie dans les tableaux
### globaux le type du signal correspondant

sub get_type
{
     @op=@_[0];
     $state= @op[0];

     @glob =grep (/.*$state.*/,@global);
     @temp= split(/\t/,@glob[0]);
     $type= @temp[5];
     return($type);
     }


### Nom de la fonction: EVAL
### EntrÉe: l'opÉration a Évaluer, et le domaine d'EntrÉe
### Sortie: les rÉsultats de l'Évaluation MAX MIN
### Traitement: la fonction : (1) appelle expr_tab qui dÉcortique
### l'expression en opÉrateurs et opÉrandes et (2) appelle tab_eval
```

```
### qui evalue le tableau en fonction du domaine et donne le MAX MIN
###du rÉsultat

 sub EVAL
{
      $eval="";
      @op=@_[0];
      @domain= @_[1];
      $expr= @_[0];
      $expr =~ s/;//g;
      $expr =~ s/\s//g;
      ######si l'assignation est une simple valeur on convertit le
binaire en décimal et on stocke la valeur dans le tableau
      if ($expr=~/^"\d+|^'\d/ )
      {
            #OPERATION IDENTIFICATION
            #OPERATION EVALUATION
            $expr =~ s/"//g;
            $expr =~ s/'//g;
            $exprdec= &bin2dec($expr);
            $eval= join("\t",$exprdec,$exprdec);
      }
       else
      {
            #OPERATION IDENTIFICATION
            @expr= &expr_tab($expr);
            #OPERATION EVALUATION
            #J'appele le tab eval avec un flag de cond et les min et
max valeurs de cond
            $eval= &tab_eval(@expr, @domain);
      }

      return($eval);
}

### Nom de la fonction: bin2dec
### EntrÉe: un vecteur de bits en binaire
### Sortie: la valeur correspondante en dÉcimale
### Traitement: la fonction convertit la valeur binaire en dÉcimal
 sub bin2dec
{
       $vec="";
       $dec="";
       @op=@_[0];
       $bin= @op[0];
       $vec = Bit::Vector->new_Bin(32, $bin);
      $dec = $vec->to_Dec();
       return($dec);
}
```

```
sub min_max_cond
 {
      @sigcond_elt="";
      $min_max="";
      @signal="";
      $i=0;
      @sigcond_elt= split(/\t/, @_[0]);
      $signal_ass= @sigcond_elt[3];
      $max=@sigcond_elt[4];
      $min=@sigcond_elt[5];
      @signal= grep(/$signal_ass/,@_);
      while($i<@signal)
      {
            @SIG_ELT= split(/\t/, @signal[$i]);
            $i++;
            if (@SIG_ELT[1]>$max)
            {
                  $max = @SIG_ELT[1];
            }
            if ($min > @SIG_ELT[2])
            {
                  $min = @SIG_ELT[2];
            }
      }
      $min_max=join("\t", $min,$max);
      return($min_max);
}

### Nom de la fonction: ExtractLStable
### EntrÉe: les tableaux d'assignation des signaux et de contraintes
### Sortie:  Tableau des États lÉgaux de haut niveau
### Traitement: basÉ sur les rÉsultats des analyses antÉrieurs cette
###fonction
### nous donne l'ensemble final des États lÉgaux du module
sub ExtractLStable
{
      $b=0;
      while ($b < $hlstate_count)
      {
            foreach (@sa)
            {
                  @temp0=split(/\t/, $_);
                  if (@temp0[1]=~ /$b/i)
                  {
                        $I_IS_O= @temp0[5];
                        $min=@temp0[5];
                        $max= @temp0[5];
                        push(@LS, "$a\t$I_IS_O\t$min\t$max\n");
                  }
            }
```

```
          foreach (@CT)
          {
               @temp1=split(/\t/, $_);
               if (@temp1[1]=~ /$b/i)
               {
                    @temp2= grep("@temp1[1]",@LS);
                    if (@temp2[0]!=~ /\w/i)
                    {
                         $I_IS_O= @temp1[5];
                         $min=@temp1[3];
                         $max= @temp1[4];
                         push(@LS, "$a\t$I_IS_O\t$min\t$max\n");
                    }
               }
          }
          $a++;
     }
     return(@LS);
}

### Nom de la fonction: I_IS_O
### EntrÉe: le nom du signal
### Sorte: un string: I: entrÉe O: sortie IS: signal interne
### Traitement: fonction qui dÉtermine si le signal est un port
### d'entrÉe, de sortie ou un signal inetrne

sub I_IS_O
 {
    @signal= grep(/@_[0]/i,@entity);
     if (@signal[0]=~/\w/)
     {

          @sig_direction= split(/\t/, @signal[0]);
          if (@sig_direction[2]=~ /in/i)
          {
               $I_IS_O = "I";
          }
          elsif (@sig_direction[2]=~ /out/i)
          {
               $I_IS_O = "O";
          }
     }
     else
     {
          $I_IS_O = "IS";
     }
     return(    $I_IS_O);
}
### Nom de la fonction: state_cond
### EntrÉe: id_cond identificateur de la condition
```

```
### Sorte: la liste des États ayant la condtion active
### Traitement: fonction qui retourne la liste des États
### pour lesquels le id_cond est actif

sub state_cond
{

      @cond_state="";
      $id_cond_temp= @_[0];
      #print"le id_cond est : $id_cond\n";
      foreach (@HLS)
      {
            @state_temp=split(/\t/, $_);
            #print"state_temp : @state_temp[1]\n";
            #print"id_cond_temo : $id_cond_temp\n";

            if (@state_temp[1]=~ /$id_cond_temp/i)
            {
                  $cond_state= join(",",$cond_state,
"@state_temp[0]");
                  #print"le cond_states est : $ond_state\n";
            }
      }
      return($cond_state);
}
### Nome de la fonction: compute
### EntrÉe: les tableaux d'opÉrations de conditions et
### d'assignations de signaux
### Sortie: le tableau de l'ensemble des États lÉgaux @HLS
### Traitement: pour chacun de États la fonction Évalue l'ensemble
### des assignations des États
### tout en prenant en considÉration les contraintes sur les signaux

sub compute
{
      $a=0;
      while ($a < $hlstate_count)
      {
            @domain_state_a=@init_val_sig;
            foreach (@oper)
            {
                  @oper_temp=split(/\t/, $_);
                  if ((@oper_temp[1]=~ /COND/i) and (@oper_temp[3]=~
                  /$a/i))
                  {
                        $max_min=&EVAL(@oper_temp[2],
@domain_state_a);
                        @operator_temp= grep(/.*\t@oper_temp[0]\t.*/,
@cond);
```

```
                    $max_min=&EVAL_COND (@operator_temp[4],
                    $max_min);
                    push(@sig_constr,
                    "@oper_temp[0]\t@oper_temp[3]\t@oper_temp[4]\t
                    $max_min\n");
                    $arg="@oper_temp[0]\t@oper_temp[3]\t@oper_temp
                    [4]\t$max_min\n";
                    @domain_state_a=
                    &updateDomain(@domain_state_a, $arg);
                    }
                    if ((@oper_temp[1]=~ /SA/i) and
                    (@oper_temp[3]=~ /$a/i))
                    {
                         $max_min=&EVAL(@operator_temp[2],
                         @domain_state_a);
                         push(@state_sig_ass,
                         "@oper_temp[0]\t@oper_temp[3]\t@oper_tem
                         p[4]\t$max_min\n");
                    }
               }
               $a++;
          }

          @LS = &ExtractLStable;
          $out= "legal_states.txt";
          open out, "> $out" or die "can't open $out: $!";
          print out "@LS";
          return(@LS);
}

### Nom de la fonction: EVAL_COND
### EntrÉe: lÕopÉrateur de comparaison cond_operator et
### le rÉsultat de l'Évaluation des opÉrations EVAL (opÉration)
### Sorte: lÕintervalle des valeurs rÉsultant R.op_id_evaluation
### Traitement: fonction qui retourne les valeurs max et min
### correspondantes
### a un signal dans une condition

sub EVAL_COND
{
     $operator_temp= @_[0];
     $max_min_temp=@_[1];
     @m_temp=split(/\t/, $max_min_temp);
     $max= @m_temp[0];
     $min= @m_temp[1];

     if ($operator_temp=~ /</)
     {
          $min= $min -1;
          $max_min = join("\t",0, $min);
```

```
	}
	elsif ($operator_temp=~ />/ )
	{
		$min= $min +1;
		$max_min = join("\t",$min, $max);
	}
	elsif ($operator_temp=~ /<=/ )
	{
		$max_min = join("\t",0, $min);
	}
	elsif ($operator_temp=~ />=/ )
	{
		$max_min = join("\t",$min, $max);
	}
	elsif ($operator_temp=~ /=/ )
	{
		$max_min = join("\t",$min, $max);
	}

	return($max_min);
}
### Nom de la fonction: hierarchical_anlysis
### EntrÉe: le tableau des instances du modules
### Sortie:  Tableau de connectivitÉ entre les diffÉrents ports
### Traitement: analyse les connectivitÉs entre les diffÉrents ports
### du module

sub hierarchical_analysis
{
	foreach (@instance)
	{
		@hier="";
		$instances= "instances.txt";
		open(instances) or die "error could not open $instances";
		@instances_f= <instances>;
		$k = @instances;
		@inst_last=split(/\t/, @instances[$k]);
		$indice_inst= @inst_last[0];

		@inst_temp=split(/\t/, @_);
		$id_instance= @inst_temp[0];
		$sig=@inst_temp[3];
		@connect_temp=grep (/$sig/i, @instance);
		foreach (@connect_temp)
		{
			@connect= split(/\t/, @connect_temp[0]);
			$inst_id_f= $indice_last + 1;
		   @direction1_temp=grep (/@inst_temp[3]/i, @entity);
		   @direction2_temp=grep (/@connect[3]/i, @entity);
		   @direction2_temp=split(/\t/, @direction1_temp);
```

```
                @direction2_temp=split(/\t/, @direction2_temp);
                $direction1= @direction1_temp[1];
                $direction2= @direction1_temp[1];
                 push (@hier,
"$indice_id_f\t@inst_temp[1]\t$direction1@inst_temp[2]=$direction2@c
onnect[2]\(@connect[1]\)\n");
                 open out, "> $instances" or die "can't open
$instances: $!";
                 print instances "@hier";
           }
      }
}
```

```
# Script perl de la fonction permettant de faire une analyse
# hiérarchique du design
# ENTRƒE: le tabelau des diffÉrentes instances du design
# (id_instance  component_name  dÉpendance
# SORTIE: Un fichier contenant le tableau des dÉpendances du mod ele
# aplati
# TRAITEMENT: pour chaque instance la fonction cherche les
# dÉpendances avec les autres instances.

#!/usr/bin/perl
use POSIX qw(ceil floor);
use list::util qw(min max);
use Switch;
use Bit::Vector;


# lecture et analyse du fichier d'entrÉe contenant l'ensemble des
# isntances
$input_file="instances.txt";
@flatten="";
@model_sim="";
open (input_file) or die " ERROR could not open $input_file\n";
@module_sim = <input_file>;
$i=0;
#Pour chaque instance Ii
while ($i < @module_sim)
{
	@inst_arg= split(/\t/,@model_sim[$i]);
	$inst_indice= @inst_arg[0];
	$module_name= @inst_arg[1];
	$inst_dep= @inst_arg[2];
	@dep= split(/=/,$inst_dep);
	if (@dep[0]=~ /in(\d)/i)
	{
		$dep_droite= @dep[1];
		if ($dep_droite=~ /out/i)
		{
			$dep_droite=~ /(.*\()(\d)(\))/i;
			$Ij= $2;
			foreach (@modelsim)
			{
				$cond1= "false";
				$cond2= "false";
				@inst_arg1= split(/\t/,@model_sim[$i]);
				$inst_dep1= @inst_arg1[2];
				@dep1= split(/=/,$inst_dep1);
				#if in(ik)= in (Ii) then in(Ik)= out (Ij)
				if((@dep1[1]=~ /@dep[0]\($module_name\)/i) and
(@dep1[0]=~ /in\d/i))
				{
```

```
                        $inst_indice1= @inst_arg[1];
                        $newdep= /@dep1[0]= in\($inst_indice\)/;
                        push (@flatten,
"$inst_indice1\t$new_dep\n");
                        $cond1 ="true";
                    }
                    #if in(ik)= in (Ii) then in(Ik)= out (Ij)
                    elsif((@dep1[1]=~ /$dep_droite/i) and
(@dep1[0]=~ /out\d/i))
                    {
                        $inst_indice2= @inst_arg[1];
                        $newdep= /@dep[0]=
out\($inst_indice2\)/;
                        push (@flatten,
"$inst_indice2\t$new_dep\n");
                        $cond2 ="true";
                    }
                    #if out(Ii)= out (Ij) then out(Ii)= out (Ik)
                    elsif ((@dep1[1]=~ /@dep[0]\($module_name/i)
and  (@dep1[0]=~ /out\d/i))
                    {
                        $inst_indice3= @inst_arg[1];
                        $newdep= /@dep1[0]=
out\($inst_indice3\)/i;
                        push (@flatten,
"$inst_indice3\t$new_dep\n");
                    }
    #if in(ik)= in (Ii) and in(ik)= in (Ii) then out(Ik)= out (Ij)
                    if ($cond1= "true" and $cond2= "true")
                    {
                        $newdep= /in$Ind_in=
out\($inst_indice2\)/i;
                        push (@flatten,
"$inst_indice1\t$new_dep\n");
                        $cond2 ="true";
                    }
                }
            }
        }
        $i++;
        $out= "flatten_model.txt";
#Écriture du tableau rÉsultat du mod le aplati dans un fichier txt

}
open OUT, ">$out" or die "can't open $out: $!";
print out "@flatten";
close (input_file);
close (OUT);
```

```
# Script perl de la fonction permettant de faire le calcul de
# l'ensemble des états finaux de tout le design
# ENTRée: les ensembles des états légaux de chaque
# module du design id_state I_IS_O sig min max et les tableaux
# du modèle aplati du design.
# SORTIE: Un fichier contenant l'ensemble des états légaux du
# designe tableau des dŽpendances du mod ele aplati
# TRAITEMENT: Il s'agit d'une multiplication de l'ensemble des
# états des modules tout en considérant les connectivités.

#!/usr/bin/perl
use POSIX qw(ceil floor);
use list::util qw(min max);
use Switch;
use Bit::Vector;

print "please enter the files name with a tab between names:\n";
$names=<>;
@names_file= split(/\t/,$names);
$id_module=1;
$nb_modules=0;
@design_legal_state= "";
# lecture de l'ensemble des isntances états légaux des modules
foreach (@names_file)
{
      $id_module++;
      open (@_) or die " ERROR could not open @_\n";
      @LS[$id_module] = <@_>;
      @state_nb[$id_module]= split(/\t/,$LS[$id_module][-1]);
      @state_nb[$id_module]= $state_nb[$id_module][0];
      close (@_);
}

# mis a jour de l'Ensemble des Žtats lŽgaux en fonction des
# connexions modulaires
$input_file="flatten_model.txt";
open (input_file) or die " ERROR could not open $input_file\n";
@flatten = <input_file>;
$i=0;
#Pour chaque instance Ii
while ($i < @flatten)
{
      @inst_arg= split(/\t/,@model_sim[$i]);
      $inst_indice= @inst_arg[0];
      $inst_dep= @inst_arg[1];
      @dep= split(/=/,$inst_dep);
      if (@dep[0]=~ /in(\d)/i)
      {
            $dep_droite= @dep[1];
            if ($dep_droite=~ /out/i)
```

```
        {
              $dep_droite=~ /(.*\()(\d)(\))/i;
              $Ij= $2;
              @new= grep (@LS[$Ij], "$dep");
        }
        elsif ($dep_droite=~ /in/i)
        {
              $dep_droite=~ /(.*\()(\d)(\))/i;
              $Ij= $2;
              @new= grep (@LS[$Ij], "$dep");
        }
  }
  elsif (@dep[0]=~ /out(\d)/i)
  {
        $dep_droite= @dep[1];
        if ($dep_droite=~ /out/i)
        {
              $dep_droite=~ /(.*\()(\d)(\))/i;
              $Ij= $2;
              @new= grep (@LS[$Ij], "$dep");
        }
  }
  @LS[$Ij]= (@LS[$Ij], @new);
  $i++;
}
# Calcul de l'Ensemble des Žtats lŽgaux final du design
$nb_modules= $id_module;
$id_module=1;
@state_design="";
$state_id = 1;
$i=0;

while ($i< $state_nb[$1])
{
  $j=0;
  while ($j< $state_nb[$2])
  {
        @state_design[$id_state]="";
        @state_temp[$1]= grep (/$state_id/i, @LS[$1]);
        @state_temp[$2]= grep (/$state_id/i, @LS[$2]);
        @state_design[$id_state] = (@state_design[$id_state],
@state_temp[$1], @state_temp[$2]);
        $id_state++;
        $j++;
  }
  $i++;
}
$id_module=3;
while ($id_module< $nb_module)
{
```

```
        $i=1;
        while ($i< $state_nb[$id_module])
        {
              @state_temp[$i]= grep (/$id/i, @LS[$i]);
              @state_design[$i] = (@state_design[$i], @state_temp[$i]);
              $i++;
        }
        $id_module++;
  }
  $nb_state= $i;
  $id_state= 0;
  while ($id_state< $nb_state)
  {
        push(@design_legal_state, "@state_design[$id_state]\n");
        $id_state++;
  }
  $out= "design_legal_states";
  open OUT, ">$out" or die "can't open $out: $!";
  print out "@design_legal_state";
```

```
#!/usr/bin/perl
# definition du fichier de configuration
$config="config.txt";
@state="";
$tmp=pop(@state);
$tmp=pop(@array_state);
$tmp=pop(@state_name_temp);
$tmp=pop(@state_value);
open (config) or die "could not opwn the $config file\n";
@conf=<config>;
foreach $line (@conf)
{
      $line =~ s/#.*//;
      $line =~ s/\n//;
      if ($line =~ /Module_File_Name/)
      {
            @arg=split(/\t/,$line);
            $module_file_name=@arg[1];
            $module_file_name =~ s/\s//;
      }
      if ($line =~ /Module_Name/)
      {
            @arg=split(/\t/,$line);
            $module_name=@arg[1];
            $module_name =~ s/\s//;
      }
      if ($line =~ /fsm_rec_name/)
      {
            @arg=split(/\t/,$line);
            $fsm_rec_name=@arg[1];
            $fsm_rec_name =~ s/\s//;
      }
}
close (config);
print "module_file_name $module_file_name\n";
print "module_name $module_name\n";
print "fsm_rec_name $fsm_rec_name\n";


open (fsm_rec_name)  or die "could not open $fsm_rec_name\n";
@state=<fsm_rec_name>;
@array_state=@state;
$i=0;

while ($i < @state)
{
#1- creation du tableau d'entree pirimaire en excluant la clock

#   declare input bus "PI" = "/clock", "/reset", "/C", "/scan_in1",
# "/scan_en";
```

```
	if (@state[$i] =~ /state\s*variable/i)
	{

		#traitement 2 split sur la virgule,PI_temp est l<ensemble
des entrees#
		@state_name_temp = split(':', @state[$i]);
			print "state_name @state_name_temp[1]\n";
	}
	if (@state[$i] =~ /state\s*set/i)
	{

		#traitement 2 split sur la virgule,PI_temp est l<ensemble
# des entrees#
		@state_set_temp = split(':', @state[$i]);
		@state_set_temp[1] =~ s/{//g;
		@state_set_temp[1] =~ s/}//g;
		@state_set = split(/,/, @state_set_temp[1]);
		$j=0;
		while ($j< @state_set)
		{
			push (@state_value,"@state_set[$j]\n");
			$j++
		}

	}
	$i++
}
print "state_value @state_value\n";

$bit_nb= int((log(@state_value)/(log(2)))+0.5);
print "number of bits= $bit_nb\n";
$j=0;
while ($j< @state_value)
{
	if (@state_value[$j]=~ /\d/) {
	$i=0;
	@quotient[0]= @state_value[$j];
	@state_value_binary="";
	while($i < ($bit_nb-1))
	{
			@quotient[$i+1] = int(@quotient[$i]/2);
			print ("quotient= @quotient[$i]");
			$state_value_bit= @quotient[$i]- (@quotient[$i+1]
*2);
			push (@state_value_binary,"$state_value_bit\t");
			$i++
			}
			push (@state_value_binary,"$state_value_bit\n");
			print ("la valeurs binaire est :
@state_value_binary\n");
```

```
}
push (@state_value_bin,"@state_value_binary\n");

$j++
}
print ("les valeurs binaires sont : @state_value_bin\n");


$out="Functions_Constraints.txt";

open OUT, "> $out" or die "Can't open $out : $!";

$i=0;
$or_all="";
$name=@state_name_temp[1];
$name =~ s/\s+$//;
$name= $name."_reg";
      foreach $value (@state_value_bin)
      {
            @scan_value="";
            $scan_value = $value;
            chomp ($value);
            @scan_value = split(/\t/ , $scan_value);
            print OUT "add atpg functions and$i and ";
            $j=0;
            while ($j < @scan_value)
            {
                  #$indice=$bit_nb-$j-1;
                  $indice=$j;
                  if (@scan_value[$j] == 1)
                  {
                        print OUT "$name"."[$indice]/q ";
                  }
                  else
                  {
                        print OUT "$name"."[$indice]/qb ";
                  }
                  $j++
            }
            print OUT "\n";
            $or_all= $or_all." "."and$i"     ;
       $i++
      }
      print OUT "add atpg functions OR_all or $or_all\n";
      print OUT "add ATPG constraints 1 OR_all\n";
```

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

Abadir, M. S., Ferguson, J., Krikland, T.. 1988. « Logic design verification via test generation ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 7, p. 138-148.

Accellera. 2004. *Property Specification Language manual reference*. Accellera.

Al Asaad, H., Hayes, J. P.. 1995. « Design verification via simulation and automatic test pattern generation ». *1995 IEEE/ACM International Conference on Computer-Aided Design*, p. 174-180.

Baumgartner, J., Kuehlmann, A., Abraham, J.. 2002. « Property Checking via Structural Analysis ». *Computer Aided Verification, 2002,* p.154-165.

Bhaskar, J.. 1998. *A VHDL primer*. Upper Saddle River: Prentice Hall.

Boland, J.F, Thibeault, C., Zilic, Z. 2007. « Using Matlab and Simulink in a SystemC verification enviroment ». *DVCon07.*

Brahme, D.S., Cox, S., Gallo, J., Glasser, M., Grundmann, W., Norris, C., Paulsen, W., Pierce, J.L., Rose, J., Shea, D., Whiting, K. 2000. *The Transaction-Based Verification Methodology*. Report # CDNL-TR-2000-0825.

Campenhout, D.V., Al Asaad, H., Hayes, J.P., Brown, R. B. . 1998. « High-level Design verification of microprocessors via error modeling ». *ACM Trans. on Design Automation of Electronic Systems,* vol. 3, p. 581-599.

Chen C.H., Menon P.R. . 1989. « An Approach to Functional Level Testability Analysis ». *ITC - International Test Conference, 1989, Proceedinghs,* p. 373-380.

Cheng, K. T., Devadas, S., Keutzer, K. 1991. « A Partial Enhanced-Scan Approach to Robust Delay-Fault Test-Generation for Sequential-Circuits ». *ITC - International Test Conference, 1991*, *Proceedings*, p. 403-410.

Davidson, S.. 1999. « Characteristics of the ITC'99 Benchmark Circuits ». <www.cerc.utexas.edu/itc99-benchmarks>.

Dempster, D.J., Stuart, M.G. 2001. *Verification Methodology Manual: Techniques for Verifying HDL Designs*, second. Teamwork International.

FarWest Research, industry study (in conjunction with Mentor Graphics). 2007. « Funtional verification study ».

Ford, B. 2004. « Parsing expression grammars: A recognition-based syntactic foundation ». *ACM SIGPLAN Notices,* vol. 39, nº 1, p. 111-122.

Foster, H., Larsen, K., Turpin, M. 2006. « Introduction to the New Open Verification Library ». *DVCon06*. California.

Giomi, J.-C. 1995. « Finite state machine extraction from hardware description languages ». In *ASIC Conference and Exhibit, 1995., Proceedings of the Eighth Annual IEEE International* (18-22 Sep 1995), p. 53-357.

Goren, G., Ferguzon, F.J. 2002. « Testing finite state machine based on a structural coverage metric ». *ITC - International Test Conference, 2002, Proceedings,* p.773-780.

Hobeika, C., Thibeault, C., Boland, J. F. 2008. « Use of Structural Tests in RTL Verification ». *2008 1st Microsystems and Nanoelectronics Research Conference*, p. 133-136.

Hobeika, C., Thibeault, C., Boland, J. F. 2009. « Automatic Verification Methodology Based on Structural Test Patterns ». *2009 Joint IEEE North-East Workshop on Circuits and Systems and Taisa Conference*, p. 292-295.

Hobeika, C., Thibeault, C., Boland, J. F. 2010. « Illegal State Extraction From Register Transfer Level ». 8th *IEEE international NEWCAS,* p.245-248.

Hobeika, C., Thibeault, C., Boland, J. F. 2011. « Functional Constraint Extraction From Register Transfer Level. ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (soumis).

Holzmann, G. (162-185). 1991. *Design and Validation of Computer Protocols.* Prentice Hall.

Huang, S.Y., Cheng, K.T. 1998. *Formal equivalence checking and design debugging*. Kluwer academic publishers.

Inc., Collett International Research. 2002. *2002 IC/ASIC Functional Verification Study*.

Jindal, R., Jain, K. 2003. « Verification of transaction-level SystemC models using RTL testbenches ». *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, Proceedings*, p. 199-203.

Kang, J., Seth, S. C., Gangaram, V. 2007. « Efficient RTL coverage metric for functional test selection ». *25th IEEE VLSI Test Symposium, Proceedings*, p. 318-324.

Konijnenburg, M. H., Van der Linden, J. T., Van de Goor, A. J. 1999. « Illegal state space identification for sequential circuit test generation ». *Design, Automation and Test in Europe Conference and Exhibition 1999, Proceedings*, p. 741-746.

Krstic, A., Liou, J. J., Cheng, K. T., Wang, L. C. 2003. « On structural vs. functional testing for delay faults ». *4th International Symposium on Quality Electronic Design, Proceedings*, p. 438-441.

Lam, William K. 2005. *Hardware Design Verification, Simulation and Formal Method Based Approaches*. Prentice Hall Modern Semiconductor Design Series.

Liang, H. C., Lee, C. L., Chen, J. E. 1997. « Identifying invalid states for sequential circuit test generation ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 16, n° 9, p. 1025-1033.

Lin, Y.-C., Lu, F., Cheng, K. . 2006. « Pseudo-functional testing ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, p. 1535–1546.

Lin, Y. C., Lu, F., Yang, K., Cheng, K. T. 2005. « Constraint extraction for pseudo-functional scan-based delay testing ». *ASP-DAC 2005: Proceedings of the Asia and South Pacific Design Automation Conference,* vol. 1 and 2, p. 166-171.

Liu, H., Li, H. W., Hui, Y., Li, X. W. 2008. « A scan-based delay test method for reduction of overtesting ». *Delta 2008: Fourth IEEE International Symposium on Electronic Design, Test and Applications, Proceedings*, p. 521-526.

Liu, X., Hsiao, M. S. 2005. « A novel transition fault ATPG that reduces yield loss ». *IEEE Design & Test of Computers,* vol. 22, n° 6, p. 576-584.

Molina, A., Cadenas, O. 2007. « Functional verification: Approaches and challenges ». *Latin American Applied Research,* vol. 37, n° 1, p. 65-69.

Namballa, R., Ranganathan, N., Ejnioui, A. 2004. « Control and data flow graph extraction for high-level synthesis ». *IEEE Computer Society Annual Symposium on VLSI* p. 187-192.

Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G. 2007. « Constraint-based random stimuli generation for hardware verification ». *AI Magazine,* vol. 28, n° 3, p. 13-30.

Parthasarathy, G., Iyer, M. K., Cheng, K. T., Wang, L. C. 2004. « Safety property verification using sequential SAT and bounded model checking ». *IEEE Design & Test of Computers,* vol. 21, n° 2, p. 132-143.

Prasad P., Assi A., Raseen M., Harb A. 2004. « BDD Based Method for Fast Equivalence Checking ». *International Conference on Computational Intelligence,* p. 474-477. Turkey.

Rabaey, J.M., Chadrakasan A.P., Nikolic B. 1996. « Validation and test of manufactured circuits ». In *Digital integrated circuits a design perspective*, 2nd. p. 721-737. New Jersey: Prentice Hall Electronics and VLSI series.

Rearick, J. 2001. « Too much delay fault coverage is a bad thing ». *ITC - International Test Conference 2001, Proceedings*, p. 624-633.

Saidi, H. 2000. « Model checking guided abstraction and analysis ». *Static Analysis,* vol. 1824, p. 377-396.

Savir, J., Patil, S. 1993. « Scan-Based Transition Test ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 12, n° 8, p. 1232-1241.

Savir, J., Patil, S. 1994. « Broad-Side Delay Test ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 13, n° 8, p. 1057-1064.

Sematech, TX: International. 1999. « International Technology Roadmap for Semiconductors. ». *Semiconductor Industry Association*.

Séméria, L., Seawright, A., Mehra, R., Ng, D., Ekanayake, A., Pangrle, B. 2002. « RTL C-based methodology for designingand verifying a multi-threaded processor ». In *DAC 2002*, p.123-128. New Orleans, Louisiana.

Smith, G. L. 1985. « Model for delay faults based upon paths ». *ITC - International Test Conference 1985, Proceedings*, p. 342-349

Society, IEEE Computer. 2005. *IEEE Standard for System Verilog - Unified Hardware Design, Specification, and Verification Language*. New York: IEEE.

Sommerville, Ian. 2000. « Verification and validation ». In *Software Engineering*, 6th edition. Addison Wesley.

Tupuri, R. S., Abraham, J. A. 1997. « A novel functional test generation method for processors using commercial ATPG ». *ITC - International Test Conference 1997, Proceedings*, p. 743-752.

Varma, P. 2003. « Design verification problems: Test to the rescue? ». *ITC - International Test Conference 2003, Proceedings*, p. 1292-1292.

Vedula V. M., Abraham J. A. 2000. « Test Generation for Gigahertz Processors using an Automatic Functional Constraint Extractor ». In *Proc. IEEE International High-Level Design Validation and Test Workshop*. p. 9-14.

Vedula, V. M., Abraham, J. A. 2000. « A novel methodology for hierarchical test generation using functional constraint composition ». *IEEE International High-Level Design Validation and Test Workshop*, p. 9-14.

Wagner I., Bertacco, V., Austin, T. 2007. « Microprocessor Verification via Feedback-Adjusted Markov Models ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 26, n° 6, p. 1126-1138.

Waicukauski, J. A., Lindbloom, E., Rosen, B. K., Iyengar, V.S. . 1987. « Transition Fault Simulation ». *IEEE Design & Test of Computers,* vol. 4, p. 32-38.

Wu, W. X., Hsiao, M. S. 2007. « Mining sequential constraints for pseudo-functional testing ». *Proceedings of the 16th Asian Test Symposium*, p. 19-24.

Yuan, J., Pixley, C., Aziz, A. 2006. « Constrained-random simulation ». In *Constraint-Based Verification*. Springer science + Buisness media.

Zhang, Z., Reddy, S., Pomeranz, I. . 2005. « On Generate Pseudo-Functional Delay Fault Tests for Scan Designs ». In *Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems* p. 215-226.