Vérification semi-formelle et synthèse automatique de PSL vers VHDL

Y. Oddos

▶ **To cite this version:**

Y. Oddos. Vérification semi-formelle et synthèse automatique de PSL vers VHDL. Micro et nanotechnologies/Microélectronique. Université Joseph-Fourier - Grenoble I, 2009. Français. NNT : . tel-00481923

HAL Id: tel-00481923
https://theses.hal.science/tel-00481923

Submitted on 7 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ GRENOBLE I - JOSEPH FOURIER

N° attribuée par la bibliothèque
978-2-84813-144-3

THÈSE

pour obtenir le grade de

DOCTEUR DE L'Université Grenoble I

Spécialité : Micro et Nano Électronique

préparée au laboratoire **TIMA**
dans le cadre de **l'École Doctorale « Électronique, Électrotechnique, Automatique et Traitement du Signal »**

présentée et soutenue publiquement par

Yann Oddos

le 27 Novembre 2009

Titre :

Vérification semi-formelle et synthèse automatique de circuits à partir de spécifications temporelles écrites en PSL

Directeur de thèse : Mme. Dominique Borrione
Co-directeur de thèse : Mme. Katell Morin-Allory

JURY

M. Patrice Quinton	Président
M. Hans Eveking	Rapporteur
M. Bruno Rouzeyre	Rapporteur
M. Gilles Depeyrot	Examinateur
Mme. Dominique Borrione	Examinateur
Mme. Katell Morin-Allory	Examinateur

If you put your mind to it, you can accomplish anything...
Dr. Emmett-L. Brown – Oct. 1985

Remerciements

Quelles que soient les capacités d'un thésard, rien n'est possible sans un bon encadrement. Je remercie vivement Dominique Borrione et Katell Morin-Allory pour m'en avoir fourni un exceptionnel. Au delà de l'aide, des conseils et des discussions, travailler sous votre supervision a été une réelle chance pour apprendre et faire progresser les travaux menés durant ces trois années. Au delà de tout ça, vous m'avez fourni l'opportunité d'apprendre à valoriser mes travaux au travers de colloques et conférences, constituant un aspect aussi important que la recherche elle-même. C'est donc plus qu'un remerciement, mais une profonde reconnaissance que j'ai envers vous deux.

Des travaux très intéressants ont pu être accomplis grâce à une collaboration de 6 mois à l'université de McGill de Montréal. Rien n'aurait été possible sans un support aussi fructueux de la part de Marc Boulé et Zeljko Zilic. Je les remercie pour toutes ces discussions, et tout le temps qu'ils ont passé pour m'intégrer dans leur équipe de recherche. Merci aussi à eux pour m'avoir fourni l'opportunité de tester les longs hivers rigoureux du Canada. Marc, merci pour le guide de survie au Canada pour Européen ! Je l'ai bien étudié et espère bien m'en servir dans le futur.

Je remercie les personnes qui ont permis la soutenance de cette thèse :

- Patrice Quinton : pour avoir accepté d'être président de mon jury malgré sa charge de travail conséquente.
- Hans Eveking et Bruno Rouzeyre : pour avoir relu ma thèse et avoir accepté de venir à Grenoble pour participer à mon jury de thèse.
- Gilles Depeyrot : pour avoir permi une collaboration très enrichissante entre l'équipe VDS et la société Dolphin Integration. Grâce à leur intervention, le transfert de certaines approches décrites ici a été possible. Je le remercie bien sûr pour avoir accepté de participer à mon jury de thèse.

C'est par le travail de groupe et les collaborations constructives que l'avancée des travaux individuels est accrue. Les travaux rapportés ici ont été réalisés en collaboration avec de nombreuses personnes qui ont toutes apporté leur "touche" personelle.

Pour cela je remercie donc : Luca Ferro (spécialiste de tout), Florent Ouchet (grand maître... de la synthèse ASICs), Amr Helmy (pour ses explications sur les NoCs), Renaud Clavel (pour se poser les questions qu'on ne se pose jamais et qui sont pourtant si cruciales), Alexandre Porcher (l'ami des technologies asynchrones), Paul Peronnard (qui m'a tant appris en matière FPGA), Gilles Foucard (pour les journées ASPROM), Jay Tong (pour sa collaboration sur le développement de l'outil MYGEN), Jerôme Oddos (spécialiste en vérification formelle de LiveBox), Jérôme Sester (pour les développements Horus), Clément Deschamps (professionel IHM semi-concrète et prog. JAVA depuis 2001), Laurence Pierre, Jurgen Fischer (l'historien de l'équipe), Cedric Koch-Hofer (pour le plus beau thesis.cls du monde), Eric Gascard (pour les projets RICM et Info-Graphique).

Je remercie tous ceux qui m'ont encadré durant la thèse et plus globalement durant ces années de FAC : Yoann B., Ludo C., Manue D., Lolo la fripouille, Pierre F., Seb F. (le physicien), Floriant F., Youl H., Pierre I., Michael K., Patrice G., Romain G., Sophie G., Djoul J., Jean-Paul & Sophie K., Guillaume M. (Tom Jones), Yoann M., Corine P., Mathieu P., David P., Patou & François P., Richard P., Siouxsie, Seb de Pontivy, Marco Q., l'Adagio, Le Select (notre salle de réunion) et bien sur "Annecy Le Bowl"...

Je remercie vivement toute ma famille (Sylvie, Yves, Jeff, Vincent, Bastien, Pierrot, Nicole...) pour m'avoir tout donné et tant appris. Enfin un merci spécial à Taniuchka pour son aide précieuse sur tous les aspects traduction français/anglais et tout le reste !

Table des matières

Glossaire xix

1 Introduction 1
1.1 Conception de circuits . . . 2
1.2 Vérification de circuits . . . 5
1.2.1 Méthodes classiques . . . 5
1.2.2 Méthodes formelles . . . 5
1.2.2.1 Model-checking . . . 6
1.2.2.2 Preuve de théorèmes . . . 7
1.3 Exemple illustratif . . . 7
1.3.1 Le composant Arbiter_P . . . 7
1.3.2 Le composant CDT . . . 9
1.4 Assertion-Based Verification . . . 11
1.5 Problème . . . 12
1.6 Contributions . . . 13

2 Contexte 15
2.1 Langages de spécification de propriétés . . . 16
2.1.1 Le langage PSL . . . 16
2.1.1.1 Généralités . . . 16
2.1.1.2 Propriétés temporelles . . . 18
2.1.2 Le langage SVA . . . 21
2.1.3 Autres langages . . . 22
2.1.4 Outils existants supportant l'ABV . . . 22
2.2 Vérification à l'aide de moniteurs . . . 26
2.3 Génération de vecteurs de tests . . . 29
2.3.1 Built In Self Test . . . 29
2.3.2 Approches à base de découpage . . . 29
2.3.3 Approches mixtes : techniques formelles/classiques . . . 29
2.3.4 Approches à base de graphes . . . 30
2.3.5 Approches originales . . . 31
2.3.6 Approches à base de propriétés temporelles . . . 32
2.3.7 Bilan . . . 32
2.4 Circuits corrects par construction . . . 33

3 Synthèse de propriétés temporelles 37
3.1 Précisions préliminaires . . . 38
3.2 Moniteurs . . . 39
3.2.1 Moniteurs primitifs . . . 40
3.2.2 Création d'un moniteur complexe . . . 41
3.2.3 Application à l'évaluation de performances . . . 44
3.2.4 Résultats en synthèse . . . 44
3.2.5 Bilan . . . 46
3.3 Générateurs méthode 1 : Horus . . . 46
3.3.1 Générateurs Booléen et FL . . . 46
3.3.1.1 Méthode de construction . . . 46
3.3.1.2 Résultats expérimentaux . . . 50
3.3.2 Générateurs SEREs . . . 51
3.3.2.1 Introduction . . . 51
3.3.2.2 Générateurs pour répétition de séquences . . . 52
3.3.2.3 Traitement de l'opérateur de parallélisation de séquences : && . . . 54
3.3.2.4 Bilan . . . 58
3.3.3 Générateurs et négation de propriétés . . . 59
3.4 Générateurs méthode 2 : MYGEN . . . 61
3.4.1 Concepts préliminaires . . . 61
3.4.1.1 Synthèse d'assertions à l'aide de MBAC . . . 61
3.4.1.2 Des moniteurs vers les générateurs . . . 62
3.4.2 Synthèse du générateur . . . 62
3.4.2.1 Description globale . . . 62
3.4.2.2 MBAC_{plugin} . . . 63
3.4.3 Le Générateur . . . 63
3.4.3.1 Description de l'automate du générateur . . . 63
3.4.3.2 Bloc aléatoire . . . 64
3.4.3.3 L'outil BoolSolve . . . 64
3.4.3.4 Gestion des transitions multiples . . . 65
3.4.4 Résultats expérimentaux . . . 65
3.4.4.1 Analyse de l'outil MYGEN . . . 65
3.4.4.2 Résultats de synthèse sur FPGA . . . 67
3.4.4.3 Comparaisons HORUS/MYGEN . . . 68
3.5 Bloc aléatoire embarqué . . . 69
3.5.1 Le composant LFSR . . . 69
3.5.2 Le composant : automate cellulaire . . . 70
3.5.3 Production de nombres aléatoires sur un intervalle quelconque . . . 73
3.5.3.1 Méthode . . . 73
3.5.3.2 Application aux générateurs pour les vecteurs de bits . . . 74
3.6 Bilan . . . 75

4 Modélisation de l'environnement 77
4.1 Incohérences . . . 78
4.1.1 Cohérence d'une propriété . . . 78
4.1.1.1 Incohérence . . . 78
4.1.1.2 Réalisabilité . . . 78

4.1.2 Cohérence d'une spécification . . . 79
4.1.2.1 Incohérence . . . 79
4.1.2.2 Réalisabilité . . . 80
4.2 Résolution d'incohérences : Description mathématique du problème . . . 80
4.3 Algorithme de résolution pour les systèmes Sys_1 . . . 81
4.3.1 Circuit de Résolution pour des signaux de type Bit : Solver_bit_{sys1} . . . 83
4.3.1.1 Module de détection d'incohérence : Solver_{err} . . . 83
4.3.1.2 Module de résolution : Solver_{sig} . . . 84
4.3.1.3 Le composant global : Solver_bit_{sys1} . . . 84
4.3.2 Circuit de Résolution pour des signaux de type vecteur de Bits : $\text{Solver_vect}_{sys1}$ . . . 85
4.4 Algorithme de résolution pour les systèmes Sys_2 . . . 87
4.4.1 Analyse statique d'un système Sys_2 . . . 87
4.4.2 Solver Sys_2 . . . 89
4.4.3 Résultats expérimentaux . . . 89
4.4.4 Comparaisons Solver_{sys2} et $\text{Solver_vect}_{sys1}$ pour des systèmes Sys_1 . . . 89
4.4.5 Analyse de couverture d'un ensemble de propriétés . . . 90
4.5 Bilan . . . 91

5 Circuits corrects par construction 93
5.1 Introduction . . . 94
5.2 Annotations de propriétés PSL . . . 95
5.2.1 Problème . . . 95
5.2.2 Vers une annotation automatique . . . 96
5.3 Synthèse de spécifications . . . 99
5.3.1 Le générateur-étendu . . . 99
5.3.1.1 Générateur-étendu primitif . . . 99
5.3.1.2 Générateur-étendu complexe . . . 100
5.3.2 Construction du circuit final . . . 102
5.4 Générateurs-étendus à l'aide de MyGen . . . 103
5.4.0.1 Condition booléenne complexe . . . 104
5.4.0.2 Couverture de l'espace de traces valides . . . 105
5.4.1 Exemple . . . 105
5.5 Résultats expérimentaux . . . 106
5.5.1 Comparaison SyntHorus/MyGen . . . 106
5.6 Conclusions . . . 108

6 Preuve des moniteurs, générateurs et générateurs-étendus 109
6.1 Modélisation en PVS . . . 110
6.1.1 Modélisation des composants de vérification . . . 110
6.1.1.1 Simulation Symbolique . . . 110
6.1.1.2 Traduction dans le format PVS . . . 111
6.2 Sémantique des opérateurs temporels de PSL . . . 112
6.3 Modélisation de la sémantique de PSL en PVS . . . 113
6.4 Modélisation de l'équivalence . . . 113
6.4.1 Définition formelle de $\text{H}(\varphi, t_0, T)$ . . . 114
6.4.2 Définition formelle de $\text{V}(\varphi, t_0, T)$ . . . 114
6.4.3 Définition formelle de $\text{P}(\varphi, t_0, T)$ . . . 115

6.4.4 Définition formelle de $S(\varphi, t_0, T)$ . . . 115
6.5 Preuve de correction des moniteurs . . . 115
6.5.1 Cas de base . . . 116
6.5.2 Etape d'induction . . . 117
6.6 Preuve des générateurs et générateurs-étendus . . . 119

7 Mise en oeuvre et applications 121
7.1 La plateforme Horus . . . 122
7.1.1 Le programme psl-bin . . . 123
7.1.2 Intrumentation de circuits à l'aide d'Horus . . . 123
7.1.3 SyntHorus . . . 124
7.2 Exemple illustratif : Arbiter$_P$ et CDT . . . 126
7.2.1 Instrumentation des circuits Arbiter$_P$ et CDT . . . 126
7.2.2 Synthèse du circuit CDT . . . 126
7.3 Cas d'étude 1 : Le contrôleur GenBuf . . . 127
7.3.1 Présentation . . . 127
7.3.2 Spécification du GenBuf . . . 128
7.3.3 Résultat de l'annotation automatique . . . 130
7.3.4 Résultats expérimentaux . . . 131
7.4 Cas d'étude 2 : Le circuit conmax-ip . . . 132
7.4.1 Contexte . . . 132
7.4.1.1 Le projet OpenCores . . . 132
7.4.1.2 La norme Wishbone . . . 133
7.4.1.3 Architecture cross-bar . . . 134
7.4.2 Le contrôleur conmax-ip . . . 136
7.4.3 Vérification du contrôleur . . . 137
7.4.3.1 Vérification de l'initialisation du conmax-ip . . . 137
7.4.3.2 Vérification des connexions . . . 138
7.4.3.3 Vérification des priorités . . . 138
7.4.4 Modélisation de l'environnement . . . 139
7.4.4.1 Synthèse des maîtres à l'aide de générateurs . . . 139
7.4.4.2 Synthèse des esclaves à l'aide de générateurs-étendus . . . 141
7.4.5 Evaluation de performances/Caractérisation du système complet . . . 141
7.4.6 Résultats en synthèse pour l'instrumentation du conmax-ip . . . 143
7.4.6.1 Synthèse pour la vérification du contrôleur . . . 143
7.4.6.2 Synthèse de l'environnement . . . 143
7.4.6.3 Synthèse pour l'évaluation de performances . . . 145
7.4.6.4 Analyses . . . 145
7.4.7 Bilan . . . 146
7.4.8 Synthèse automatique du conmax-ip à partir de sa spécification . . . 146

8 Conclusion et Perspectives 149
8.1 Bilan . . . 150
8.1.1 Instrumentation de circuits . . . 150
8.1.2 Synthèse automatique . . . 151
8.2 Perspectives . . . 152

A Exemple Illustratif : Arbiter$_P$ et composant CDT 155

B Propriétés utilisées pour les expérimentations **157**
B.1 Batterie de propriétés FLs et Booléennes : Bench_FLBool 157
B.2 Batterie de propriétés SEREs : Bench_SERE 158

C Spécification du contrôleur GenBuf **159**
C.1 Spécification du GenBuf donnée dans [BGJ+07a] 159
C.2 Spécification du GenBuf pour SyntHorus 159

D Spécification du contrôleur CONMAX-IP pour 4 maîtres et 3 esclaves **161**

Bibliographie **173**

Index **183**

Publications de l'Auteur **185**

Table des figures

1.1 Flot de conception simplifié . . . 3
1.2 Flot de conception des circuits complexes de type SoC . . . 4
1.3 Le composant $\mathsf{Arbiter}_P$ . . . 8
1.4 Exemple d'arbitrage pour 3 unités . . . 9
1.5 Architecture d'une unité de traitement . . . 10
1.6 Transfert des données de la ressource vers un composant externe par le CDT 10
1.7 Verification : a major cause of first time silicon-failures . . . 11

2.1 Structure du langage PSL . . . 17
2.2 Exemple d'unité de vérification vunit pour le composant $\mathsf{Arbiter}_P$ . . . 18
2.3 Illustration des caractères fort et faible en PSL . . . 19
2.4 Trace respectant la propriété SERE1 . . . 20

3.1 Interface générique pour les moniteurs primitifs . . . 40
3.2 Architecture générique pour les moniteurs primitifs . . . 41
3.3 Architecture du moniteur $A1_{cdt}$ . . . 42
3.4 Trace satisfaisant l'assertion $A1_{cdt}$ . . . 43
3.5 Résultats de synthèse des moniteurs Horus . . . 45
3.6 Comparaisons des moniteurs Horus et MBAC . . . 45
3.7 Interface générique pour les générateurs primitifs . . . 47
3.8 Architecture pour un générateur primitif . . . 48
3.9 Architecture du générateur $H1_{cdt}$ . . . 49
3.10 Résultats en synthèse pour les générateurs Horus . . . 51
3.11 Architecture du générateur pour l'hypothèse $H2_{arbitre}$ . . . 52
3.12 Architecture du générateur pour l'hypothèse $H3_{arbitre}$ . . . 53
3.13 Nouvelle architecture pour les générateurs primitifs booléens . . . 60
3.14 Exemple pour le générateur until . . . 60
3.15 Application de l'algorithme First_Fail pour le moniteur A2b(i) . . . 62
3.16 Synthèse de générateur à l'aide de MyGen . . . 63
3.17 Partie d'automate d'un générateur . . . 65
3.18 Code HDL pour l'automate décrit figure 3.17 . . . 66
3.19 MyGen statistiques : temps d'exécution et taille du code HDL des générateurs 66
3.20 Résultats de synthèse : Surface (gauche) et fréquences (droit) . . . 67
3.21 Comparaison des résultats de synthèse pour Horus et MyGen . . . 68
3.22 $\mathrm{LFSR}_{xor}[2]$ 3 bits de Fibonacci . . . 69

3.23 Séquence produite par un $LFSR_{xor}$[2] 3-bits 70
3.24 Comportement du CA30 avec un seul site initial actif 71
3.25 Comparaison en surface d'un $LFSR_{xor}$ et d'un CA30 72
3.26 Comparaison de la qualité de traces entre un LFSR et différents CA 73

4.1 Instance du composant SOLVER_BIT_{sys1} pour 3 signaux dupliqués 84
4.2 Instance du composant $\text{SOLVER_VECT}_{sys1}$ pour un signal dupliqué 5 fois . . 85
4.3 Process VHDL du composant PRIM_SOLVER 86
4.4 Architecture du SOLVER_{sys2}pour la spécification S1 89

5.1 Flots de conception classique et supporté par la synthèse automatique de spécifications 94
5.2 Architectures des moniteurs, générateurs et générateurs-étendus 100
5.3 Architecture du générateur-étendu complexe pour $F1_{cdt}$ 102
5.4 Résultat de la synthèse de la spécification Spec_cdt : Le contrôleur CDT . . 103
5.5 Automate obtenu pour la propriété A2b(i) 104
5.6 Code VHDL produit pour la condition Trans1 106
5.7 Comparaisons des résultats de synthèse Synthorus/MYGEN 107

6.1 Code VHDL du moniteur always 111
6.2 Fonction PVS pour le calcul du signal *Trigger* du moniteur always 111
6.3 Exemple 5 : trace v satisfaisant la propriété $A1_{cdt}$ 112
6.4 Trace pour laquelle Sem((*Busy* until *Ack*),2,7) est fausse. 114
6.5 Moniteur pour la propriété P1 116
6.6 Théorème d'équivalence pour l'opérateur next_e 117
6.7 Code source du moniteur et générateur-étendu until pour la production de l'opérande gauche (*Trigger*) 119
6.8 Code source du générateur until pour la production de l'opérande gauche (*Trigger*) 119
6.9 Code source du moniteur et générateur-étendu until pour la production du *Pending* 120
6.10 Code source du générateur until pour la production du *Pending* 120

7.1 Temps de production des générateurs pour Horus et MyGen 123
7.2 Instrumentation de circuit par Horus 124
7.3 Onglet Horus pour la synthèse de moniteurs et de générateurs 125
7.4 Flot de syntèse pour SyntHorus 125
7.5 Architecture du contrôleur GenBuf pour 3 émetteurs 128
7.6 Exemple de communication à l'aide du protocole 4 phases 129
7.7 Résultat de synthèse pour le GenBuf : Méthode [BGJ+07a]/Méthode SyntHorus 132
7.8 Interfaces Wishbone maître/esclave 134
7.9 Protocole Wishbone : Ecriture en rafale (burst) 135
7.10 Architecture de type cross-bar 135
7.11 conmax_ip pour 3 maîtres et 4 esclaves 136
7.12 Contrôleur conmax_ip : architecture 137
7.13 Simulation pour la vérification du circuit conmax_ip au reset 138
7.14 Simulation pour la vérification d'une connexion correcte entre Master0 et Slave1 139

7.15 Exemple de simulation pour la propriété PrioMj_Mk . . . 140
7.16 Caractéristiques obtenues pour le circuit conmax_ip . . . 147
7.17 Résultats en synthèse pour le conmax_ip . . . 147
7.18 Comparaison entre le conmax_ip original et celui produit par SyntHorus . . 148

A.1 vunit pour le circuit $\mathsf{Arbiter}_P$ . . . 155
A.2 vunit pour le circuit CDT . . . 156

Liste des tableaux

2.1 Restrictions du sous-ensemble simple PSL_{ss} 21
2.2 Principaux outils de vérification supportant l'ABV 25

3.1 Taps pour le LFSR générique 39
3.2 Type pour chaque moniteur primitif de PSL 40
3.3 Résultats en synthèse pour les générateurs SERE 58
3.4 Signification du couple de ports (*Trigger*, *Pending*) 59
3.5 Symétrie entre moniteurs et générateurs 63
3.6 Règle de calcul numéro 30 71

4.1 Résultats en synthèse pour un composant de résolution gérant 9 duplications de 32 bits 90

7.1 Résultats de l'instrumentation des circuits $\mathsf{Arbiter}_P$ et CDT 126
7.2 Circuit CDT produit avec SyntHorus 127
7.3 Circuit CDT produit avec MyGen 127
7.4 Annotation automatique de la propriété MyG1 131
7.5 Résultats de synthèse pour les moniteurs 143
7.6 Résultats de synthèse pour les générateurs 143
7.7 Résultats de synthèse pour les générateurs-étendus 144
7.8 Résultats de synthèse pour l'évaluation de performances 145

Glossaire

A
ABA *Alternating Büchi Automaton*
ACM *Association for Computer Machinery*
AEFD *Automate d'Etats Finis Déterministe*
ASP *Answer Set Programming*
ATPG *Automatic Test Pattern Generator*

B
BDD *Binary-Decision Diagram*
BIST *Built In Self Test*
BSV *BlueSpec SystemVerilog*

C
CFG *Control Flow Graph*
CTL *Computational Tree Logic*

D
DFA *Deterministic Finite Automaton*
DTS

F
FBDD/FreeBDD *Free Binary Decision Diagram*
FIFO *First In First Out*
FPGA *Field Programmable Gate Array*
FSM *Finite State Machine*

G
GaLs *Globally Asynchronous Locally Synchronous*

H
HDL *Hardware Description Language*
HLDD *Hardware Level Decision Diagram*

I
IEEE *Institute of Electrical and Electronics Engineers*
IOLTS *Input Output Labelled Transition System*
IPs *Intellectual Properties*

L
LTL *Linear Temporal Logic*

N
NBA *Non-deterministic Büchi Automaton*
NFA *Non-deterministic Finite Automaton*
NIOS
NoC *Network on Chip*

O
OBE *Optionnal Branching Extension*
OVA *Open Vera Assertion Language*
OVL *Open Verification Library*

P
PSL *Property Specification Language*
PSL_{ss} *Sous ensemble simple de PSL*
PVS *PV System*

Q

R
ROBDD *Reduced Ordered Binary Decision Diagram*
RTPG *Random Test Pattern Generator*

S
SERE *Sequential Extended Regular Expression*
SoC *System on Chip*
SVA *SystemVerilog Assertions*

V
VHDL *Very high speed integrated circuit Hardware Description Language*

Chapitre 1

Introduction

Sommaire

1.1 Conception de circuits **2**

1.2 Vérification de circuits **5**

1.2.1 Méthodes classiques 5

1.2.2 Méthodes formelles 5

1.3 Exemple illustratif **7**

1.3.1 Le composant Arbiter_P 7

1.3.2 Le composant CDT 9

1.4 Assertion-Based Verification **11**

1.5 Problème **12**

1.6 Contributions **13**

L'électronique a connu une telle croissance et une telle diffusion au niveau du grand public que son usage est devenu indispensable aujourd'hui. Né dans le but de pouvoir effectuer efficacement des calculs complexes, l'utilisation du composant électroniques a connu une diversification étonnante qui lui a permis d'infiltrer profondément notre quotidien. La *dépendance* de l'être humain vis-à-vis des systèmes électronique est devenue non négligeable et s'accentue au fil du temps.

De ce phénomène de dépendance, émerge tout naturellement la nécessité de garantir une fonctionnalité correcte des circuits produits. À l'heure actuelle, plus d'une quarantaine de processeurs sont embarqués à l'intérieur d'une voiture classique. Ceux-ci contrôlent des points clés tels que la direction ou le freinage. L'utilisateur est donc dépendant de ces processeurs et la sécurité ne peut être assurée à 100% que si la fonctionnalité correcte des systèmes embarqués est-elle aussi garantie.

Or, comme nous le verrons tout au long des travaux rapportés ici, garantir la fonctionnalité correcte d'un composant est un problème très difficile, qui ne possède toujours pas de solution satisfaisante aujourd'hui. La majorité des circuits actuellement sur le marché ne sont pas testés à 100% et peuvent contenir des bugs. Ceci remet en question la confiance que l'utilisateur peut porter à des systèmes critiques assurant des fonctions clés pouvant mettre en jeu des vies humaines.

1.1 Conception de circuits

Ces dernières années, la complexité des circuits électroniques a explosé, notamment grâce à l'intégration de plusieurs composants différents sur une même puce : les systèmes sur puces (SoC). Alors que jusqu'à quelques années en arrière, l'augmentation des performances était basée sur la diminution des finesses de gravure et le découpage des fonctionnalités en pipelines complexes, cette méthode a récemment trouvée ses limites. Aujourd'hui, c'est l'architecture des circuits qui évolue pour continuer d'améliorer les circuits conçus.

La conception de nouvelles architectures est actuellement axée sur la décomposition de circuits complexes en plusieurs composants interconnectés directement via un bus ou un réseau sur la puce. Ceci permet non seulement une parallélisation des tâches, mais aussi d'utiliser différents cœurs spécifiques traitant efficacement certains types de calcul. Par exemple, le processeur GT200 de Nvidia (1,4 milliards de transistors répartis sur 600mm^2) contient plusieurs centaines de processeurs graphiques GPU, des mémoires sur la puce et des processeurs de contrôle de flux, tous interconnectés entre-eux par des bus de différents types[1].

Le développement de circuits aussi complexes nécessite des outils et des méthodologies de conception adaptés. La figure 1.1 illustre le principe d'un flot de conception de circuit. Tout d'abord, une exploration d'architecture est effectuée. Celle-ci analyse de nombreux points différents :

- partitionnement matériel/logiciel
- consommation
- fréquence de fonctionnement, surface et bande passante
- température

[1] *Nvidia dévoile la gamme GeForce GTX 200* 16/06/2008 disponible sur : http ://www.silicon.fr/fr/news/2008/06/16/nvidia_devoile_la_gamme_geforce_gtx_200_

- coût du système

Une fois l'architecture définie, la description des parties matérielles et l'implémentation des parties logicielles est effectuée. Une phase de vérification est nécessaire pour s'assurer que la fonctionnalité du circuit respecte la spécification. Si la vérification réussit, une étape de prototypage permet de tester le circuit en conditions réelles de fonctionnement. Finalement la liste de portes est envoyée au fondeur pour produire le circuit en série.

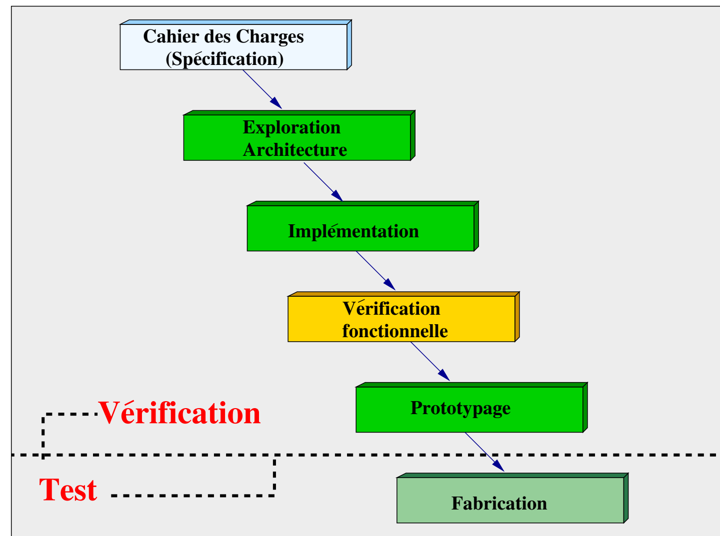

Figure 1.1 – Flot de conception simplifié

La maîtrise de la complexité des circuits est dûe à deux facteurs : l'amélioration des outils de conception et une montée dans les niveaux d'abstraction. Malheureusement, les outils de vérification n'ont pas suivi cette tendance et le fossé entre outils de conception et de vérification ne cesse de croître.

Les flots de conception se sont fortement complexifiés et spécialisés selon les outils utilisés afin d'enrayer cette tendance. La figure 1.2 est une représentation généralisée d'un flot de conception pour systèmes sur puce. Deux facteurs importants permettent une meilleure maîtrise de la vérification :

- conception modulaire : chaque composant d'un système sur puce est développé de manière indépendante. Certains modules peuvent être disponibles sur le marché et directement réutilisés. Les modules ont une complexité largement plus faible que le circuit global, ce qui permet une vérification plus facile.
- couplage de la vérification et de la conception en répartissant des étapes de vérification tout au long du flot.

Une différenciation doit être faite entre test et vérification. Le test est une technique appliquée après la fabrication pour s'assurer que chaque exemplaire du circuit est fonctionnellement correct. Ceci permet de détecter si durant le processus de fabrication aucune erreur physique n'est présente dans le circuit : court-circuit, transistor défectueux etc.

La vérification s'effectue avant l'étape de fabrication, sur la description du circuit. Elle permet de vérifier que d'étape en étape, le long du flot de conception, la fonctionnalité du circuit est préservée et correspond à la spécification de départ.

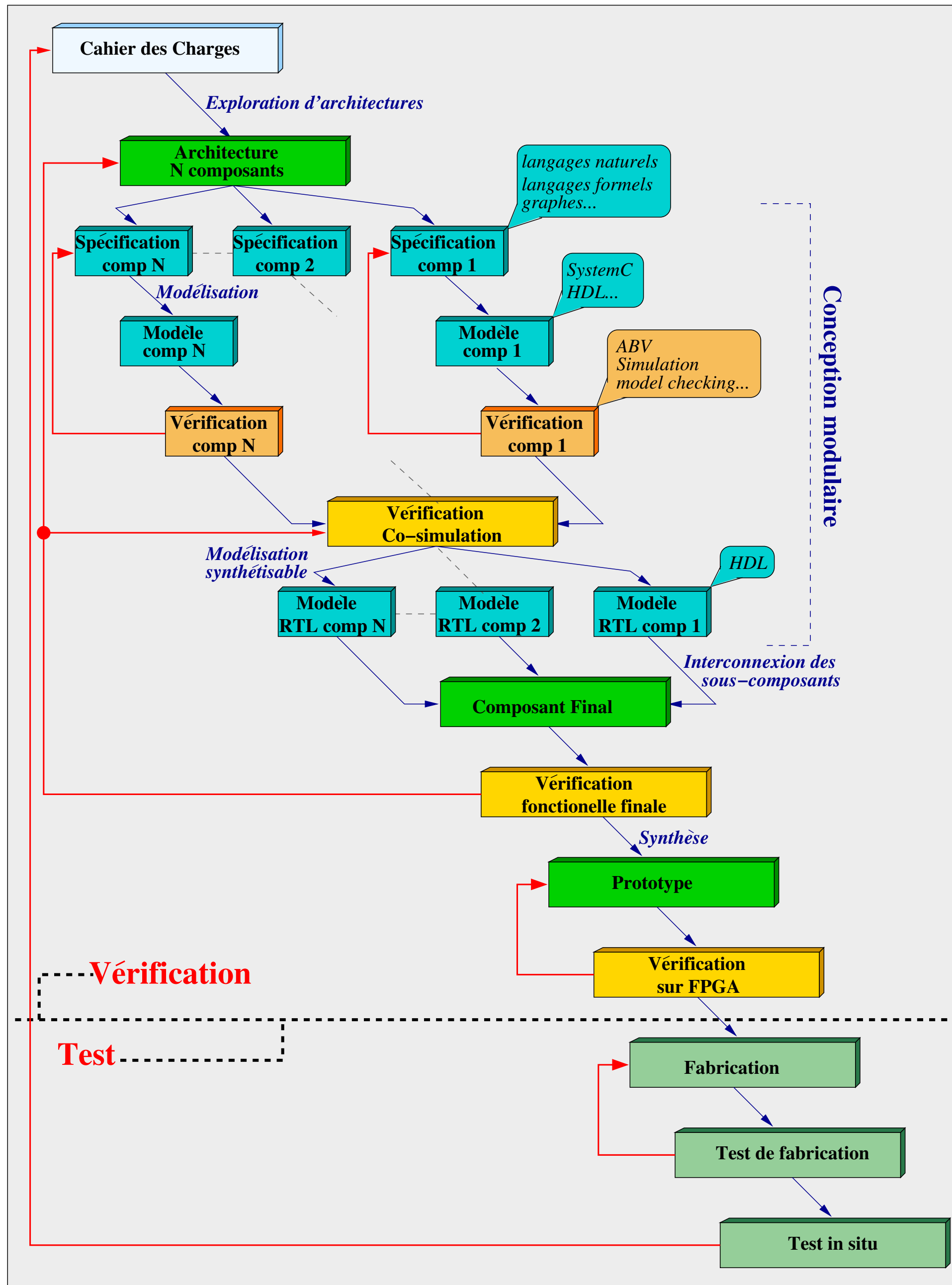

FIGURE 1.2 – Flot de conception des circuits complexes de type SoC

1.2 Vérification de circuits

De nombreuses méthodes sont disponibles pour la vérification. Celles-ci sont classées en deux groupes : les méthodes classiques et formelles.

1.2.1 Méthodes classiques

Depuis l'existence des langages de description de matériel, la technique de vérification la plus répandue a été la simulation. Applicable tout au long du flot de conception, elle permet une vérification efficace et simple à mettre en œuvre. Mais la complexité des composants actuels est telle, que la simulation requiert un temps et une puissance de calcul considérables. Synopsys rapporte des temps de simulation de plusieurs jours pour un système sur puce standard. Le coût de la simulation devient donc critique et des alternatives doivent être trouvées.

Une réponse a été apportée grâce au prototypage sur des circuits reprogrammables de type FPGA (Field Programmable Gate Array). Ils permettent de tester une version prototype matérielle de la description HDL, ce qui améliore considérablement la rapidité du test comparée à une simulation classique. La simplicité et l'efficacité de cette approche est largement reconnue depuis plusieurs années et utilisées dans de nombreux flots de conception.

Actuellement, les industriels utilisent des approches hybrides mixant simulation, émulation et/ou prototypage. Les parties critiques sont portées sur le FPGA alors que les composants plus simples sont simulés directement.

Mais ces méthodes sont incapables de tester les circuits actuels de manière exhaustive, car la complexité de la vérification est exponentielle par rapport à la complexité des systèmes. Ainsi, une simple mémoire RAM de 2Mb possède 2^{2048} états possibles. Si la vérification de chaque état de la RAM dure 1 nanoseconde, le test exhaustif requiert plusieurs millions années. Les méthodes de vérification classique sont incapables de garantir à 100% la fonctionnalité d'un circuit, mais peuvent seulement fournir une évaluation du niveau de fiabilité du composant. La vérification de systèmes critiques, requiert d'autres types de vérification.

1.2.2 Méthodes formelles

Basées sur des théories mathématiques formellement définies, les techniques de vérification formelles permettent la vérification exhaustive de circuits. Mais ceci a un prix : la puissance de calcul nécessaire explose avec la complexité du circuit, et ces techniques ne sont généralement pas automatiques. Depuis une dizaine d'années, de gros efforts de recherche sont effectués afin d'améliorer l'efficacité et la facilité d'utilisation de telles méthodes.

Ces techniques se divisent en quatre groupes principaux :

- **Equivalence-checking** : utilise un circuit, ou modèle de référence et le circuit à vérifier. Les sorties sont combinées à l'aide d'une porte XOR. Comme les deux circuits doivent avoir le même comportement, il suffit de vérifier que les sorties finales ne sont jamais actives. Cette approche peut aussi être effectuée à l'aide de techniques formelles à base de BDDs ou de SAT [PK00, KPKG02].

- **Simulation symbolique** : effectue la simulation sur des ensembles de valeurs abstraites au lieu d'utiliser des valeurs explicites [Ber06]. Tester un additionneur simple d'entrées A et B et de sortie C, avec une simulation classique requiert un test de toutes les valeurs possibles, ce qui est impossible en pratique. Avec la simulation symbolique, les deux entrées sont spécifiées de manières ensembliste : A∈ [0..10], B∈ [0..10]. L'équation symbolique du circuit est calculée et le résultat obtenu, si le circuit est correct, sera : C∈ [0..20]=A+B.
- **Model-checking** : transforme la description du circuit en automate et vérifie qu'un ensemble de propriétés est vérifié en parcourant tous les chemins possibles de l'automate.
- **Preuve de théorème** : vérifie des propriétés mathématiques sur un modèle du circuit.

1.2.2.1 Model-checking

Les premiers travaux sur le model checking de formules de logique temporelle ont été menés par Edmund M. Clarke et E. Allen Emerson en 1981, ainsi que par Jean-Pierre Queille et Joseph Sifakis en 1982 [CES86]. Cette technique consiste à créer un modèle du circuit à vérifier sous forme d'automate d'états fini. Un ensemble de propriétés que doit vérifier le circuit est définit. Celles-ci sont souvent issues de la spécification. L'outil de model-checking parcoure alors toutes les exécutions possibles du systèmes en traversant tous les chemins de l'automate à partir de l'état initial et vérifie que la propriété est respectée dans tous les états atteignables [Sch99].

Une propriété temporelle est une propriété se déroulant à travers le temps. La propriété suivante peut être exprimée : *toute requête est satisfaite au pire 30 cycles d'horloge après son envoi.* L'écriture de propriétés temporelles s'effectue dans un cadre formel : la logique temporelle. Elle se décline en deux ensembles : (Computational Tree Logic) et (Linear Temporal Logic). Un historique complet de ces logiques temporelles est donné dans [Var06].

La première est utilisé pour exprimer des propriétés sur plusieurs, voire toutes, les exécutions possibles du circuit à vérifier. Elle peut être utilisée uniquement par des outils de vérification statique, c'est à dire qui vérifient le modèle du circuit sans exécuter explicitement celui-ci, mais en parcourant un modèle de ce dernier.

La logique est linéaire et permet d'exprimer des propriétés relatives à une seule exécution du circuit, l'exécution courante. Il est ainsi possible de vérifier ces propriétés à la fois de manière statique, mais aussi dynamique, le long de l'exécution du système.

Comme le parcours d'un automate explose avec le nombre d'états de celui-ci, de nombreuses techniques dérivées du model-checking classique ont vu le jour. Le model-checking borné permet de vérifier exhaustivement une partie de l'automate en restreignant le nombre de cycles analysés [BCC+03]. Le model-checking dirigé permet de sélectionner les chemins à parcourir et d'orienter l'exécution à vérifier. Le model-checking symbolique utilise les concepts de la simulation symbolique en utilisant une abstraction de l'automate initial où les états sont regroupés en ensembles abstraits d'états [McM93].

De nombreux outils de model-checking sont actuellement disponibles (*cf.* Section 2.1.4). Ils sont de plus en plus utilisés afin de vérifier certaines parties critiques d'un circuit, s'affranchissant ainsi de la complexité globale du circuit.

1.2.2.2 Preuve de théorèmes

Contrairement à la preuve de propriétés, la preuve de théorèmes n'est pas totalement automatique et elle nécessite une intervention humaine plus ou moins poussée selon les outils utilisés. La preuve de théorèmes utilise un modèle de circuit, un ensemble d'axiomes et effectue des preuves sur une modélisation du circuit. Si une conversion de la description HDL en modèle formel peut être automatiquement obtenue, alors prouver le modèle équivaut à prouver la description du circuit.

Une des principales force de cette technique est sa capacité à décrire le comportement de circuits à différents niveaux d'abstraction. Ceci permet donc de réduire le fossé entre le haut niveau apporté par une spécification et une vérification d'un circuit qui peut être décrite à des niveaux d'abstractions croissant. En opposition au model-checking, cette technique peut traiter des systèmes ayant un espace d'états infini.

Un des grands problèmes de cette technique est le fait qu'elle suscite un effort considérable du concepteur autant au niveau de la spécification de chaque composant qu'au moment de la vérification lorsqu'il faut guider le prouveur de théorèmes.

Plusieurs prouveurs de théorèmes peuvent être utilisés. Ils ont chacun leurs propres caractéristiques. Nous donnons trois exemples :

- ACL2(**Applicative Common Lisp 2**) : Cet outil fournit des règles de déduction et des stratégies pour les appliquer dans un ordre fixé. Il utilise une logique du premier ordre avec induction. Cet outil permet de conserver dans un fichier appelé "livre", les définitions, les lemmes et les théorèmes. Ceci facilite grandement la réutilisation de théories et ainsi facilite le développement. Il convient de noter que cet outil n'est pas typé. L'outil est basé sur LISP, ce qui rend sa logique exécutable.
- HOL(**Higher Order Logic**) : Cet outil utilise une logique d'ordre supérieure. Comme pour ACL2, il est possible de transformer une règle prouvée en rgle de base afin de la réutiliser plus tard. L'interface utilisateur est constituée d'un méta langage fonctionnel `ML`. L'utilisateur guide le système en choisissant des stratégies de démonstration durant le processus de vérification. Contrairement à ACL2, la logique de HOL est typée et n'est pas exécutable.
- PVS(**Prototype Verification System**) : Cet outil, obtenu d'après les bases conceptuelles de HOL et de ACL2, est basé sur une logique d'ordre supérieure fortement typée et contient un système de types très riche. Comme pour HOL sa logique n'est pas exécutable. Cet outil inclut de plus un vérificateur de propriétés pour des automates d'états finis où les propriétés temporelles sont exprimées à l'aide du μ-calcul.

Alors que jusqu'à maintenant les méthodes formelles étaient utilisées comme complément aux méthodes de vérification classiques. Kaivola *et al.* ont montré qu'avec les outils actuels, il est possible d'utiliser les techniques formelles pour assurer la totalité de la vérification de systèmes complexes tels que le dernier processeur Intel Core i7 [KGN+09].

1.3 Exemple illustratif

1.3.1 Le composant Arbiter$_P$

Cette section présente le composant VHDL synthétisable nommé Arbiter$_P$ permettant à plusieurs unités de traitement, d'accéder de manière exclusive à une unique ressource. Ses entrées sont :

- *clk* et *reset_n* : pour synchroniser le composant.
- *use* : vecteur de P bits. Chacune des entrées use_i de ce vecteur est active lorsque l'unité I correspondante signale l'utilisation de la ressource.
- *ask* : vecteur de P bits. Chacune des entrées ask_i de ce vecteur est active lorsque l'unité I correspondante signale une demande d'accès à la ressource.

L'arbitre possède une sortie de P bits appelée *grant*. Chacun des bits $grant_i$ composant ce vecteur est utilisé par l'arbitre pour indiquer l'attribution de la ressource à l'unité I correspondante.

Cet arbitre est composé de blocs logiques élémentaires appelés `Slices`. Pour arbitrer l'accès à une ressource pour P unités, l'arbitre contient P `Slices`. La figure 1.3 illustre la structure du composant $\mathsf{Arbiter}_P$.

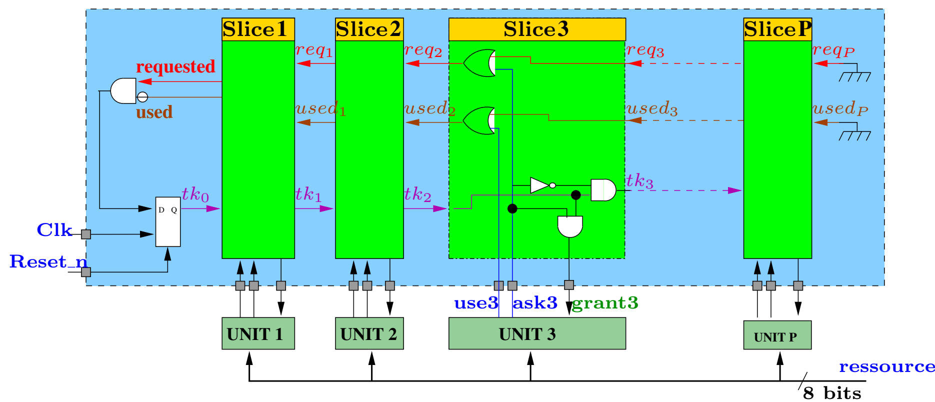

FIGURE 1.3 – Le composant $\mathsf{Arbiter}_P$

Supposons qu'une seule unité i demande la ressource en activant son port de sortie ask_i (i.e. ask_i ='1'). La demande se transmet au travers des signaux req_i puisque les portes OR conservent la valeur du signal. Si la ressource est libre, le signal *used* est inactif. Alors au prochain cycle, l'arbitre produit un jeton en plaçant le signal tk_0 à '1'. Ceci active la sortie $grant_i$; la i-ème unité reçoit alors un jeton. La possession d'un jeton permet à une unité de commencer immédiatement à accéder à la ressource. L'unité active son port use_i jusqu'à la fin du traitement. Ce processus est illustré sur la figure 1.4. La première unité demande la ressource au cycle ♯1, l'obtient au cycle suivant, et la relâche au cycle ♯6.

Si plusieurs unités demandent simultanément un accès à la ressource, trois phénomènes se produisent à l'intérieur de $\mathsf{Arbiter}_P$:

- l'unité de numéro le plus faible sera la plus prioritaire. Dans chaque Slice, l'activation du signal ask_i désactive la transmission du jeton pour les unités suivantes. Le traitement des unités n'est pas égalitaire et des cas de famine peuvent se produire dans le cas où les premières unités accèdent intensivement à la ressource partagée.
- Le signal *requested* reste actif puisque la chaîne de signaux $req_1,\ldots,req_j$ est maintenue à 1, dès qu'une requête ask_j au moins, est effectuée.
- un jeton est produit si et seulement si la ressource n'est allouée à aucune unité au cycle courant. Sinon, au moins une unité i a son port use_i actif et la chaîne des signaux $used_1,\ldots,used_i$ est maintenue à '1'. Ceci place obligatoirement le signal tk_0 à '0' à partir du cycle suivant. Aucun jeton n'est produit tant que la ressource n'est pas libérée.

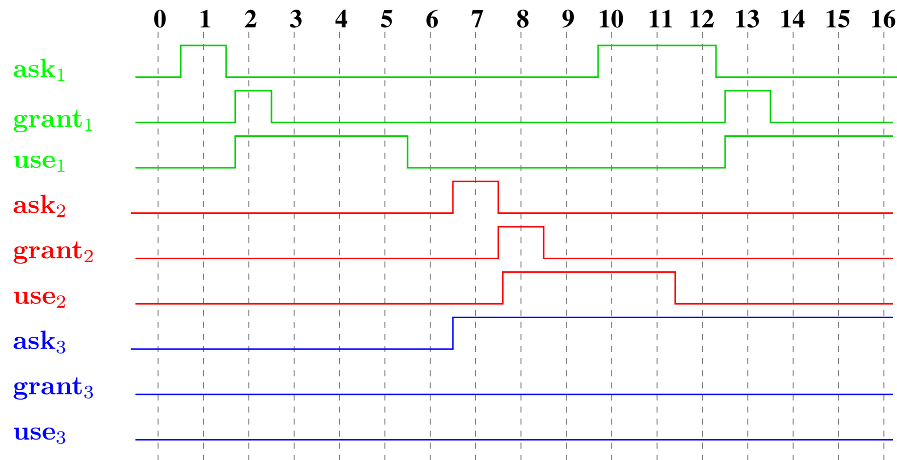

FIGURE 1.4 – Exemple d'arbitrage pour 3 unités

La demande simultanée est illustrée sur la figure 1.4. Les unités 2 et 3 demandent la ressource au cycle ♯7. L'unité 2 étant la plus prioritaire, elle obtient un jeton au cycle ♯8. Alors que l'unité 2 utilise la ressource, l'unité 1 demande une fois de plus un accès à la ressource. Etant prioritaire, elle l'obtient au cycle ♯13. La ressource 3 n'obtient jamais la ressource sur cet exemple.

1.3.2 Le composant CDT

Une unité de traitement accède à une ressource et transmet son contenu à un composant extérieur. Pour cela, chaque circuit UNIT est composé de deux blocs : Interface Arbiter_P, et CDT. L'architecture du circuit UNIT est illustrée figure 1.5.

L'interface Arbiter_P sert à communiquer avec l'arbitre pour effectuer les demandes d'accès à la ressource. Ce composant contrôle aussi le composant CDT.

Le contrôleur CDT est une interface de communication permettant d'envoyer des données reçues sur le port *ressource* à l'aide du port *data*. Le transfert se termine lorsque le signal d'acquittement *Ack* provenant du composant externe est reçu. Le contrôleur requiert 4 cycles pour initialiser un nouveau transfert.

Le chronogramme de la figure 1.6 illustre les étapes d'un transfert de données de la ressource vers un composant extérieur via le CDT. Celui-ci est ré-initialisé au cycle ♯0, ce qui place toutes ses sorties à '0'. L'unité de traitement demande la ressource et l'obtient au cycle ♯4. L'interface transmet cette information au CDTen plaçant son port d'entrée *Req* à '1'. À partir de ce cycle, les données de la ressource sont disponibles sur le port d'entrée *Ressource* du CDT. Le signal *Send* est mis à '1' pour indiquer au composant que les données de la ressource sont transmises sur le port de sortie *Data* du CDT.

Au cycle ♯10, un acquittement est reçu et le transfert est terminé. Le signal *Send* repasse à '0' ainsi que le port *Data*. Le CDT reste occupé durant encore 4 cycles, ce qui est indiqué au composant externe via le port *Busy*. Aucun transfert ne peut être initié entre les cycles ♯10 et ♯14. Le signal *Use* de l'interface signale à l'arbitre les cycles où la ressource est occupée. Après chaque cycle de transfert, 4 cycles sont nécessaires pour traiter les données en interne, c'est pourquoi le signal *Busy* reste actif jusqu'au cycle ♯14.

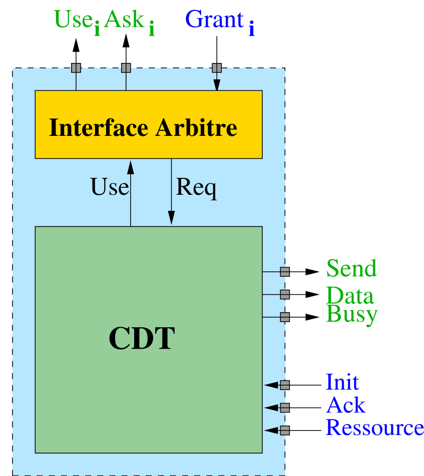

FIGURE 1.5 – Architecture d'une unité de traitement

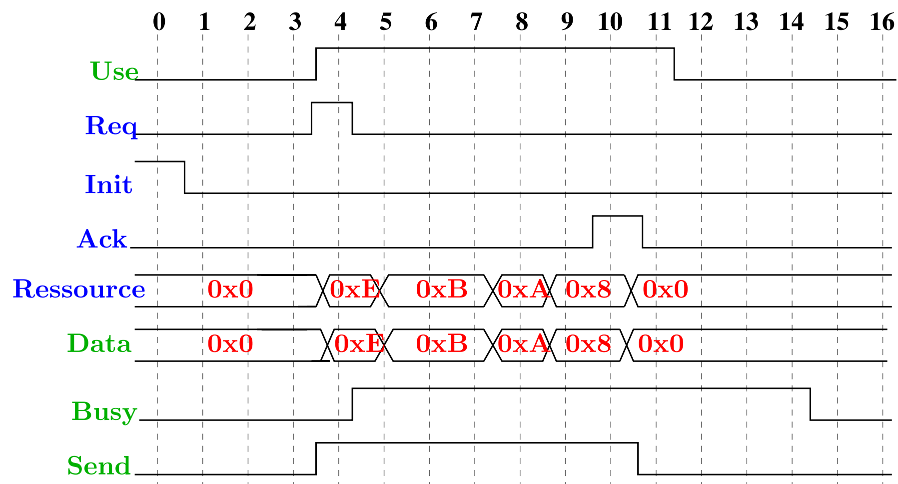

FIGURE 1.6 – Transfert des données de la ressource vers un composant externe par le CDT

1.4 Assertion-Based Verification

La découverte d'erreurs au plus tôt dans le flot de conception est primordiale. Le coût d'une erreur explose avec l'avancement de la conception du circuit. Ces dernières années, une méthode à la popularité grandissante a fait son apparition dans les domaines académiques et industriels : la **vérification à base d'assertions** (ABV) [FKL03]. Une assertion est une description concise d'un comportement complexe que le circuit doit satisfaire.

Cette approche, qui consiste à supporter la vérification à l'aide d'assertions, a largement prouvé son efficacité (figure 1.7), et la plupart des industriels ont aujourd'hui adopté des flots de conception basés sur l'ABV. Son utilisation peut augmenter l'efficacité de la

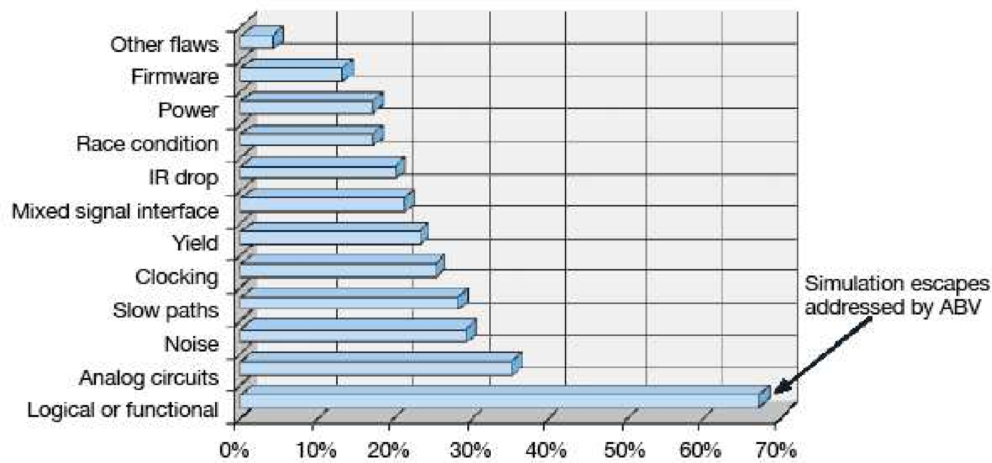

FIGURE 1.7 – Verification : a major cause of first time silicon-failures

phase de test jusqu'à 50%. Ceci est possible car l'ABV couple directement des éléments de vérification aux éléments de conception. Alors que dans une approche classique, lorsqu'une erreur est détectée, une analyse coûteuse doit être effectuée pour remonter à la cause du dysfonctionnement, l'utilisation d'assertions tout au long du circuit permet de capturer les erreurs au plus près de leur source. Le gain de temps au niveau du debug est considérable.

De plus, écrire ces propriétés lors de l'implémentation permet aussi d'approfondir la réflexion sur la fonctionnalité du circuit, et donc une première focalisation du développeur sur ce qu'il est en passe d'implémenter.

L'ABV utilise deux types de propriétés :

– **Hypothèse**[2] : décrit l'environnement du circuit à tester. Plus précisément, les hypothèses sont des propriétés exprimant des contraintes sur les entrées du circuit à vérifier. Dans le cas du composant $\mathsf{Arbiter}_P$, l'hypothèse suivante peut être faite :

  Hypothèse $\mathbf{H1}_{arbitre}$: *une unité ne demande jamais l'accès à la ressource si elle l'utilise déjà.*

– **Assertion** : décrit les comportements que le composant doit assurer. Ces propriétés portent sur les signaux internes ou sur les sorties du circuit à vérifier. Dans le cas du contrôleur $\mathsf{Arbiter}_P$, l'assertion suivante peut être effectuée :

[2] En anglais : assumption

Assertion $\mathbf{A1}_{arbitre}$: *toute demande d'accès à une ressource est satisfaite.*

Ceci permet d'obtenir un environnement complet de vérification où les comportements du circuit sous test sont analysés lorsque l'environnement respecte les protocoles de communication avec celui-ci.

De plus, l'ABV définit un cadre de travail appelé **Assume-Guarantee Paradigm** qui permet une vérification hiérarchique de systèmes complexes. Le principe consiste à vérifier au mieux un composant à l'aide de propriétés. Celui-ci est ensuite validé et il peut alors être inclus dans un composant plus complexe. La vérification se réduit alors aux protocoles de communication utilisés entre l'environnement et le circuit pré-vérifié.

L'utilisation de ces propriétés peut s'effectuer aussi bien en vérification statique (model-checking) que dynamique (simulation, émulation, prototypage). Dans le dernier cas, un concept fondamental est utilisé par tous les outils de vérification : la synthèse de propriétés.

D'une part les assertions sont transformées en **moniteurs**. Ces composants analysent les comportements du circuit à vérifier et reportent une erreur si ceux-ci ne vérifient pas les propriétés associées. Les moniteurs peuvent être logiciels et exécutés en parallèle de la simulation par un moteur de vérification ; ou matériels et connectés physiquement au circuit à vérifier.

D'autre part, les hypothèses sont synthétisées en **générateurs**. Un générateur produit des séquences de signaux correspondant à l'hypothèse de départ. L'idée consiste alors à définir l'ensemble des contraintes que doit satisfaire l'environnement sous forme d'hypothèses. La synthèse de celles-ci fournit un modèle d'environnement permettant de vérifier le circuit en conditions normales de fonctionnement.

1.5 Problème

Avec l'avènement des systèmes sur puces et le développement de nouvelles architectures permettant l'intégration de centaines de composants hétérogènes au sein d'un même circuit physique, les méthodes de test traditionnelles ne sont plus efficaces. Alors que le temps de mise sur le marché doit sans cesse diminuer et que la complexité des composants explose, de nouvelles méthodes doivent être mises en place pour conserver la continuité de l'évolution technologique telle que nous l'avons connue ces 20 dernières années.

De plus, le test est actuellement non exhaustif et la garantie d'une fonctionnalité correcte à 100% est souvent sacrifiée pour mettre le circuit sur le marché le plus rapidement possible. Or pour des circuits embarqués dans des outils de médecine, des avions, ou comme c'est le cas maintenant, des voitures, la garantie d'une fonctionnalité correcte se doit d'être totale puisque des vies humaines sont en jeu.

L'efficacité de la vérification fonctionnelle est la clé du succés dans le développement d'un nouveau produit, puisque plus de la moitié du temps et du coût de conception sont consommés par cette étape. Aucune solution n'est disponible aujourd'hui, ni dans les laboratoires, ni dans l'industrie.

Nous proposons ici deux approches puisant leurs origines aussi bien dans le domaine de la vérification formelle que dans les techniques classiques de vérification de circuits. Celles-ci fournissent des outils simplifiant la phase de test pour l'ingénieur de test et augmentent significativement la qualité de la vérification comparée à des méthodes classiques.

1.6 Contributions

Les travaux effectués dans cette thèse s'articulent autour de deux thèmes principaux : la génération de vecteurs de test, et la synthèse automatique de circuits à partir de spécifications temporelles écrites en PSL.

La vérification fonctionnelle de descriptions matérielles de circuits est une étape clé et coûteuse du flot de conception d'un circuit. Celle-ci se décompose en deux parties : la génération de scénarios de test, l'analyse du comportement du circuit sous test. Nous proposons une approche synthétisant automatiquement des hypothèses PSL ou SVA en générateurs (*cf.* chapitre 3). Ceux-ci produisent des vecteurs de test conformes aux propriétés associées. Les propriétés temporelles permettent de décrire simplement des scénarios complexes. L'approche que nous avons définie permet de transformer un ensemble de propriétés en un composant matériel simple et rapide permettant de produire des vecteurs de test respectant les propriétés temporelles de départ. Il est ainsi possible d'aller au delà de la modélisation de scénario, et d'atteindre un certain degré d'automatisation du test-bench en modélisant l'environnement lui-même.

Les circuits ainsi synthétisés sont appelés générateurs. La synthèse de ces composants est indépendante de la complexité du circuit à traiter. La construction est modulaire, résultant en une méthode très efficace (quelques secondes pour produire des générateurs complexes) et simple à appréhender. Basée sur une approche existante pour produire des moniteurs [MAB05], les générateurs ont été prouvés corrects par construction à l'aide du prouveur de théorèmes PVS(*cf.* chapitre 6).

La modélisation d'un environnement s'obtient en combinant différents générateurs. Comme plusieurs générateurs peuvent piloter un même ensemble de signaux, une résolution doit être effectuée. Un ensemble de composants de résolution a été construit dans ce but. Ceux-ci permettent de résoudre dynamiquement la valeur d'un signal possédant plusieurs sources.

L'aspect certainement le plus intéressant réside dans la méthode qui a été développée pour synthétiser efficacement une description HDL d'un circuit à partir de sa spécification donnée sous forme de propriétés temporelles écrites en PSL (*cf.* chapitre 5). Cette technique consistant à synthétiser une spécification directement en une description de niveau RTL intéresse grandement de nombreux chercheurs depuis plusieurs années puisqu'elle permettrait de supprimer deux étapes extrêmement coûteuse du flot de conception : implémentation de la spécification et vérification fonctionnelle. Malheureusement les techniques de synthèse de spécifications explosent avec la taille de la spécification et rendent le procédé impraticable pour des circuits réels.

La méthode de synthèse proposée ici possède une complexité linéaire en fonction du nombre d'opérateurs contenus dans la spécification. Ceci permet une application efficace de notre approche, même sur des circuits complexes. Notre technique utilise les concepts de moniteurs et de générateurs. En combinant ceux-ci de manière spécifique, il est possible de produire en quelques secondes des circuits efficaces, à partir de centaines de propriétés. Non seulement tout le sous ensemble simple de PSL est pris en compte, mais des circuits complexes peuvent être traités par cette approche.

Tout ceci est supporté par la plateforme Horus (*cf.* chapitre 7.1). Horus est un outil permettant la synthèse de moniteurs, de générateurs et de circuits. Une partie du développement de cette plateforme a été effectué dans le cadre de cette thèse. La force d'Horus réside dans l'efficacité de la synthèse (quelques secondes pour des spécifications complexes), et dans les outils qu'il fournit pour l'aide au debug : connexion assistée des

moniteurs et générateurs au circuit sous test ; observation de signaux internes ; encapsulation du circuit instrumenté et création d'un outil fournissant un rapport de test.

Plusieurs cas d'études complexes sont étudiés dans le chapitre 7, pour illustrer la méthode de synthèse de spécifications. Ces cas d'études montrent clairement l'efficacité de notre approche et permettent des comparaisons avec plusieurs outils pré-existants.

Enfin, le chapitre 8 dresse le bilan des travaux effectués. Il donne aussi les principales lignes directrices des travaux en cours afin de compléter l'approche décrite ici. Plusieurs champs d'investigation sont passés en revue afin de repousser les limites de la vérification de circuits à l'aide de l'ABV.

Chapitre 2

Contexte

Sommaire

2.1 Langages de spécification de propriétés **16**
- 2.1.1 Le langage PSL 16
- 2.1.2 Le langage SVA 21
- 2.1.3 Autres langages 22
- 2.1.4 Outils existants supportant l'ABV 22

2.2 Vérification à l'aide de moniteurs **26**

2.3 Génération de vecteurs de tests **29**
- 2.3.1 Built In Self Test 29
- 2.3.2 Approches à base de découpage 29
- 2.3.3 Approches mixtes : techniques formelles/classiques 29
- 2.3.4 Approches à base de graphes 30
- 2.3.5 Approches originales 31
- 2.3.6 Approches à base de propriétés temporelles 32
- 2.3.7 Bilan 32

2.4 Circuits corrects par construction **33**

2.1 Langages de spécification de propriétés

Quel que soit le type de circuit, un document de départ fiable supportant le projet de conception est crucial. Très vite, il s'est avéré que les langages naturels n'étaient pas assez précis et trop ambigus, laissant les ingénieurs concevoir des systèmes souvent non conformes aux attentes de départ.

Parallèlement, le besoin de pouvoir coupler directement des éléments formels de vérification et de conception est apparu très tôt, bien avant l'explosion de l'ABV. Depuis 1987, le langage VHDL permet d'embarquer directement à l'intérieur du code source des assertions simples (réduites à des propriétés booléennes) supportant la vérification du circuit tout au long de la conception. Dans le cas du circuit $\mathsf{Arbiter}_P$, l'assertion suivante peut être incluse dans la description VHDL :

ASSERT (*reset_n*='0' **IMPLIES** *requested*='0' **AND** *used*='0')
REPORT "Initialization Violation" **SEVERITY ERROR**;

Cette assertion vérifie que lors de l'initialisation du composant $\mathsf{Arbiter}_P$, tous les signaux internes sont réinitialisés à '0'.

Tous les simulateurs VHDL supportent actuellement ces assertions. Elles sont vérifiées continuellement par l'outil qui renvoie des messages d'erreur en cas de violation d'une ou plusieurs assertions.

Les assertions VHDL se réduisent à des propriétés booléennes, n'exprimant donc que des combinaisons de signaux invalides, et sont ainsi parfaitement appropriées à des circuits combinatoires. L'omniprésence des circuits séquentiels a montré la limite de ce type d'assertions. Dans ce domaine, les relations de causalité à travers le temps sont primordiales pour le bon fonctionnement du circuit. Par exemple, une vérification efficace doit pouvoir s'assurer que la propriété $A1_{arbitre}$ est respectée dans le cas de l'arbitre : *si une requête est reçue par l'arbitre, alors elle sera desservie dans le futur*. Ce type de propriété, où le temps entre en jeu, est nommé propriété logico-temporelle. Cette élévation du niveau booléen au niveau temporel est apparue notamment grâce aux langages de propriétés temporelles LTL, CTL etc. D'autres langages ont fait leur apparition par la suite. Ils reposent sur LTL et CTL, mais offrent un sucre syntaxique rendant les propriétés plus faciles à lire et à écrire.

Nous présentons ici deux langages polyvalents (standardisés IEEE) qui répondent à ces deux problèmes en fournissant à la fois un moyen d'écrire des spécifications précises et une méthode efficace pour coupler la vérification à l'implémentation de descriptions HDL.

2.1.1 Le langage PSL

2.1.1.1 Généralités

Historique : PSL est né sous le nom de Sugar dans les laboratoires d'IBM au début des années 2000[1]. Il a été standardisé par Accelera en 2003, puis par IEEE en 2004 [FWMG05]. La dernière révision de la norme 2005 date de 2007. Elle permet d'écrire des propriétés logico-temporelles complexes et adresse trois domaines liés à la vérification de circuits [EF06] :

- spécification : fournit un langage structuré, doté d'une sémantique formellement définie, supprimant ainsi les ambiguïtés liées aux langages naturels.

[1]Documentation disponible sur *http ://www.pslsugar.org/technical_papers.html*

- vérification formelle : model-checking.
- vérification classique : analyse des propriétés en parallèle de la simulation fonctionnelle HDL.

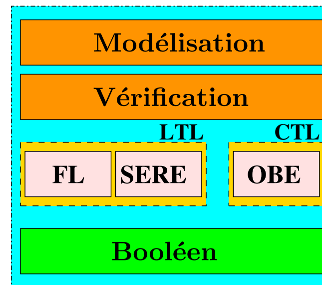

Figure 2.1 – Structure du langage PSL

Structure générale : Le langage PSL supporte cinq syntaxes, dont VHDL et Verilog, et est structuré en 4 couches (*cf.* figure 2.1) :

- **Booléenne** : regroupe les expressions booléennes classiques. Leurs valeurs se réduit à vrai ou faux. Sauf mention contraire, nous considérerons que la valeur logique '0' représente faux et '1', vrai. Cette couche est construite sur les opérateurs booléens classiques : {not, and, or, →}.
- **Temporelle** : contient des relations entre des expressions booléennes au cours du temps. Cette couche est formée de trois ensembles : **FL** (Foundation Language), SERE (Sequential Extended Regular Expressions) et **OBE** (Optional Branching Extension). Les deux premiers ensembles utilisent la logique LTL(Linear Temporal Logic), alors que OBE utilise de la logique CTL et cible donc la vérification statique. La couche temporelle représente le cœur de PSL. Sa présentation en détails est abordée dans la suite de cette section.
- **Vérification** : fournit des directives précisant comment utiliser les propriétés. Le mot clé assert indique que la propriété doit être vérifiée, alors que assume définit le comportement que les entrées doivent respecter pour effectuer la vérification. Ceci peut être utilisé pour contraindre les entrées du circuit sous test, et plonger ce dernier dans un environnement correct lors de la phase de vérification. Enfin, le mot clé cover est utilisé pour des calculs de couverture. Il existe d'autres mots clés disponibles tels que restrict_guarantee etc. [FWMG05].
- **Modélisation** : permet de définir le modèle d'environnement dans lequel est effectuée la vérification. Il est possible de spécifier les contraintes sur les entrées du circuit à tester, ou encore d'affecter des valeurs à des variables auxiliaires. L'environnement ainsi que les propriétés sont généralement groupés dans une structure appelée vunit, comme le montre la figure 2.2. L'assertion $A1_{arbitre}$(i) est la version formelle de l'assertion A1 exprimée en langage naturel dans la section 1.4. Elle permet de vérifier que toute requête émise par la i-ème unité (signal $ask(i)$ actif) se conclura toujours dans le futur par une affectation de la ressource à cette même unité (signal $grant(i)$='1'). L'hypothèse $H1_{arbitre}$(i) est la formalisation en PSL de l'hypothèse $H1_{arbitre}$ décrite dans la section 1.4. Elle permet de contraindre les

entrées du circuit sous test de telle manière qu'une unité ne demande jamais l'accès à la ressource ($ask(i)$ actif) si elle est déjà en train de l'utiliser ($use(i)$).

```
vunit {Arbiter_spec}(Arbiter){
    default clock is rising_edge(Clk) ;
    for i in 0 to P loop
        property A1_arbitre(i) is assert always(ask(i) → eventually! grant(i)) ;
        property H1_arbitre(i) is assume always (not(ask(i) and use(i))) ;
    end loop ;
}
```

FIGURE 2.2 – Exemple d'unité de vérification vunit pour le composant Arbiter_P

2.1.1.2 Propriétés temporelles

La couche temporelle est formée de trois ensembles de propriétés : FL, SERE et OBE. Ces trois ensemble permettent d'exprimer différentes propriétés logico-temporelles.

Le sous-ensemble FL : L'ensemble noté FL contient les opérateurs temporels suivants : {always, eventually!, before, before_, until, until_, next, next[k], next_a[k :l], next_e[k :l], next_event, next_event[k], next_event_a[k :l], next_event_e[k :l]}.

Certains opérateurs possèdent une condition de terminaison future. Si la date de terminaison est fixe et connue, on dit que l'opérateur est borné. L'opérateur next![k] l'est, alors que eventually! ne l'est pas puisqu'il est impossible de connaître à priori la date de terminaison.

Les propriétés temporelles s'étalent sur plusieurs cycles, ce qui entraîne le problème suivant en vérification dynamique : comment interpréter les résultats de vérification si le nombre de cycles utilisé n'est pas suffisant ? Dans le cas de la propriété $\text{A1}_{arbitre}$(i), une vérification trop courte où le signal *ask(i)* est actif à un cycle, mais où *grant(i)* n'est jamais activé, ne permet pas de vérifier complètement la propriété.

L'état de la propriété doit alors être défini, même si la simulation a terminé avant que la condition de terminaison de la propriété ne soit atteinte. Pour cela, le langage PSL fournit deux déclinaisons pour les opérateurs temporels : **fort** et **faible**. La caractère fort est spécifié en ajoutant le caractère '**!**' a la fin de l'opérateur. Nous avons donc until et until !.

La relation entre la force d'un opérateur et la force de satisfaction d'une propriété est définie dans la norme comme suit. Prenons une propriété P où les opérateurs de négation n'apparaissent que sur les expressions booléennes. Soit la propriété P_s où tous les opérateurs de P sont dans leur version forte. Alors la propriété P "Holds Strongly" sur une trace finie si et seulement la propriété P_s "Holds" sur cette trace.

Pour une propriété forte, si la condition de terminaison n'est pas rencontrée, alors la propriété est dans l'état Failed. Le chronogramme de la figure 2.3 illustre la différence de vérification entre une version faible et forte de la même propriété : F2_{cdt}.

Cette propriété énonce qu'à chaque fois que le circuit CDT émet un signal *Send* et qu'il n'est pas en état de réinitialisation, alors il est occupé durant les 4 cycles suivants (signal *Busy* actif). La simulation est stoppée au cycle ♯6. Aux cycles ♯1 et ♯2, la partie gauche de l'implication a la valeur '0' : la propriété est donc dans l'état "Holds" **H**. La propriété

faible termine avec un statut Pending **P**, alors que la propriété forte termine dans l'état Failure **F**.

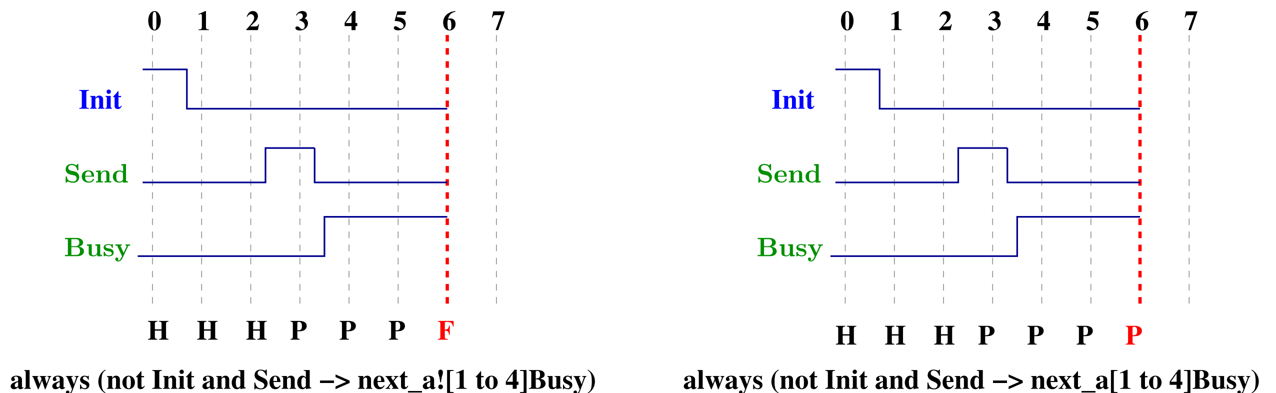

FIGURE 2.3 – Illustration des caractères fort et faible en PSL

Toute propriété PSL possède un des quatre états suivants à tout instant :

- **Holds Strongly** : la propriété est vérifiée fortement et toute extension de la trace conserve cette caractéristique.
- **Holds** : la propriété est vérifiée faiblement. Il peut exister un prolongement de la trace violant la propriété, notamment à cause d'une réactivation de la vérification de celle-ci. C'est le cas au cycle ♯0 de la trace figure 2.3. La partie gauche de l'implication étant fausse, la propriété est vraie à ce cycle. Mais cette propriété peut-être violée dans le futur.
- **Pending** : la propriété est en cours de vérification. Ceci est illustré sur la figure 2.3. La propriété est dans l'état Pending entre les cycles ♯3 et ♯6. Au cycle ♯3, la partie gauche de l'implication est vérifiée, et la vérification de la partie droite débute, ce qui met la propriété dans l'état Pending. Au cycle ♯6, la vérification est toujours en cours puisqu'il reste encore 1 occurrence du signal *Busy* à vérifier.
- **Failed** : la propriété a été violée et toute extension de la trace préservera cette violation, ou la propriété est forte et la vérification n'est pas terminée. La figure 2.3 illustre ceci. Au cycle ♯6, la simulation est stoppée et la propriété est toujours en cours de vérification. La propriété forte est alors dans l'état Failed.

Le sous-ensemble SERE : Cet ensemble est formé d'expressions régulières étendues permettant d'exprimer des propriétés basées sur des séquences de signaux. Il contient les expressions régulières classiques : séquentialité { **;** , **:**}, répétitions consécutives {[*], [*i]} , auxquels s'ajoutent des opérateurs complexes : répétitions non consécutives {[->i], [=i]}. Ces derniers peuvent être réécrits en termes d'opérateurs classiques.

La propriété SERE1 suivante est un exemple de propriété SERE. La sous-séquence {a ;b} signifie : a suivi de b. La partie gauche de l'implication est vraie si la séquence {a ;b} est observée deux fois de suite. Si ce n'est pas le cas, alors SERE1 est vraie avec vacuité. Sinon, la propriété est vraie si à partir du cycle où se termine la partie gauche de l'implication, la sous-séquence {c[→3] :d} est vérifiée. La séquence {c[→3]} est vérifiée s'il y a trois répétitions non forcément consécutives du signal *c*. Si lors de la dernière répétition de *c*, au même cycle, le signal *d* est actif, alors la propriété SERE1 est vérifiée.

$$\text{property SERE1 is assert } \{\{a\,;b\}[*2]\ |\!\rightarrow \{c[\rightarrow 3]\ :d\}\}$$

La trace de la figure 2.4 respecte la propriété SERE1. La vérification commence au cycle ♯0. La partie gauche est vérifiée au cycle ♯3. L'évaluation de la partie droite commence à ce même cycle ♯3. La première occurrence du signal c arrive au cycle ♯5, la seconde au cycle ♯8 et la dernière au cycle ♯9. À ce cycle, le signal d vaut '1', ce qui termine la validation de la propriété SERE1.

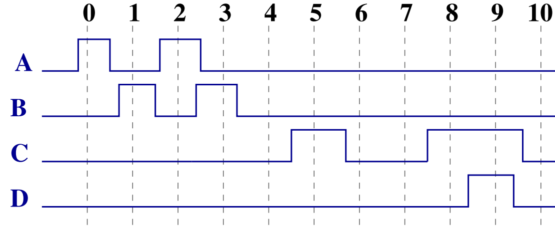

FIGURE 2.4 – Trace respectant la propriété SERE1

Le sous-ensemble OBE : Utilisé pour la vérification statique uniquement, cet ensemble supporte les opérateurs A[2] et E[3] de la logique CTL. La propriété $\text{OBE12}_{arbitre}$(i) suivante appartient à l'ensemble OBE :

$$\text{property OBE12}_{arbitre}(\text{i}) \text{ is assert } \mathbf{A}\ (\text{never}(use(1) \text{ and } use(2)))\,;$$

L'outil de vérification statique retournera "vrai" si quelle que soit l'exécution du système, il n'existe aucun état où deux unités ont accès en même temps à la ressource.

Le sous ensemble simple de PSL : De la même manière que VHDL possède un sous-ensemble pour la synthèse, PSL possède un sous-ensemble pour la vérification dynamique : le **sous-ensemble simple**, noté PSL_{ss}. L'idée consiste à imposer des restrictions sur le langage pour permettre aux outils la vérification à la volée de propriétés temporelles. Pour réaliser ceci, l'état d'une propriété à un instant donné ne doit dépendre que des évènements passés. Cet écoulement du temps, de gauche vers la droite à travers la propriété, est obtenu grâce aux restrictions fournies dans le tableau 2.1.

La plupart des propriétés utilisées en vérification dynamique n'appartenant pas à cet ensemble peuvent être réécrites pour entrer dans le sous-ensemble simple. La propriété $\text{A2a}_{arbitre}$(i) ne fait pas partie du sous-ensemble simple, mais peut se réécrire sous la forme $\text{A2b}_{arbitre}$(i) qui entre dans PSL_{ss}.

- property $\text{A2a}_{arbitre}(\text{i})$ is assert always$((ask_i \text{ and next!}[2]use_i) \rightarrow \text{next!}[1]grant_i)$;
- property $\text{A2b}_{arbitre}(\text{i})$ is assert always$(\{ask_i\,;\text{not } grant_i\}|\!\rightarrow\!\{\text{true}\,;\text{not } use_i\})$;

La propriété $\text{A2a}_{arbitre}$(i) se lit : si une unité demande un accès à la ressource au cycle courant et qu'elle l'utilise deux cycles plus tard, alors forcément elle a obtenu un jeton au cycle suivant. La propriété équivalente $\text{A2b}_{arbitre}$(i) exprime quant à elle : si une unité demande un accès à la ressource au cycle courant et qu'elle n'obtient pas de jeton au cycle

[2]Ap : tous les chemins d'exécution doivent vérifier la propriété p.
[3]Ep : il existe au moins un chemin d'exécution respectant p.

Opérateur PSL	Restriction
Not	opérande booléen
never	opérande booléen ou séquence
eventually!	opérande booléen ou séquence
Or	au moins un opérande booléen
$\rightarrow$	opérande gauche booléen
$\leftrightarrow$	deux opérandes booléens
until until !	opérande droit booléen
until_ until_!	deux opérandes booléens
before*	deux opérandes booléens
next_e	opérande booléen
next_event_e	opérande droit booléen

Table 2.1 – Restrictions du sous-ensemble simple PSL_{ss}

suivant, alors elle n'utilise pas la ressource deux cycles plus tard. Cette dernière propriété est plus claire puisque l'écoulement du temps est linéaire dans la propriété.

Les propriétés en dehors de PSL_{ss} peuvent être compliquées à comprendre puisque les relations de causalité sont dispersées au sein même de la propriété et cassent l'écoulement du temps de gauche à droite à travers celle-ci. De plus, implémenter des outils de vérification dynamique n'est pas possible pour des propriétés non bornées en dehors de PSL_{ss}. Dans le cadre de la vérification dynamique, toutes les propriétés que nous avons eu à utiliser ont toujours pu s'écrire en respectant les contraintes de PSL_{ss}.

2.1.2 Le langage SVA

Le langage SVA (SystemVerilog Assertions) est basé sur la syntaxe C. C'est un langage de spécification de propriétés logico-temporelles standardisé en 2005 [SMB+05] au cœur de la norme SystemVerilog. SVA et SystemVerilog sont donc complètement couplés et la couche "assertion" peut être combinée avec le code SystemVerilog. Toute une méthodologie basée sur SVA a été définie pour l'ABV [BCHN06].

Le langage contient deux types d'assertions : immédiates et concurrentes. Les assertions immédiates sont réduites à des expressions booléennes (pas d'horloge ni de reset). Ces assertions peuvent être placées n'importe où dans le code et seront exécutées comme des instructions classiques. Les assertions concurrentes sont exécutées en continu, parallèlement à la simulation, et peuvent être booléennes ou temporelles.

Le langage SVA est structuré en 4 couches :

- **booléen** : contient les opérateurs classiques booléens : {&&, ||, !}. Même couche que pour PSL.
- **séquence** : ensemble d'expressions booléennes au cours du temps : séquences, répétitions etc. Couche identique aux SERE de PSL.
- **propriété** : implications de séquences. Une propriété est une séquence vérifiée à chaque cycle.
- **assertion** : contient les directives assert, assume et cover indiquant à l'outil comment traiter la propriété courante.

Bien que PSL et SVA, soient à première vue, assez proches, plusieurs différences les distinguent [HW05]. SVA est basé sur les séquences, ce qui n'est pas le cas de PSL où les opérateurs FL constituent vraiment le cœur du langage. Le langage SVA ne possède aucun équivalent des OBE de PSL, ce qui oriente son utilisation bien plus pour la vérification dynamique que statique. De plus, SVA ne possède pas directement de différenciation entre les caractères fort et faible.

2.1.3 Autres langages

En 2002, Synopsys met sur le marché le langage de vérification matériel OVA 2.0 (Open Verification Library), basé sur ForSpec. Sa syntaxe est basée sur SystemVerilog. Ce langage est polyvalent (vérification, création de test bench etc.) et semble offrir des caractéristiques proches de celles fournies par SVA.

La puissance des langages polyvalents tels que PSL, SVA ou encore OVA est quelquefois un frein dans le monde de l'industrie où l'expérience nécessaire pour maîtriser un nouveau langage, une nouvelle approche, doit être minimale et facile à acquérir. C'est dans ce but de faciliter l'accès à la vérification à l'aide d'assertions, que de nombreux groupes de travail, issus aussi bien du monde industriel qu'académique, ont développé des alternatives aux langages de vérification purs.

Hewlett Packard a développé dès 1990 une bibliothèque de composants paramétriques pour la vérification en ligne : l'Open Verification Library (OVL) [FLT06]. Standardisée par Accellera en 2001, OVL est une bibliothèque d'assertions pré-conçues (paramétriques) permettant aux utilisateurs d'écrire facilement des propriétés temporelles (sur de la logique à 4 valeurs), sans avoir à apprendre un langage spécifique tel que SVA ou PSL. Une documentation et des outils sont fournis afin de comprendre le comportement des assertions utilisées et simplifie grandement la définition d'assertions correctes. Cette simplification est faite au détriment du pouvoir d'expression, puisque les composants OVL ne couvrent pas tout l'ensemble de possibilités offertes par PSL ou SVA.

Enfin, grâce à l'augmentation grandissante de l'importance de la vérification supportée par assertions, de nombreuses compagnies fournissent maintenant des bibliothèques propriétaires de vérification de composants standards : contrôleurs (USB, RS232), protocoles (AMBA AHB, Ethernet) etc. Chaque élément de ces bibliothèques contient toute une batterie de propriétés documentées afin de vérifier le composant concerné. Ceci évite ainsi de passer un temps conséquent à définir les propriétés à vérifier, et assure de plus que celles-ci sont correctes. Les principales bibliothèques actuelles sont : QVL, Checker-Ware [Gra08] et IAL [Siw06].

2.1.4 Outils existants supportant l'ABV

De nombreux outils supportent l'ABV et fournissent un riche panel d'aide au debug basé sur l'utilisation d'assertions. Cette section présente rapidement ces outils et tente de dresser un début de comparaison basé sur les possibilités offertes pour supporter la vérification. Seuls les outils FoCs, MBAC et MyGen ont été utilisés afin de les comparer à notre approche nommée Horus (*cf.* Chapitre 7).

Le tableau 2.2 regroupe toutes les caractéristiques des outils les plus communément utilisés. Il est divisé en trois parties : les solutions logicielles complètes intégrant à la fois des outils de vérification classique et formelle ; les outils de vérification statique à base de

model checking ; et enfin des outils spécifiques pour la création de moniteurs et générateurs synthétisables.

Le symbole "$\surd$" signifie que l'outil possède la caractéristique correspondante, alors que "$\times$" indique le contraire.

Incisive, Questa et Verdi sont les logiciels de vérification semi-formelle les plus complets. Incisive et Questa possèdent leurs propres bibliothèques d'assertions. Ils possèdent tous des aides au debug plus ou moins évoluées, basées sur l'utilisation d'assertions. On distinguera notamment le proof-radius de la suite Questa. Il est utilisé pour donner automatiquement la couverture de l'assertion par rapport au circuit. La couverture donne une estimation sur le nombre de chemins pour lesquels l'assertion a été testée. Un proof-radius de 100% équivaut à une couverture complète de la propriété et correspond ainsi à une vérification statique par model-checking puisque cela signifie que l'assertion a été testée et validée sur tous les chemins du circuit.

Incisive fournit une extraction automatique d'assertions simples (principalement basée sur la fonction de transition...). Ceci peut s'avérer pratique pour une première étape de vérification.

L'outil Verdi propose le post-process assertion mode. Il permet d'ajouter ou de modifier des assertions sans re-simuler. Conquest est aussi une solution complète, mais semble fournir moins de support pour le debug que ses concurrents.

Les outils de model-checking sont bien plus difficiles à différencier sans test de performances. Ils possèdent néanmoins certaines caractéristiques qui peuvent être intéressantes : langages pris en entrée, type de moteur de vérification utilisés etc. L'outil RuleBase est celui qui a été utilisé dans les expérimentations rapportées ici. Il est très efficace et simple d'utilisation. Il possède lui aussi plusieurs moteurs de vérification. La description HDL peut être passée directement en entrée ainsi que le fichier contenant les propriétés exprimées en PSL.

Enfin, les outils spécialisés dans la synthèse de propriétés sont particulièrement intéressants, puisqu'ils rejoignent les travaux effectués ici. Il est donc primordial de pouvoir effectuer une comparaison, non seulement sur les outils fournis, mais aussi sur les performances de ces outils : temps de synthèses et complexité des descriptions HDL produites.

L'outil FoCs d'IBM est le premier à avoir permis la synthèse d'assertions. Malheureusement, les premières versions de ce logiciel produisaient des descriptions HDL non synthétisables. Aujourd'hui, ce problème est résolu. Comme le montrera en détails le chapitre 7, les techniques de synthèse sont à base d'automates et produisent des circuits trop complexes pour être utilisés efficacement sur FPGA. Ceci limite l'utilisation de FoCs à de la vérification par simulation.

Le logiciel de synthèse de propriétés qui se rapproche le plus, au niveau performances, de celui développé dans notre équipe est MBAC. Développé à McGill par Marc Boulé, il prend en entrée aussi bien PSL que SVA et fournit en sortie des composants de vérification très efficaces : faibles complexités et hautes fréquences de fonctionnement. Cet outil sera décrit en détails dans la section 3.4.

L'outil Cando [EBS+07] permet la création de composants modélisant le comportement défini par un ensemble de propriétés temporelles. Cet outil est très proche des générateurs produits par Horus.

La plateforme Horus est l'outil développé au sein de l'équipe VDS où mes travaux ont été effectués. Une partie de ce développement a été effectué dans le cadre de cette thèse. Horus permet non seulement de fabriquer des moniteurs, mais aussi des générateurs. Elle

sera présentée en détails dans la section 7.1.

Outils	Supports					Langages de spécifications			
	Model Checking	Génération de stimulis	Simulation	Emulation	Débogue	PSL	SVA	OVL	Autre
Solutions complètes supportant l'Assertion-Based Verification									
Questa	0-In	OVM, SCV constraints	√	×	tracking fault, ATV,MVC libraries,coverage	√	√	√	QVL
Incisive	Incisive Verifier	SystemC, HDL	√	×	Assertion Extraction	√	√	√	IAL
Verdi	×	SVTB	√	×	Fault isolation, post process assertion, assertion analysis	×	√	×	×
Conquest	Multiple Engines	×	√	×	assertion counter example	√	√	√	×
Active-HDL	×	random, SCV, waveform	√	×	assertion diagram evaluation	√	√	×	×
Magellan	Multiple Engines	×	VCS	×	mix simu/model checking	×	√	√	OVA
Outils de Model-Checking									
RuleBase	Multiple Engines	×	×	×	counter example	√	√	√	×
0-In	Multiple Engines	×	√	×	proof radius, coverage	√	√	√	QVL, Cware, 0-In
Jasper	Multiple Engines	×	×	×	×	√	√	×	×
Improve-HDL	√	×	×	×	contre exemple	√	√	×	×
Outils de construction de moniteurs synthétisables									
FOCS	×	×	×	×	×	√	×	×	×
MBAC	×	×	√	√	×	√	√	×	×
Cando	×	√	√	√	×	×	×	×	ITL
HORUS	×	√	√	√	×	√	×	×	×

Table 2.2 – Principaux outils de vérification supportant l'ABV

2.2 Vérification à l'aide de moniteurs

La synthèse d'assertions en moniteurs est apparue en 2005 et s'est répandue notamment grâce à la commercialisation de l'outil FoCs d'IBM. Celui-ci utilise une approche à base d'automates permettant de transformer des propriétés PSL en moniteurs Verilog, VHDL, ou C++. Alors que dans les premières versions, les moniteurs HDL n'étaient pas synthétisables, il est maintenant possible de les utiliser en émulation ainsi qu'en simulation. De nombreuses autres approches ont alors vu le jour, en tentant d'obtenir des composants le plus simple possible, pour minimiser au maximum l'impact des moniteurs à la fois sur le temps de simulation, et sur la taille du circuit correspondant dans le cas d'une utilisation matérielle.

Dans [PLBN05], Pellauer *et al.* synthétisent des assertions SVA en modules BlueSpec SystemVerilog synthétisables. BlueSpec SystemVerilog (BSV) est un système de synthèse de propriétés. Seul un sous ensemble de SystemVerilog peut être traité.

La méthode n'est pas efficace puisqu'il y a duplication des FSMs pour traiter la réentrance. Prenons par exemple la propriété A⇒B avec A et B deux propriétés temporelles durant respectivement N et P cycles. Le moniteur contient alors N fois l'automate de la propriété A et P fois celui de B, ce qui fait exploser la taille de tels moniteurs. Le traitement d'opérateurs temporels non bornés est donc interdit.

Dans le même temps d'autres approches se développent. Gheorghita et Grigore présentent dans [GG05] une méthode de production de moniteurs non synthétisables à base d'automates non déterministes contenant des compteurs. Ceci permet de réduire drastiquement le nombre d'états de l'automate. Dans de nombreux cas, l'automate doit être déterminisé afin d'obtenir le moniteur final. Ceci casse l'efficacité de l'approche. Les résultats sont moins bons que ceux obtenus par FoCs mais la sortie est bien plus lisible. Cette approche est intéressante, car elle utilise le même style de concepts (en bien plus simples) que ceux supportant la création de moniteurs pour MBAC [BZ05].

Gascard utilise la méthode des résidus [Gas05], pour la construction d'automates d'états finis (AEFD) permettant la création de moniteurs pour des propriétés SERE de PSL. Dans la plupart des cas, les moniteurs produits ne prennent pas en compte la réentrance. Aucun résultat pratique n'est donné : ni le temps de construction, ni les caractéristiques en synthèse.

Dans [PKBMF05], Pidan *et al.* se focalisent sur des propriétés de type sûreté (safety properties). Celles-ci sont synthétisées en moniteurs HDL ou C++. L'ensemble de l'approche est formalisé et la construction est prouvée correcte. La propriété est transformée en automate non déterministe qui est lui même transformée en DTS (Discrete Transition System). Un DTS est une représentation du programme sous forme de graphe où les sommets sont des instructions et les arrêtes les chemins du programme. De nombreuses optimisations sont effectuées tout au long de la construction. La création de moniteurs en C++ est spécifique et la méthode est détaillée.

Le lien entre propriétés temporelles et automates de Büchi est étroit et fondamental dans l'utilisation de ces propriétés aussi bien en model-checking qu'en vérification en ligne à l'aide de moniteurs. Toute propriété LTL peut être modélisée par un automate de Büchi dont le langage accepté représente toutes les traces satisfaisant la propriété. De nombreuses techniques ont été développées afin de transformer de manière efficace une propriété en un automate de Büchi le plus simple possible. Les approches classiques obtiennent un automate final dont la taille explose lorsque la propriété devient complexe.

Cimatti *et al.* présentent une méthode efficace pour effectuer la transformation d'une

propriété PSL en un automate de Büchi non déterministe dans [CRST06]. Pour cela, deux concepts sont appliqués : représentation symbolique de l'automate ; approche modulaire via la séparation des parties LTL et SERE de chaque propriété. L'objectif est principalement la construction d'automates pour le model-checking, mais il est aussi possible de construire des moniteurs. Non seulement le temps de construction des automates est meilleur que les méthodes existantes, mais le temps de vérification en model-checking est lui aussi diminué puisque les automates obtenus sont plus simples. Aucun résultat pertinent n'est donné pour les moniteurs. Enfin, la méthode de transformation est prouvée correcte.

Au fil du temps, les approches se sont raffinées et la qualité des moniteurs s'est améliorée. Une approche pour construire des moniteurs dont la complexité est linéaire vis-à-vis de la propriété a été définie par Jin et Shen [JS07]. Ceux-ci sont uniquement matériels. La méthode repose sur l'utilisation de LAFA : Local-variables Alternating Finite Automaton. La méthode est formalisée et les moniteurs sont corrects par construction. Il serait possible d'obtenir des moniteurs matériels, mais une étape de déterminisation doit être appliquée et la complexité du moniteur final n'est plus linéaire en la propriété. Aucun résultat expérimental n'est fourni : pas de temps de construction, de taille etc. D'après le type d'approche, il semble que les temps de construction vont être conséquents pour des propriétés complexes.

Une méthode à base de HLDD (High Level Decision Diagram) est décrite dans [JRCU07]. Les assertions PSL sont traduites en moniteurs logiciels grâce à FoCs. Ceux-ci sont alors transformés en HLDD. Malgré l'originalité dans l'utilisation d'un HLDD, un seul résultat expérimental est fourni. L'utilisation de cette approche réduit l'impact des moniteurs sur les temps de simulation d'un facteur 10. Mais FoCs produit des moniteurs peu efficaces, et comme la production du moniteur final se base sur ceux-ci, il semble bien difficile d'obtenir des résultats comparables aux méthodes les plus efficaces.

Jusqu'à maintenant, les approches produisant des moniteurs ne prenaient pas en compte les couches modélisations des langages PSL et SVA. Dans [SJE06], Safari *et al.* lèvent cette restriction pour des propriétés SVA seulement. Il est alors possible d'utiliser des modules de vérification embarquant des variables locales et de synthétiser ceux-ci en moniteurs. Si une propriété est relancée plusieurs fois, alors plusieurs threads sont exécutés ayant chacun une instance des variables locales potentielles. Ceci n'est applicable que pour des moniteurs logiciels. L'efficacité de la méthode peut chuter dans le cas où le nombre de threads explose durant la vérification, car le nombre de variables ou de propriétés est trop important.

À notre connaissance, l'approche la plus efficace à l'heure actuelle pour la synthèse de moniteurs est fournie par Boulé et Zilic [BZ05]. L'approche se base sur un traitement spécifique des parties gauche et droite d'une implication, et un ensemble de règles de ré-écriture. Un sous ensemble des opérateurs primitifs de PSL possède des automates pré-définis. Les autres opérateurs sont traités à l'aide de règles de ré-écriture. L'obtention d'automates pour des propriétés complexes s'effectue en combinant les automates primitifs. Des optimisations sont effectuées, en particulier en partie droite d'une implication où seules les violations doivent être détectées. Une instrumentation est ajoutée dans [BCZ06], permettant une aide au debug. Il est possible de compter les erreurs, les activations de parties gauches d'implications, ou même d'identifier, pour chaque erreur, quelle instance de la propriété a été violée (en cas de ré-entrance).

Alors que la plupart des approches se basent sur des langages de spécification déjà

existants et largement utilisés, Chen *et al.* font le choix inverse dans [CHBW04]. Ils présentent une méthode de synthèse de moniteurs logiciels à partir de propriétés temporelles exprimées en LOC (Logic of Constraints). LOC permet de raisonner sur les traces d'exécutions de systèmes. Il contient tous les opérateurs classiques de la logique propositionnelle, auxquels s'ajoutent des opérateurs arithmétiques permettant d'exprimer des aspects liés aux performances. La propriété ci-dessous vérifie qu'une donnée *Data* est reçue toute les 10 unités de temps :

$$\textsf{property}\ \text{freq_data} : t(Data(i+1)) - t(Data(i)) = 10$$

LOC est différent des langages de type LTL car certaines formules peuvent être exprimées en LOC, mais pas en LTL et vice-versa. Par exemple LOC n'exprime que des propriétés de sûreté. La ré-entrance peut faire exploser l'espace mémoire requis pour la vérification en simulation. Concernant la vérification statique, la méthode est applicable pour un sous ensemble de LOC.

L'écriture d'assertions est toujours complexe et requiert une certaine expérience. C'est pourquoi Nepal *et al.* s'intéressent à l'extraction automatique d'assertions à partir de l'analyse d'une description du circuit au niveau portes [NADB08]. Tout d'abord une simulation est effectuée afin d'analyser toutes les propriétés de type “A→B” où A et B sont des signaux, au sein du circuit. Ensuite, l'ensemble d'implication est validé et optimisé à l'aide d'un solveur SAT. Ensuite les implications sont combinées et il est possible d'obtenir des propriétés temporelles du type “A→next[k]B”.

Wagner et Bertacco utilisent les assertions non pas comme une fin, mais comme un moyen pour concevoir des processeurs qui s'auto-vérifient et s'auto-corrigent durant leurs exécutions [WB07, WBA08]. Le principe consiste à développer deux modèles d'un même composant : un complexe et efficace (pipelines, prédiction de branchements etc.) ; et un autre très simple mais formellement vérifié. L'utilisateur sélectionne un ensemble de signaux qu'il juge critiques pour le bon fonctionnement du système. Une étape de vérification est alors effectuée. Durant celle-ci, l'ensemble des états du système est partitionné en deux : états sûrs et états non-sûrs. Pour ceci, une métrique de confiance est utilisée. Ensuite, un moniteur (nommé “guardian” dans le papier) est alors construit pour détecter tout configuration amenant le système dans un état non sûr. Si celui-ci détecte un tel cas, alors le composant principal est stoppé et le flot d'exécution est passé au composant simple dont on sait que l'exécution est correcte. Tout ceci entraîne un faible surcoût : +3,5% surface et +5% performance. Deux faiblesses apparaissent tout de même dans cette méthode : c'est le concepteur qui choisit l'ensemble des signaux pour la création du guardian ; suivant la finesse de la vérification, des états peuvent être marqués sûrs alors qu'ils contiennent des bugs.

Makris *et al.* proposent une autre utilisation des moniteurs [MBO04] dans le cadre du test en ligne de circuits. L'idée consiste à coupler au circuit (modélisé par une fonction f(x)) une fonction de transparence g(x) qui capture le fonctionnement correct du circuit à partir de descriptions de haut niveau. Un moniteur est utilisé de manière à vérifier qu'une certaine relation existe entre f(x) et g(x). Si ce n'est pas le cas, le circuit contient une ou plusieurs erreurs. Les moniteurs sont très simples et construits de manière structurelle directement au niveau RTL (en combinant registres, portes logiques etc.). Il serait intéressant de pouvoir complexifier les relations entre f(x) et g(x) et d'utiliser des assertions temporelles pour les exprimer.

La synthèse de propriétés en moniteurs est actuellement efficace grâce notamment aux nombreux travaux qui l'ont fait évoluer ces dernières années. À notre connaissance, les

méthodes de synthèse de moniteurs matériels les plus efficaces sont : MBAC (développé par Marc Boulé [BZ05]), et Horus (inventé par Miao Liu *et. Al* [BLOF06]). L'utilisation de ces assertions a commencé plus récemment à montrer sa puissance grâce à différentes mises en œuvre [NADB08, WB07] permettant d'exploiter de manière originale les moniteurs.

2.3 Génération de vecteurs de tests

La génération de vecteur de tests est la clé permettant d'effectuer un test efficace, et ainsi de garantir la fiabilité d'un circuit. Sans stimulus de qualité, il est impossible, ni d'obtenir de réponses exploitables, ni de couvrir les fonctionnalités importantes du circuit. De nombreuses méthodes ont été définies depuis une quinzaine d'années [JPJ97, Edv99, JJJ+01].

2.3.1 Built In Self Test

Une des méthodes les plus populaires pour le test consiste à embarquer directement dans le circuit les vecteurs de test et un mécanisme permettant de basculer le circuit en mode test : le BIST (Built In-Self Test). Les vecteurs étant embarqués, la surface occupée par le circuit est plus grosse. De plus, les mécanismes pour le test abaissent la fréquence du circuit. Trois techniques principales sont utilisées pour mettre en œuvre des techniques BIST :

- test exhaustif : tous les vecteurs sont enregistrés tels quels dans une ROM. La surface est maximale, la fréquence est peu impactée [BBM91].
- test pseudo-aléatoire : un bloc de génération aléatoire est embarqué et produit les vecteurs en ligne. Le nombre de vecteurs est plus important puisque de nombreux cas sont inutilisables ou redondant [LM00].
- test spécifique : un ensemble de vecteurs pré-calculés sont placés sur le circuit afin de ne tester que certains aspects du composant [BK95].

Dans [CCPSR97], le circuit est simulé grâce à un premier ensemble de vecteurs de test aléatoire. Ceux-ci sont alors optimisés par un algorithme génétique qui effectue des mutations afin d'augmenter le taux de couverture des vecteurs. Une fois l'ensemble final obtenu, les vecteurs sont produits à l'aide d'un automate cellulaire spécifique. Ceci optimise la surface et la fréquence du circuit instrumenté.

2.3.2 Approches à base de découpage

Alors que la complexité des circuits a explosé au fil des années, une idée très répandue dans le domaine de la micro-électronique a consisté à appliquer une approche : "diviser pour régner". Dans [TA97, TKA99, MO99], les auteurs découpent les circuits complexes en plusieurs sous modules. Des contraintes sont extraites pour chaque module afin de guider et de simplifier la production pseudo-aléatoire de vecteurs de test. La recomposition de tous les modules produit les vecteurs de test pour le circuit final.

2.3.3 Approches mixtes : techniques formelles/classiques

Plusieurs approches utilisent le model-checking comme outil de calcul de vecteurs de test. L'idée consiste à utiliser le model checking sur un circuit incorrect et à extraire un

ensemble de contre exemples. Ceux-ci sont utilisés pour produire des vecteurs de test ciblant des fautes spécifiques.

Dans [ABM98, BOY01], Black *et al.* fabriquent plusieurs versions erronées de la spécification initiale du circuit en appliquant des opérateurs de mutation. Chaque opérateur définit une catégorie de fautes à vérifier (Stuck-at, oubli de condition etc.). Le model checker va ensuite comparer les deux spécifications et enregistrer les traces correspondant aux exécutions des deux machines à états jusqu'à ce qu'il y ait une différence de comportement. Ces traces sont sauvegardées sous forme de vecteurs de tests à fournir en entrée du circuit afin de détecter la faute associée. Le model checker peut aussi servir à déterminer le taux de couverture du jeu de test créé. Une des limitations réside dans la complexité de la spécification puisque le model checker peut se trouver confronté au problème d'explosion du nombre d'états.

Toujours en utilisant des outils de model-checking, Ugarte et Sanchez produisent des vecteurs de test pour des descriptions comportementales de circuit [US03]. L'approche s'appuie sur une analyse d'intervalles. Le programme est modélisé sous forme de polynôme, des contraintes sur la valeur des variables sont données et une exploration de l'espace d'états du programme est effectuée par un model-checker pour trouver tous les chemins violant ces contraintes. Ceci fournit des vecteurs de test.

L'approche décrite par Seater et Dennis [SD05] utilise aussi des opérateurs de mutation. Elle prend en entrée un programme puis produit plusieurs variantes incorrectes à l'aide d'opérateurs de mutation. Le programme est modélisé par une formule mathématique. Les modèles corrects et contenant des fautes sont alors passés par un solveur SAT qui calcule une solution transformée en vecteur de test.

Wen *et al.* utilisent aussi des modélisation mathématiques pour la génération de vecteurs de test [WWC05]. Un RTPG (Random Test Pattern Generator) est utilisé et les séquences d'entrées et sorties sont enregistrées. À partir de ces données récoltées, une fonction mathématique est créée pour modéliser le module. Celle-ci réalise un mapping des sorties sur les entrées. Elle sert à calculer les vecteurs de tests corrects à fournir en entrée du circuit et à observer la propagation d'erreurs dans le circuit. Les techniques de modélisation pour les parties contrôle et opérative sont :

- Boolean Mapping : chaque fonction de sortie est représentée sous forme de BDD. La conjonction de tous ceux-ci donne les vecteurs corrects qui doivent être appliqués en entrée.
- Arithmetic Mapping : le calcul s'effectue en considérant les sorties comme des fonctions polynômes des entrées.

Les résultats sont variables selon les circuits. La difficulté vient de la complexité introduite lors de la création du modèle mathématique du circuit.

Toujours en utilisant des techniques formelles, Keim *et al.* présentent dans [KDB99] comment utiliser la simulation symbolique couplée à un algorithme génétique pour la production de vecteurs de test. La simulation symbolique fourni ainsi une séquence de test pour détecter une faute, et l'algorithme génétique va optimiser cette séquence pour réduire sa taille tout en maximisant le taux de couverture.

2.3.4 Approches à base de graphes

De nombreuses approches sont basées sur des techniques logicielles, adaptées pour le test de composant électronique. Une méthode très répandue consiste à modéliser le circuit

sous forme d'automate et à parcourir les chemins de celui-ci pour extraire les vecteurs de test. Cette technique fut introduite par McCabe [McC76].

Dans [Pao01], Paoli modélise le circuit à l'aide d'un CFG (Control Flow Graph) et calcule l'ensemble minimal de vecteurs de test couvrant tous les chemins possibles du CFG. Plusieurs expérimentations ont été menées sur des circuits issus de ITC99 Benchmarks. Le même type d'approche est utilisé par Krug *et al.* [KLM06]. Une étape de plus est appliquée. Les vecteurs de test obtenus sont réutilisés pour en produire d'autres au niveau portes à l'aide d'un ATPG (Automatic Test Pattern Generator). Ceci améliore le taux de couverture.

Ferrandi *et al.* présentent dans [FFS98] une approche basée sur l'injection de fautes. La description HDL est transformée en CFG (Control Flow Graph), puis en RBDD[4]. Plusieurs fautes sont injectées à la fois dans le BDD pour obtenir une version incorrecte. Le BDD modifié est comparé à l'original et des vecteurs de test sont extraits. Une synthèse de haut niveau est appliquée sur le VHDL et l'outil Converter transforme les vecteurs obtenus en séquences. basée sur deux modèles de fautes : 'Bit failure' : bit collé à 1 ou 0 ; ou 'Condition failure' : condition collée à 1 ou 0. Les résultats expérimentaux montrent que les taux de couverture sont très élevés, mais ceci implique une forte augmentation de la taille des vecteurs de test.

2.3.5 Approches originales

Une approche à base d'algorithme génétique est présentée par Corno *et al.* [CRS99]. La description HDL de niveau RTL est simulée à l'aide de stimuli aléatoires. Ceux-ci sont optimisés par un algorithme génétique permettant d'arriver à un ensemble final de vecteurs de test fournissant un taux de couverture (en terme d'instructions) maximum. L'outil se nomme RAGE99. Des résultats expérimentaux montrent que malgré un bon taux de couverture, le temps mis par l'algorithme génétique peut rapidement devenir un problème pour des circuits complexes. Un cas d'étude pour cette approche est étudié dans [CSRS04].

Un début de méthode très originale est présenté dans [WTFAK07]. Waeselynck *et al.* utilisent le “recuit simulé” (simulated annealing) pour produire des vecteurs de test optimums. Dans le cadre du recuit simulé, trouver le meilleur vecteur de test parmi une population de vecteurs possibles revient à trouver un pic (altitude maximale) ou un creux (altitude minimale) dans un paysage en trois dimensions. Contrairement aux algorithmes génétiques qui calculent tout un ensemble de solutions via de multiples mutations pour arriver à une solution finale optimale, le recuit simulé calcule une seule solution à la fois. Il utilise une exploration pas à pas du paysage pour déduire le meilleur vecteur. La recherche est influencée par des paramètres fournis en entrée de l'algorithme. Le but de l'article est de déduire ces paramètres via une analyse de la topologie du paysage définissant le problème. La méthode est présentée et des études empiriques sont exposées. Bien qu'à ses débuts, cette approche est intéressante par son originalité, mais aussi car peu de gens ont étudié l'utilisation du recuit simulé dans ce contexte.

[4]Reduced BDD : compaction d'un BDD en éliminant les terminaux dupliqués. Il existe un unique RBDD pour un ensemble de BDDs équivalents.

2.3.6 Approches à base de propriétés temporelles

Quelques méthodes utilisent des propriétés temporelles pour spécifier les contraintes sur les vecteurs de test. La méthode décrite par Shimizu et Dill [SD02, Shi02] utilise des propriétés du type : *expr1* → *next (expr2)* pour contraindre à la volée des vecteurs quelconques. À chaque cycle d'horloge le programme évalue les parties gauche des propriétés et marque “Active” celles qui sont valides. La conjonction de toutes les règles marquées ”Active” est effectuée ; ce qui conduit à une formule booléenne pour chaque interface. Un solveur à base de BDD détermine les combinaisons satisfaisant cette formule. L'une d'entre elles est choisie et constitue alors le vecteur d'entrée du cycle d'horloge courant. En modifiant les probabilités d'activation de chaque propriété, l'ensemble des vecteurs produits est enrichi, ce qui améliore le taux de couverture.

Pal *et al.* présentent eux aussi une approche afin d'optimiser la couverture d'un ensemble de propriétés [PBSD08]. L'ensemble des propriétés LTL est pris en compte. Mais les auteurs se ramènent à des propriétés du même type que précédemment (*cf.* [SD02]) en découpant la propriété à chaque instant de la manière suivante : *contraintes au cycle courant* → *contraintes au cycle suivant*. L'amélioration du taux de couverture s'effectue indépendamment sur chaque propriété ainsi découpée. Un solveur booléen calcule les affectations qui rendent vrai la partie gauche, puis la partie droite. Ainsi toute les propriétés sont vérifiées sans vacuité, ce qui augmente le taux de couverture des vecteurs de test.

Dans [MADS07], Mathaikutty *et al.* présentent un environnent de test pour des descriptions de circuits en SystemC. À partir d'une spécification fournie en anglais, un modèle ESTEREL (Modèle Fonctionnel) et un ensemble de propriétés PSL (Modèle de vérification) sont extraits. Le modèle est instrumenté pour exprimer les contraintes des vecteurs de test. Trois utilisations différentes sont faites sur cette instrumentation pour prendre en considération les taux de couverture suivants : instructions, branches et MCDC (Modified Condition/Decision Coverage).

Rusu *et al.* posent les bases d'une méthode de génération de vecteurs de test fondée sur un modèle d'automate adapté aux circuits électroniques : les IOLTS (Input Output Labelled Transition System) [RMJ04]. Ce modèle est basé sur les automates classiques et ajoute le concept d'entrées/sorties. Le principe consiste à modéliser le circuit d'une part, et la propriété temporelle d'autre part, sous forme d'IOLTS. La méthode produit des vecteurs de test pour violer une propriété. Les automates sont composés et parcourus afin d'extraire les séquences de test pour une propriété donnée. L'outil TGV utilise ces concepts et automatise l'approche décrite ci-dessus [JJ05, Cal05].

2.3.7 Bilan

Les deux sections précédentes dressent le tableau actuel concernant la vérification de circuits à l'aide d'assertions et la génération de vecteurs de test via différentes approches. Les méthodes sont globalement efficaces sur des circuits simples ou sont focalisées sur un type précis de composants. Malgré l'effort important déployé pour supporter la vérification de systèmes complexes, le problème reste au fil des années à peu près inchangé : nous développons les outils de demain capables de vérifier seulement les circuits d'hier.

Une autre solution consiste à s'affranchir de la vérification fonctionnelle en produisant des circuits corrects par construction. La section suivante décrit comment fonctionnent de telles approches et quelles sont leur limites.

2.4 Circuits corrects par construction

Construire un circuit correct par construction à partir de sa spécification est un des challenges les plus prometteurs de la vérification. Réaliser ceci de manière efficace permettrait d'améliorer l'efficacité du flot de conception en supprimant deux étape extrêmement coûteuses : l'écriture de la description HDL, et sa vérification fonctionnelle.

L'étude de synthèse automatique de circuits à partir de spécifications débuta en 1982 lorsque Church posa le problème suivant : "étant donné une spécification, existe-t-il une réalisation satisfaisant celle-ci ?". Ce problème a une solution malheureusement de complexité triplement exponentielle qui se traduit à plusieurs niveaux :

- explosion de la mémoire utilisée
- explosion du temps de synthèse
- le circuit résultant est peu efficace (surface utilisée importante et fréquence faible)

Toute les méthodes explorées jusqu'à aujourd'hui réduisent la complexité du problème soit en restreignant le type de propriétés pris en compte, soit en se focalisant sur un seul type précis de circuit. Toutefois, les approches conservent une complexité exponentielle ou au mieux polynomiale, ce qui contraint leur application sur des circuits jouets servant d'exemples pour la validation des méthodes proposées.

Bien qu'il existe des méthodes de complexité linéaire pour des spécification booléennes, il n'existe pas à notre connaissance de telle méthode pour des spécifications temporelles. Par exemple Kukula et Shiple décrivent dans [KS00] comment synthétiser efficacement des relations mathématiques en circuit combinatoires. L'approche transforme la relation T(x,y) (où x sont les entrées et y les sorties du circuit) en un FBDD (Free-BDD) d'où est extrait le circuit final. La taille du circuit est proportionnelle à celle du FBDD et il convient donc d'optimiser au maximum celui-ci. Une preuve est détaillée afin de prouver que le circuit obtenu respecte la relation de départ.

Toujours sur la base d'expressions mathématiques, Aziz *et al.* décrivent dans [ABBSV00] comment synthétiser des circuits séquentiels à partir de formules logiques S1S. S1S est une logique du second ordre sur les entiers permettant de décrire efficacement des systèmes séquentiels. La formule est transformée en automate d'états fini qui sera lui même synthétisé en description matérielle au niveau portes. Pour cela, la relation fondamentale suivante est utilisée : *un ω-langage est descriptible en S1S si et seulement s'il est reconnu par un automate de Büchi.* Autrement dit, toute spécification fournie en S1S possède un automate de Büchi qui peut être synthétisée en une netlist. La méthode souffre d'une forte complexité due à l'utilisation de négations et de déterminisations (si nécessaire) d'automates de Bûchi. Même si l'approche se restreint à la synthèse de composants simples (mais aux fonctionnalités critiques) dans un système global plus complexe, l'étude formelle est très intéressante et permet d'établir un parallèle entre : logique S1S, automates, et construction de netlists.

Une approche originale est décrite par Greaves [Gre04]. Elle s'intéresse à la synthèse de petits composants. La spécification est fournie en entrée et un solveur SAT est utilisé pour produire la chaîne de bits qui va servir au placement routage sur un FPGA donné. Les propriétés sont de la forme A$\Rightarrow$next(B). L'approche n'est pas encore automatique, mais quelques expériences ont déjà été tentées et ont montré la faisabilité de cette méthode. La limitation est cependant forte à la fois dans le type de propriété et dans la complexité qui peut être prise en compte. Il semble donc difficile pour une telle méthode de concurrencer les autres approches sans améliorations conséquentes.

La spécification peut être fournie sous forme d'un ensemble de règles. Nous présentons ici deux méthodes utilisant des grammaires BNF comme support de spécification, dans deux contextes différents.

L'outil ClairVoyant développé par Seawright et Brewer [SB94] synthétise automatiquement des descriptions HDL au niveau RTL à partir d'une spécification écrite en PBS (Production-based Specification). PBS est un langage identique en de nombreux points aux expressions régulières. L'approche consiste à transformer la spécification en BDD, puis à coder ce BDD en RTL. Les résultats expérimentaux sont bons pour des circuits simples : construction rapide et circuits fonctionnels. Deux méthodes permettent d'optimiser le circuit produit : analyse des états atteignables et analyse des actions partageant les même ressources (ou étant activées en même temps).

Au lieu d'appliquer des restrictions sur le type de propriétés pris en compte, Öberg se focalise sur la synthèse de protocoles de communication [ÖKH96, Öbe99]. Un langage PRO-GRAM a été créé pour décrire le protocole sous forme de grammaire BNF. Alors que dans ClairVoyant la description du système s'effectue au cycle près, PROGRAM permet une approche de plus haut niveau et donc une spécification plus concise. La méthode est basée sur l'utilisation de DAG (Graphe acycliques orientés) et une exploration de l'espace d'états pour analyser tous les comportements possibles du circuit. Le temps de synthèse de la machiné à états explose lorsque les circuits deviennent complexes (3heures et 30 minutes pour un code VHDL de 3400 lignes). De plus, la synthèse est aussi particulièrement sensible à la taille des vecteurs de bits manipulés, toujours à cause de l'exploration de l'espace d'états que cela implique.

Siegmund *et al.* s'intéressent aussi à la synthèse d'interfaces de communications à partir de protocoles dans [SM02]. L'idée est de partir d'une description du protocole de communication entre 2 composants en SV. SV est un langage basé sur SystemC qui permet de décrire à haut niveau le protocole de communication entre les composants. Ce langage est basé sur des transmissions de communication au travers de canaux abstraits. Une transmission peut être une écriture, l'envoi d'un paquet spécifique etc. La description est alors synthétisée en une description SystemC. La partie SV est analysée et un PFG (Protocol Flow Graph) est construit. Un PFG donne la définition du protocole de communication. Celui-ci est construit par combinaison de PFG primitifs prédéfinis. Les descriptions SystemC synthétisables sont obtenues en transformant ce PFG en deux FSMs : une pour chaque interface.

Des applications sur des exemples complexes montrent l'efficacité de la méthode. Néanmoins, aucun résultat sur le temps de synthèse n'est donné. L'algorithme utilise notamment une déterminisation d'automates, il est donc possible que les temps de synthèse explosent avec la complexité du protocole.

Plusieurs approches ont été définies afin de partir directement de propriétés temporelles écrites dans des langages standards tels que PSL ou SVA. Ceci fournit un pouvoir d'expression meilleur que dans les cas précédents, et permet surtout à l'utilisateur de concevoir les spécifications de manière plus intuitive.

Bien que n'ayant pas pour but principal la synthèse de spécifications, Eveking *et al.* présentent dans [SNBE07a] une méthode pour transformer un ensemble de propriétés en un composant facile à vérifier : le *cando-object*. L'approche se restreint aux propriétés temporelles bornées de PSL_{ss}. Les propriétés sont mises sous une forme normalisée et une analyse statique est appliquée afin de calculer la cohérence de la spécification. Pour cela, il faut vérifier qu'à tout instant, aucun signal n'est contraint à deux valeurs différentes à

travers la spécification. La méthode est efficace et présente plusieurs similarités avec notre approche. Nous y reviendrons plus en détails dans le chapitre 5. Des entrées additionnelles sont ajoutées au composant afin de pouvoir utiliser une source aléatoire externe. Ceci est utile lorsque certains signaux ne sont pas contraints. Schickel détaille la synthèse d'un contrôleur de cache à l'aide de cette approche dans [SOSE08].

Toujours à base de propriétés LTL, Bloem *et al.* décrivent une approche permettant la synthèse de spécifications temporelles en descriptions HDL synthétisables [JB06, BGJ+07a]. Les propriétés temporelles sont de type GR(1) (Generalized Reactivity(1)), qui est un sous ensemble de LTL. Dans ce sous-ensemble, les propriétés ont la forme suivante :

$$(\Box\Diamond \mathrm{p}_1 \wedge ... \wedge \Box\Diamond \mathrm{p}_m) \rightarrow (\Box\Diamond \mathrm{q}_1 \wedge ... \wedge \Box\Diamond \mathrm{q}_n)$$

où chaque p_i et q_j sont des expressions booléennes[5][6]. Cette restriction permet de réduire la complexité de l'approche de triplement exponentielle à N^3 où N est le nombre d'états du circuit. L'approche utilise la théorie des jeux pour synthétiser la spécification. L'idée consiste à calculer tous les comportements possibles permis par la spécification sous toutes les actions possibles de l'environnement. On obtient un arbre de tous les comportements corrects. Celui-ci est codé en dur pour produire le circuit final. Le circuit se compose d'un bloc logique dont la taille explose avec la complexité de la spécification. Celui-ci est entouré de deux bancs de registres pour les entrées et sorties. Les fréquences obtenues ne sont pas fournies, mais souffrent sûrement de manière importante de ce déséquilibre entre quantité de cellules logiques et de registres. L'outil Lily supporte l'approche décrite ici. La synthèse du bus AMBA est présentée en détails dans [BGJ+07b]. Filiot *et al.* présentent dans [FJR09] l'outil Acaccia qui est une amélioration de Lily apportant de meilleurs résultats expérimentaux. D'autres approches du même type peuvent être trouvées dans [Reg05, SF07].

Alors que les approches présentées jusqu'à maintenant utilisent des propriétés de type LTL comme base de la synthèse, Heymans utilise ASP (Answer Set Programming) pour synthétiser des squelettes de synchronisation pour programmes [HNV05]. L'idée est donc de synthétiser non pas des propriétés LTL, mais CTL permettant d'exprimer des propriétés sur des programmes concurrents. Dans un premier temps, un modèle de la spécification CTL est construit en utilisant ASP. Ensuite, une analyse de cohérence est effectuée et le squelette de synchronisation est extrait.

Les améliorations qu'apporterait une méthode synthétisant efficacement des circuits à partir de leurs spécifications sont telles, que des efforts conséquents sont actuellement développés dans ce domaine. Les seules solutions partielles passent, soit par une focalisation sur des circuits de contrôle ou de protocoles, soit par une restriction sur le type de propriété pris en compte.

Dans les deux cas, le pouvoir des méthodes de synthèse est amoindri, et l'intérêt résultant limité. Nous proposons dans ce document une méthode de synthèse dont les complexités de l'approche, et du circuit final, sont linéaires par rapport à la spécification. Ceci rend alors possible une vraie synthèse de spécification PSL_{ss} contenant des centaines de propriétés complexes.

[5] $\Diamond$: opérateur eventually!

[6] $\Box$: opérateur always

Chapitre 3

Synthèse de propriétés temporelles

Sommaire

3.1 Précisions préliminaires . . . 38
3.2 Moniteurs . . . 39
3.2.1 Moniteurs primitifs . . . 40
3.2.2 Création d'un moniteur complexe . . . 41
3.2.3 Application à l'évaluation de performances . . . 44
3.2.4 Résultats en synthèse . . . 44
3.2.5 Bilan . . . 46
3.3 Générateurs méthode 1 : Horus . . . 46
3.3.1 Générateurs Booléen et FL . . . 46
3.3.2 Générateurs SEREs . . . 51
3.3.3 Générateurs et négation de propriétés . . . 59
3.4 Générateurs méthode 2 : MYGEN . . . 61
3.4.1 Concepts préliminaires . . . 61
3.4.2 Synthèse du générateur . . . 62
3.4.3 Le Générateur . . . 63
3.4.4 Résultats expérimentaux . . . 65
3.5 Bloc aléatoire embarqué . . . 69
3.5.1 Le composant LFSR . . . 69
3.5.2 Le composant : automate cellulaire . . . 70
3.5.3 Production de nombres aléatoires sur un intervalle quelconque . . . 73
3.6 Bilan . . . 75

Ce chapitre présente la méthode utilisée par la plateforme Horus pour la synthèse de propriétés temporelles : assertions et hypothèses. Le concept est identique pour ces deux types de composants et repose sur l'utilisation d'une bibliothèque d'éléments primitifs, interconnectés selon l'arbre syntaxique de la formule temporelle considérée.

3.1 Précisions préliminaires

Tout au long de ce document, des expérimentations ont été effectuées afin de valider les différents outils et les circuits obtenus en synthèse. Le contexte dans lequel se sont déroulées toutes les expérimentations est identique pour tous les résultats obtenus.

Les données logicielles (temps d'exécution de programmes, mémoire utilisée etc.) sont données pour un PC portable équipé d'un processeur Dual Core cadencé à 2Ghz et d'une mémoire vive de 2Go. Le système d'exploitation est un Linux Mandriva 2008. Les outils développés dans ces travaux ont été implémentés uniquement en C ou Java.

Tous les résultats en synthèse ont été obtenus en utilisant le logiciel QuartusII 8.0 avec une optimisation de synthèse "balanced" (effort d'optimisation équilibré entre fréquence maximale et surface minimale). La plateforme cible est un FPGA de type CycloneII EP2C35F672C6 [Cor05] monté sur une carte Altera DE2. Sur cette plateforme, la fréquence maximale de fonctionnement est de 420,17 Mhz. Lorsque les mesures en fréquences rapportent cette valeur, alors la fréquence peut être potentiellement plus élevée qu'elle n'y paraît. Tous les résultats obtenus sur FPGA sont composés de trois caractéristiques :

- **LCs (Logical Cells)** : nombre de cellules logiques. La cellule logique (nommée parfois LE : Logical Element) est l'élément de base des FPGA de type CycloneII. Elle est formée d'une LUT (Look-Up Table) à quatre entrées, un élément de mémorisation et de la logique pour la propagation rapide de retenues [Cor05].
- **FFs (Flip-flops)** : nombre de registres utilisés.
- **Freq. (Maximum Frequency)** : fréquence maximale du circuit pour le FPGA CycloneII utilisé ici. Lorsqu'un composant ne contient pas de partie séquentielle, l'information de fréquence n'est pas pertinente. Dans ce cas, le temps du chemin critique du circuit est fournit.

Les expérimentations utilisent deux ensembles de propriétés fournies dans l'annexe B.1 : Bench_FLBool et Bench_SERE. Le premier contient des propriétés FLs et booléennes, alors que le second contient uniquement des séquences SEREs. L'ensemble Bench_FLBool est divisé en trois parties :

- **PRIM** : cet ensemble est formé des propriétés numérotées de 0 à 18. Il contient tous les opérateurs de base de PSL (booléen et FL).
- **CPX** : propriétés 19 à 31. Elles modélisent un ensemble caractéristique de diverses propriétés complexes utilisées dans l'industrie.
- **LIM** : propriétés 32 à 40, utilisées pour tester les limites de l'outil MyGen. Cet ensemble contient 4 types de propriétés : next[i], next_e[1..i], next_event[i], avec $i \in \{128, 256, 512\}$.

L'ensemble Bench_SERE est organisé de manière similaire :

- **SERE_PRIM** : cet ensemble contient les propriétés numérotées de 0 à 7. Il contient tous les opérateurs primitifs SERE.
- **SERE_CPX** : propriétés 8 à 17. Elles modélisent un ensemble caractéristique de diverses propriétés complexes utilisées dans l'industrie.

– **SERE_LIM** : propriétés 18 à 20, utilisées pour tester les limites de l'outil MyGen. Cet ensemble contient des répétitions consécutives [*i], avec $i \in \{128, 256, 512\}$.

Sauf mention contraire, tous les générateurs ont été synthétisés avec un bloc aléatoire embarqué de type LFSR (*cf.* section 3.5) linéaire de 32 bits. Ce LFSR est générique et contient un ensemble de Taps prédéfinis quel que soit le nombre de bits entre 2 et 32. Lorsque le nombre de bits varie entre 2 et 32, le nombre de cellules logiques et de registres du LFSR est égal au nombre de bits de celui-ci. Sa fréquence est toujours maximale et vaut 420,17Mhz. Le tableau 3.1 donne les caractéristiques du LFSR générique pour différents nombres de bits.

Nombre de bits	Polynôme caractéristique	Nombre de bits	Polynôme caractéristique
1	-	17	$x^{16}+x^2$
2	$x+1$	18	$x^{17}+x^6$
3	x^2+1	19	$x^{18}+x^4+x+1$
4	x^3+1	20	$x^{19}+x^2$
5	x^4+x	21	$x^{20}+x$
6	x^5+1	22	$x^{21}+1$
7	x^6+1	23	$x^{22}+x^4$
8	$x^7+x^3+x^2+x$	24	$x^{23}+x^3+x^2+1$
9	x^8+x^3	25	$x^{24}+x^2$
10	x^9+x^2	26	$x^{25}+x^5+x+1$
11	$x^{10}+x$	27	$x^{26}+x^4+x+1$
12	$x^{11}+x^5+x^3+1$	28	$x^{27}+x^2$
13	$x^{12}+x^3+x^2+1$	29	$x^{28}+x$
14	$x^{13}+x^4+x^2+1$	30	$x^{29}+x^5+x^3+1$
15	$x^{14}+1$	31	$x^{30}+x^2$
16	$x^{15}+x^4+x^2+x$	32	$x^{31}+x^6+x^5+x$

Table 3.1 – Taps pour le LFSR générique

3.2 Moniteurs

La première méthode à voir le jour fut celle consacrée à la construction de moniteurs. Les détails sont donnés dans [BLOF06]. L'approche est totalement modulaire : un moniteur primitif est défini pour chaque opérateur élémentaire de PSL (ou de SVA), et chacun de ceux-ci est interconnecté en suivant l'arbre syntaxique de la formule.

Depuis, de nombreuses améliorations ont été apportées pour augmenter l'efficacité des moniteurs matériels obtenus après synthèse. Ces travaux ont été réalisés par Katell Morin-Allory. Nous présentons ici cette nouvelle version des moniteurs et détaillons son fonctionnement.

3.2.1 Moniteurs primitifs

Contrairement à l'ancienne version des moniteurs qui ne possédait qu'un seul type de moniteur primitif, nous distinguons maintenant deux types de moniteurs : les **connecteurs** et les **observateurs**. Un connecteur est utilisé pour l'activation d'une sous propriété (ses opérandes sont de type FL), alors qu'un observateur détecte toute violation de la propriété. Il possède uniquement des opérandes booléens.

Le tableau 3.2 donne le type (connecteur ou observateur) de chaque moniteur primitif de PSL. Le moniteur de base mnt_Signal est le moniteur correspondant à l'observation d'un simple signal booléen.

Observateur	mnt_Signal, iff, eventually!,never, next_e, next_event_e, before
Connecteur	→, and, or, always, next!, next_a, next_event, next_event_a, until

Table 3.2 – Type pour chaque moniteur primitif de PSL

Tous les connecteurs possèdent une même interface générique, comme le montre la figure 3.1. Ils prennent en entrée les signaux *Clk* et *Reset_n* afin de synchroniser le moniteur sur le circuit à tester, le signal d'activation *Start* permettant de lancer le moniteur ; et une ou deux entrées *Expr* et *Cond* pour l'observation des opérandes. Ils fournissent deux signaux de sortie *Trigger* et *Pending*. Le premier déclenche la vérification de la sous-propriété de l'opérande courant alors que le second permet simplement de propager les informations sur l'état de vérification de l'opérande : en cours de vérification ou non.

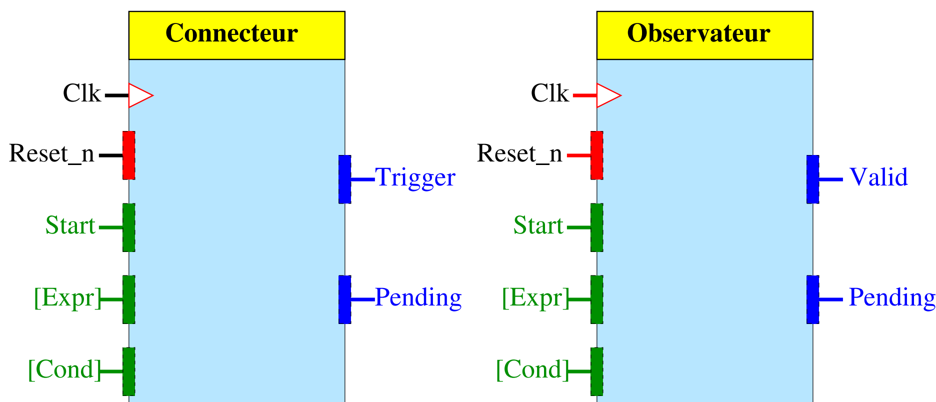

FIGURE 3.1 – Interface générique pour les moniteurs primitifs

De la même manière, comme le montre la figure 3.1, les observateurs possèdent une interface générique. La seule différence se situe au niveau de la sortie *Trigger* qui est remplacée par un port *Valid*. Le couple de ports (*Valid*, *Pending*) renseigne sur l'état de la propriété : pendante (1,1), vérifiée (1,1), fortement vérifiée (1,0), violée (0,1) ou (0,0) si la propriété n'a pas commencé à être vérifiée.

Tout opérande de type booléen est connecté aux entrées *Expr* ou *Cond* d'un moniteur-primitif. Si l'opérande est de type FL, alors le moniteur de la sous-propriété correspondant à ce moniteur est construit. Celui-ci est connecté au moniteur courant via le port *Trigger*. Ainsi le moniteur courant pourra déclencher la vérification de la sous propriété.

Les moniteurs primitifs possèdent tous une architecture basée sur le même modèle, comme le montre la figure 3.2. Un bloc SEM implémente la sémantique de l'opérateur PSL et un bloc CTRL gère le contrôle du moniteur : initialisation des signaux, activation et désactivation du bloc SEM. Les opérateurs PSL paramétriques sont traités à l'aide de paramètres génériques au niveau HDL :

- `OP_TYPE` : pour définir le type d'opérateur (fort, faible, recouvrant ou non recouvrant).
- `TIMING`, `TIMING_LOW` etc. : pour les next, next_a etc.

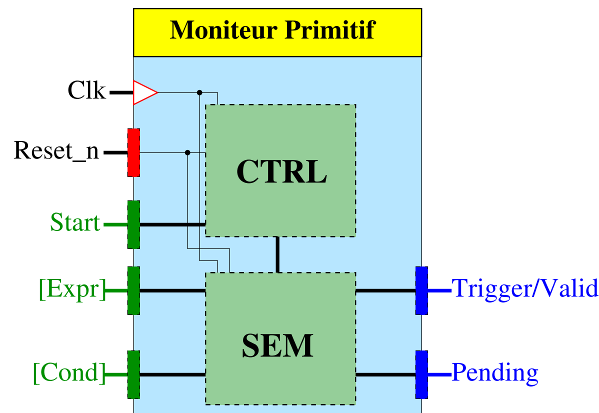

FIGURE 3.2 – Architecture générique pour les moniteurs primitifs

3.2.2 Création d'un moniteur complexe

La création d'un moniteur, nommé *top_monitor*, pour une propriété complexe, s'obtient en connectant les moniteurs primitifs entre-eux. Cela s'effectue en suivant le schéma de l'arbre syntaxique. À chaque noeud *n* de l'arbre correspond un moniteur primitif *mon*. Les fils d'un noeud sont notés *n.left* pour le fils gauche et *n.right* pour le fils droit (auxquels correspondent les moniteurs *mon_left* et *mon_right*). Ces noeuds fils représentent respectivement les opérandes gauche et droit de l'opérateur du noeud *n*.

Le noeud le plus en bas à droite de l'arbre représente l'opérateur qui sera activé en dernier dans la propriété. Chaque moniteur complexe contient ainsi un seul observateur correspondant à ce noeud spécifique. Les noeuds restants sont de type connecteur. La procédure d'interconnexion est fournie par l'algorithme 1. C'est une procédure récursive qui est lancée à partir de la racine de l'arbre syntaxique et qui connecte chaque moniteur primitif à son ou ses fils éventuels.

- Ligne 2 : le composant HDL "*mon*" correspondant au noeud *n* est construit grâce à la fonction `Create_Crt_Monitor`. Celle-ci créée l'entité HDL correspondante.
- Ligne 3 : dans le cas où le noeud est de type observateur (noeud en bas à droite de l'arbre), le composant est un observateur. La sortie *Valid* est connectée au port de même nom du moniteur global. L'arbre des sorties *Pending* est construit grâce à la fonction `Build_Pending`. L'obtention du *Pending* final s'effectue simplement en connectant tous les signaux *Pending* des moniteurs primitifs à une porte OR. La sortie de celle-ci est connectée sur le port *Pending* du moniteur global *top_monitor*.
- Lignes 6, 7 : le nom du signal observé pour l'opérande gauche est récupéré et un port d'entrée portant ce nom est ajouté au moniteur global mnt_monitor. Ce dernier

est connecté au port *Expr* du moniteur courant. Dans le cas où l'opérande courant est binaire, la même opération est effectuée pour l'opérande droit (lignes 8 à 10).

- Lignes 13 à 33 : la connexion d'un composant de type connecteur dépend du type des opérandes (des noeuds fils) du moniteur courant. Tout d'abord la fonction `Build_Monitor` est appelée sur le ou les fils non booléens. Ceci permet de construire les composants HDL fils et ainsi de pouvoir effectuer les connexions de signaux entre chaque port.
- Lignes 9, 10, 15, 16, 20, 21 : dans le cas où l'opérande est un signal booléen, le nom du signal est récupéré, puis un port d'entrée portant ce nom est ajouté au moniteur global *top_monitor*. Le port *Expr* ou *Cond* (considérant respectivement un opérande gauche ou droit) du moniteur courant est connecté au port qui vient d'être créé
- Lignes 18, 23, 27, 28, 32 : si l'opérande est un composant FL, alors le port *Trigger* du moniteur courant est connecté au port *Start* du moniteur fils.

Considérons la propriété $A1_{cdt}$ suivante :

$$\textsf{property } A1_{cdt} \textsf{ is assert always } (Req \rightarrow (Busy \textsf{ until! } Ack))$$

La propriété $A1_{cdt}$ permet de vérifier que lorsqu'une requête de transmission de données est reçue par le composant CDT, alors celui-ci passe en mode transmission (signal *Busy* activé) jusqu'à ce que le composant externe termine le transfert en passant le signal *Ack* à '1'.

L'architecture du moniteur correspondant pour $A1_{cdt}$ est donnée figure 3.3. Les moniteurs complexes possèdent tous le même type d'interface : deux signaux de synchronisation *Clk* et *Reset_n*, un ensemble d'entrée pour les signaux observés et deux sorties *Valid* et *Pending* fournissant l'état de la propriété.

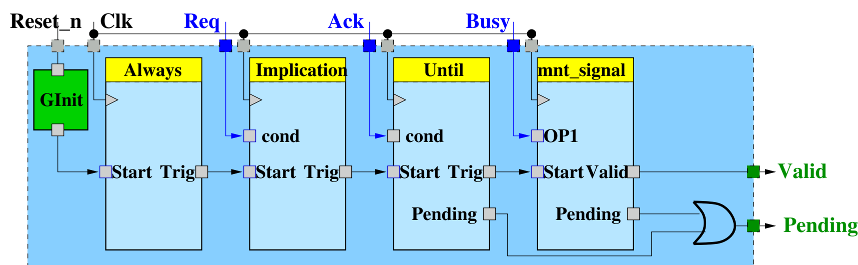

FIGURE 3.3 – Architecture du moniteur $A1_{cdt}$

Les fils de connexion entre moniteurs primitifs sont des bits. Lorsqu'un moniteur est actif (port *Start* actif), on dit qu'il possède un jeton. Ce jeton est transmis entre moniteurs primitifs via les ports *Trigger* et *Start*. Comme plusieurs moniteurs primitifs peuvent être actifs à un moment donné, plusieurs jetons peuvent être présents dans le circuit. La valeur d'un jeton est '1' (moniteur primitif actif), ou '0' (moniteur inactif).

Nous illustrons le fonctionnement du moniteur complexe $A1_{cdt}$ sur la trace de la figure 3.4. Au cycle 0, nous supposons que le circuit vient d'être réinitialisé. Le moniteur always reçoit un jeton et le transmet directement au moniteur suivant ($\rightarrow$). Le moniteur always émettra continuellement des jetons jusqu'à une prochaine réinitialisation du circuit. Celui-ci permet de lancer la vérification de la propriété à chaque cycle.

Au cycle $\sharp 1$, le signal *Req* passe à '1'. La partie de gauche de l'implication est vérifiée, donc la partie droite doit l'être aussi. Cela cause la transmission d'un jeton du moniteur

Algorithme 1 Construit un moniteur pour une propriété temporelle donnée

```
1:  BUILD_MONITOR(node : n)
2:  mon ← Create_Crt_Monitor(n) ;
3:  if Is_Observer(n) then
4:     Build_Pending(mon(pending),top_monitor(pending)) ;
5:     Connect(mon(valid),top_monitor(valid)) ;
6:     signal ← Get_Signal_Name(n.left) ;
7:     Connect(top_monitor(signal), mon(expr)) ;
8:     if Is_Binary_Op(n) then
9:        signal ← Get_Signal_Name(n.right) ;
10:       Connect(top_monitor(signal), mon(cond)) ;
11:    end if
12: else
13:    if Is_Binary_Op(n) then
14:       if Is_Boolean(n.left) and not Is_Boolean(n.right) then
15:          signal ← Get_Signal_Name(n.left) ;
16:          Connect(top_monitor(signal), mon(expr)) ;
17:          mon_right ← BUILD_MONITOR(n.right) ;
18:          Connect(mon(trigger), mon_right(start)) ;
19:       else if not Is_Boolean(n.left) and Is_Boolean(n.right) then
20:          signal ← Get_Signal_Name(n.right) ;
21:          Connect(top_monitor(signal), mon(cond)) ;
22:          mon_left ← BUILD_MONITOR(n.left) ;
23:          Connect(mon(trigger), mon_left(start)) ;
24:       else if not Is_Boolean(n.left) and not Is_Boolean(n.right) then
25:          mon_left ← BUILD_MONITOR(n.left) ;
26:          mon_right ← BUILD_MONITOR(n.right) ;
27:          Connect(mon(trigger), mon_left(start)) ;
28:          Connect(mon(trigger), mon_right(start)) ;
29:       end if
30:    else
31:       mon_left ← BUILD_MONITOR(n.left) ;
32:       Connect(mon(trigger), mon_left(start)) ;
33:    end if
34: end if
35: return mon
```

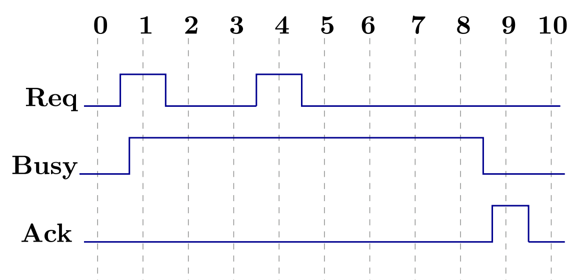

FIGURE 3.4 – Trace satisfaisant l'assertion $A1_{cdt}$

implication au moniteur suivant (until). Le moniteur until transmet un jeton au moniteur mnt_Signal tant que le signal *Ack* n'apparaît pas. Ceci permet de vérifier que le signal *Busy* est bien actif à chaque cycle. Dès que *Ack* passe à '1' (au cycle ♯9), le moniteur until arrête de transmettre des jetons. Le moniteur mnt_Signal n'en possède plus non plus.

Comme le signal *Req* repasse à '1' au cycle ♯4, un autre jeton est introduit dans le moniteur until. Celui-ci en possède déjà un, et comme aucune différence n'est faite entre les jetons, rien de spécial ne se passe, le until reste actif jusqu'à l'observation du signal *Ack*. Au cycle ♯10, seul le générateur always possède un jeton.

3.2.3 Application à l'évaluation de performances

Les moniteurs peuvent être particulièrement adaptés à l'évaluation de certaines caractéristiques du circuit, non directement liées à la fonctionnalité du circuit. Les propriétés temporelles se prêtent bien à la description concise de scénarios complexes. Il est alors possible d'utiliser les moniteurs pour dénombrer certains types de scénarios. Dans le cas de l'arbitre, la propriété $\text{Nb_Access}_{arbitre}$ suivante permet de compter le nombre d'accès à la ressource partagée durant une phase de test :

$$\textsf{property}\ \text{Nb_Access}_{arbitre}\ \textsf{is cover always rose}(\text{tk}_0)\,;$$

Pour l'évaluation de performances, le mot clé cover de PSL est utilisé. Chaque validation de la propriété incrémentera un compteur permettant de dénombrer le nombre total de fois où la propriété a été validée. De la même manière, la propriété $\text{NB_Collisions}_{arbitre}(\text{i})$ indique le nombre de fois où une unité demande la ressource alors que celle-ci est déjà allouée.

for i 0 to P loop
  property $\text{NB_Collisions}_{arbitre}(\text{i})$ is cover always(ask_i and used) ;
end loop ;

Pour cela, une version légèrement modifiée des moniteurs est utilisée. Lorsqu'un moniteur classique détecte une erreur, une information de sévérité WARNING est levée par le simulateur. Dans le cas de l'évaluation de performances, ce niveau est abaissé à NOTE pour une simple notification.

La bibliothèque est aussi modifiée. Au lieu d'utiliser des signaux *Valid* qui sont à '1' par défaut et permettent de détecter une erreur, ceux-ci sont placés à '0' par défaut. Ils détectent alors toute satisfaction sans vacuité de la propriété. Un exemple d'application est détaillé dans la section 7.4.5.

Les moniteurs étant synthétisables, l'évaluation de performances peut être effectuée aussi bien en émulation qu'en simulation.

3.2.4 Résultats en synthèse

Les moniteurs ont été synthétisés sur FPGA afin d'évaluer leur coût en surface et leur fréquences de fonctionnement.

Les propriétés utilisées pour les expérimentations sont fournies dans l'Annexe B.1. Les courbes de la figure 3.5 montrent que les moniteurs obtenus pour les opérateurs primitifs de PSL (19 premières propriétés) sont très petits et leur fréquence maximale d'utilisation

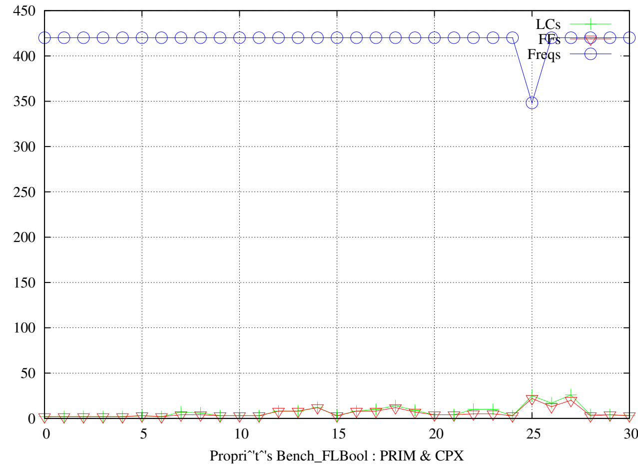

FIGURE 3.5 – Résultats de synthèse des moniteurs Horus

est toujours très élevée, puisque supérieure à 350 Mhz, ce qui permet de tester le circuit à sa vitesse réelle.

Pour les propriétés complexes, la surface occupée par les moniteurs est largement influencée par l'utilisation de registres à décalage dans certains moniteurs primitifs tels que : next!, next_event...

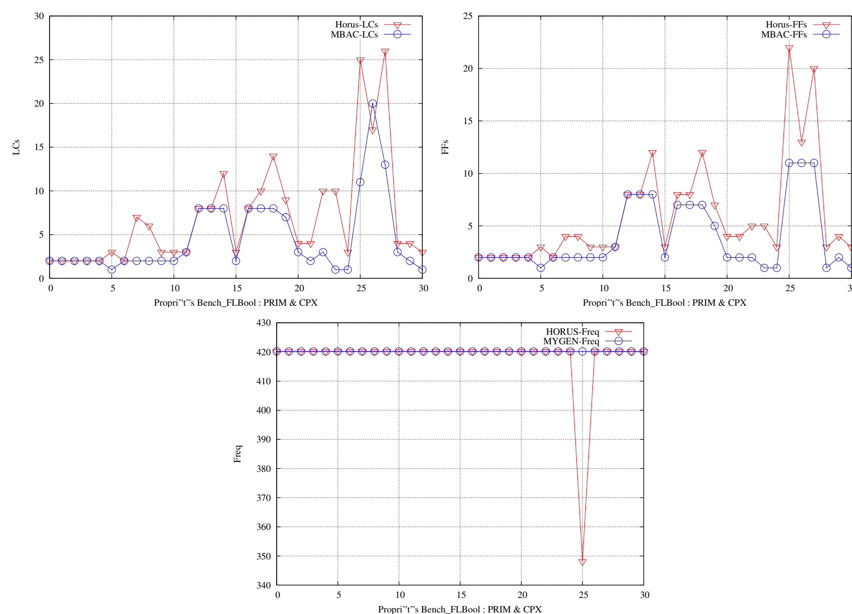

FIGURE 3.6 – Comparaisons des moniteurs Horus et MBAC

Les courbes de la figure 3.6 donnent les résultats obtenus en synthèse pour les moniteurs MBAC et Horus. Malgré la forte ressemblance des résultats obtenus, Horus accuse une très légère faiblesse pour certains opérateurs qui occupent quelques cellules logiques et registres de plus. Les fréquences sont globalement meilleures pour les moniteurs obtenus avec MBAC.

3.2.5 Bilan

Cette approche de conception de moniteurs est complètement modulaire. Elle possède les caractéristiques suivantes :
- la complexité du moniteur est linéaire en fonction de la complexité de la propriété
- la création du moniteur à partir de la propriété est quasi-instantanée
- les moniteurs étant totalement paramétriques, cela permet de construire des bibliothèques pouvant être réutilisées.
- toute la méthode a été prouvée correcte par preuve de théorèmes. Le chapitre 6 est consacré à la preuve des composants de vérification Horus.

Jusqu'à présent, seules des propriétés FLs ont été synthétisées grâce à notre approche. L'adaptation aux propriétés SEREs est en cours de développement. L'approche est définie et une première version de l'implémentation est disponible dans la version actuelle de la plateforme Horus.

Même si la complexité des moniteurs est quasi-équivalente pour MBAC et Horus, l'outil MBAC produit des moniteurs sensiblement plus petits et surtout possédant des fréquences meilleures que pour Horus. A notre connaissance, ces deux approches de synthèse de propriétés en moniteurs matériels sont les plus efficaces aujourd'hui.

Alors qu'un moniteur observe des séquences de signaux pour s'assurer qu'ils vérifient une propriété donnée, l'idée qui fut le point de départ des travaux rapportés ici était la suivante : peut-on concevoir automatiquement un composant synthétisable produisant des signaux conformes à une propriété donnée ? Cette question a débouché sur la création d'un nouvel ensemble de composants de vérification, identiques sous de nombreux aspects aux moniteurs : les générateurs de vecteurs de tests.

La suite de ce chapitre présente deux méthodes de construction de générateurs : l'une à l'aide de la plateforme Horus, et l'autre à l'aide de l'outil MyGen.

3.3 Générateurs méthode 1 : Horus

La section précédente a détaillé comment transformer une assertion en un moniteur. Ceci permet de couvrir les aspects concernant la vérification comportementale des circuits, dans le cadre de l'Assertion Based Verification. Nous nous intéressons maintenant aux aspects liés à l'environnement du circuit sous test. Comment définir cet environnement, le spécifier et le modéliser de manière efficace ?

Le comportement de l'environnement du circuit sous test est spécifié à l'aide de propriétés temporelles de type hypothèse ("assumption"). Celles-ci sont alors synthétisées en générateurs, imitant le plus précisément possible le comportement de l'environnement réel, dans lequel sera plongé le circuit final. Les hypothèses sont transformées en composants appelés générateurs.

3.3.1 Générateurs Booléen et FL

3.3.1.1 Méthode de construction

Malgré quelques subtilités qui seront dévoilées dans la suite de cette section, les principes de synthèse de propriétés en générateurs sont identiques à ceux utilisés pour les moniteurs. Les différences tiennent principalement dans la bibliothèque de composants primitifs.

Interface des générateurs primitifs : La première version des générateurs a été développée dans le cadre de mes travaux. Ensuite, une redéfinition de la bibliothèque a été effectuée pour harmoniser les générateurs avec la dernière version des moniteurs . De la même manière que pour les moniteurs, les générateurs ont été modifiés au cours de cette thèse. Les détails de la méthode initiale sont donnés dans [OMAB06] et nous expliquons ici les concepts de la dernière méthode développée.

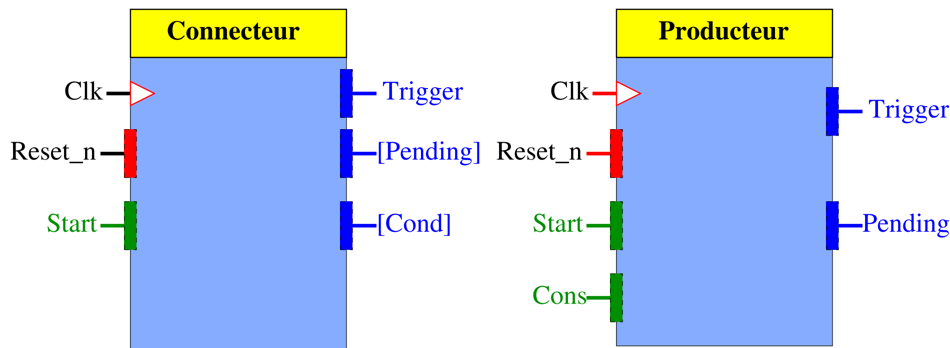

FIGURE 3.7 – Interface générique pour les générateurs primitifs

La bibliothèque de générateurs contient elle aussi deux types de composants : **connecteurs** et **producteurs**. Cependant les ensembles sont différents de ceux définis pour les moniteurs. Les générateurs de type connecteur regroupent tous les opérateurs de PSL. L'ensemble des producteurs est réduit à un seul élément qui est le générateur pour un simple signal : gnt_Signal.

Un générateur connecteur possède le même type d'interface que le moniteur connecteur. On distingue deux différences :

- *Clk*, *Reset_n*,*Start* : identique aux moniteurs
- le rôle de la sortie *Trigger* ne change pas et sert à la production de l'opérande gauche.
- le port de sortie *Cond* est ajouté pour la production de l'opérande droit.
- le port *Pending* indique si la génération est contrainte (port *Pending* actif), ou aléatoire.
- le port d'entrée *Cons* présent sur les Producteurs indique si le générateur doit produire un '0' ou un '1' dans le cas d'une génération contrainte.

Architecture des générateurs primitifs : Tous les générateurs (connecteurs et producteurs) possèdent une architecture composée d'au moins deux blocs : CTRL et SEM (*cf.* figure 3.8). La fonctionnalité de ces blocs est identique à celle des moniteurs .

Certains générateurs possèdent en plus un bloc ALEA. Celui-ci sert à produire des entiers de manière pseudo-aléatoire. Comme une propriété peut être satisfaite par une, plusieurs, ou même voire une infinité de traces, les générateurs doivent être capables de couvrir cet ensemble. Ceci est rendu possible via l'utilisation d'un bloc aléatoire ALEA. Il sert par exemple à déterminer le cycle où l'opérande droit du until sera produit.

Les générateurs possèdent une interface de sortie identique dans la structure aux observateurs, mais différente dans son utilisation.

Alors que sur le moniteur, les deux sorties *Trigger* et *Pending* indiquent l'état de la propriété, celles d'un générateur indiquent l'état de la génération du signal concerné. Lorsque la sortie *Pending* est active, alors la valeur sur le port *Trigger* est dite contrainte.

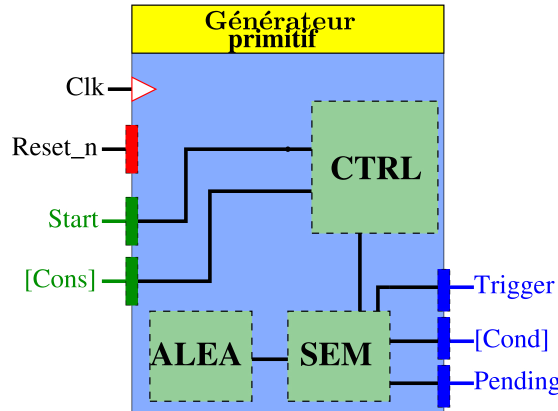

FIGURE 3.8 – Architecture pour un générateur primitif

Ceci signifie que cette valeur n'est pas produite par le bloc aléatoire, mais résulte du traitement par le bloc SEM.

Dans le cas où *Pending* est inactif, la sortie *Trigger* peut valoir '0' ou '1' sans compromettre la satisfaction de la propriété. Le port *Trigger* est dans ce cas là non contraint. L'utilité de l'information de contrainte sera discutée dans la section 4.1.1.

Dans le cas où *Pending* est actif, alors la valeur du signal produit est contrainte. La valeur de cette contrainte est donnée par le port *Trigger*. Dans le cas du `until`, le cycle futur où la génération de l'opérande droit sera produit est effectué aléatoirement. Jusqu'à ce cycle, l'opérande droit est contraint à '0'. La valeur des sorties *Trigger* et *Pending* sera donc (0,1).

Cette information est utilisée uniquement par les producteurs. Ceux-ci récupèrent l'information de contrainte et sa valeur associée sur les ports respectifs *Cons* et *Start*(*cf.* figure 3.9). Une utilisation plus poussée de ces ports est détaillée section 3.3.3, notamment dans le cas de la génération de la négation d'expressions booléennes.

L'architecture est complètement modulaire et entièrement paramétrable. Ainsi, il est possible de changer le module ALEA pour le remplacer par un autre permettant une meilleur modélisation de l'environnement sans avoir besoin de retoucher le bloc CTRL ni le bloc SEM.

Chaque générateur est paramétré via deux types de paramètres génériques[1] :

- **Dépendant de la formule** : ce sont les même que pour les moniteurs (OP_TYPE, TIMING etc.)
- **Liés à la génération** : ils permettent de choisir la taille des registres utilisés par le bloc ALEA pour produire des nombre pseudo-aléatoire (paramètre `MAX_SIZE`). Ceci permet d'exprimer des formules sur des intervalles de temps plus ou moins longs, tout en restant le plus optimal possible en terme de surface et de fréquence. Un paramètre `RANDOM` permet de définir le comportement des générateurs lorsqu'ils sont inactifs. Si ce paramètre vaut '0', alors les générateurs ne produisent que des '0' durant chaque cycle ou *Start* est inactif, sinon les signaux prennent aléatoirement les valeurs '0' ou '1' (toujours si *Start*='0'). Un paramètre SEED_INIT peut être utilisé pour fournir la graine au bloc aléatoire.

[1] Les paramètres génériques sont fixés par l'utilisateur via l'outil Horus.

Un autre aspect commun aux générateurs non booléens est l'utilisation d'un registre à décalage. Celui-ci est utilisé pour effectuer des prédictions sur la génération des opérandes. La taille de ce registre est liée à celle du LFSR et donc implicitement paramétrée via `MAX_SIZE`. La prédiction est effectuée en plaçant la valeur '1' à l'indice i du registre à décalage. Alors i cycles plus tard, le port *Trigger* prendra la valeur '1'. Le nombre i peut être par exemple fourni par le bloc ALEA. Cette utilisation permet la ré-entrance puisque N prédictions peuvent être faite si le registre à décalage est de taille N.

Par défaut, le bloc aléatoire est embarqué. Cela permet d'obtenir des générateurs indépendants de la technologie cible puisqu'aucune source aléatoire externe ne sera utilisée. En revanche, le bloc ALEA consommera de la surface et pourra avoir un impact indésirable sur la fréquence du circuit instrumenté final.

Une étude a été menée sur la génération de nombres aléatoires afin d'optimiser le bloc ALEA. Deux types de composants ont été intégrés aux générateurs : les LFSRs et les automates cellulaires (CAs). La section 3.5.1 est consacrée à ces travaux. Dans le cas où une source aléatoire externe est disponible, le bloc ALEA peut être retiré. Une version des générateurs possédant une entrée supplémentaire doit être utilisée afin de connecter la source aléatoire à ceux-ci.

Générateurs complexes : L'interface d'un générateur complexe possède en entrée deux signaux de synchronisation *Clk* et *Reset_n*. Les sorties sont composées de deux ensembles de ports : l'un regroupe tous les signaux à générer qui sont impliqués dans la propriété et l'autre regroupe tous les signaux de contraintes associées. L'algorithme 3.9 illustre un exemple d'une telle interface pour le générateur correspondant à la propriété $H1_{cdt}$. Cette propriété est identique à $A1_{cdt}$ définie section 3.2, mais en version hypothèse :

$$\text{Property } H1_{cdt} : \text{assume } \mathsf{always}\ (Req \rightarrow (Busy\ \mathsf{until!}\ Ack))$$

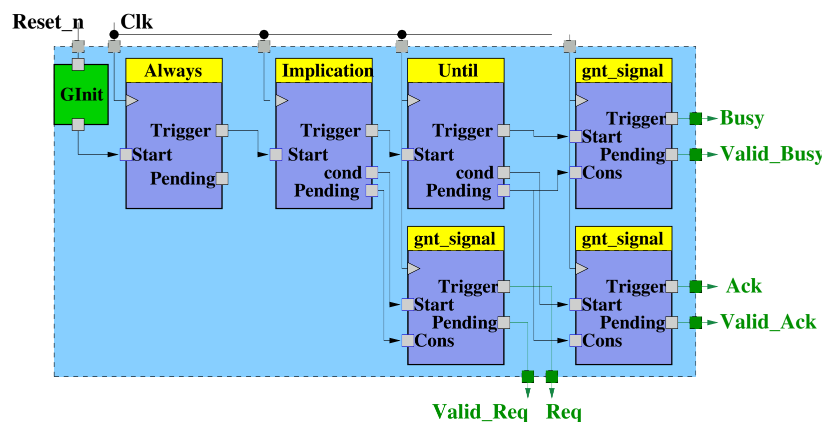

FIGURE 3.9 – Architecture du générateur $H1_{cdt}$

La construction d'un générateur complexe est basée sur la même idée que pour les moniteurs . L'arbre syntaxique est parcouru et les éléments primitifs sont interconnectés à l'aide de l'algorithme 2. Pour les générateurs, les feuilles de l'arbre syntaxique sont toujours modélisées par des producteurs.

– Ligne 2 : création du générateur courant à partir des informations contenues dans le noeud n.

– Lignes 3 à 6 : le noeud contient le nom du signal à produire. Le nom de ce signal est récupéré ligne 4. Le générateur courant est de type producteur. Son port de sortie *Trigger* est connecté au port `name` du générateur global. Le port *Pending* de la contrainte du signal est aussi passée sur un port de sortie du générateur global.
– Lignes 8 à 23 : connexion des connecteurs. Les ports *Trigger* et *Cond* sont connectés aux *Start* des opérandes. Dans le cas où les opérandes sont de type producteur, le port *Pending* du générateur courant est connecté au port *Cons* de l'opérande (lignes 14 et 17). Dans le cas où le fils du connecteur est un autre composant de type connecteur, il est inutile de connecter le port *Pending*. L'outil de synthèse supprimera ce port et toute la circuiterie associée.

Algorithme 2 Construit un générateur pour une propriété temporelle donnée

```
1: BUILD_GENERATOR(node : n)
2: gen←Create_Crt_Generator(n)
3: if Is_Leaf(n) then
4:    name←Get_Signal_Name(gen)
5:    Connect(gen(Trigger),top_generator(name))
6:    Connect(gen(Pending),top_generator(pending_name))
7: else
8:    if Is_Binary_Op(n) then
9:       gen_left←BUILD_GENERATOR(n.left)
10:      gen_right←BUILD_GENERATOR(n.right)
11:      Connect(gen(Trigger), gen_left(Start))
12:      Connect(gen(Cond), gen_right(Start))
13:      if Is_Leaf(n.left) then
14:         Connect(gen(Pending),gen_left(Cons))
15:      end if
16:      if Is_Leaf(n.right) then
17:         Connect(gen(Pending),gen_right(Cons))
18:      end if
19:   else
20:      gen_left←BUILD_GENERATOR(n.left)
21:      Connect(gen(Trigger), gen(Start))
22:      if Is_Leaf(n.left) then
23:         Connect(gen(Pending),gen_left(Cons))
24:      end if
25:   end if
26: end if
27: return gen
```

La figure 3.9 illustre l'architecture du générateur complexe pour la propriété $H1_{cdt}$.

3.3.1.2 Résultats expérimentaux

La figure 3.10 donne les résultats obtenus pour les générateurs concernant les propriétés Bench_FLBool. La quantité de logique combinatoire, ainsi que de registres, est faible

puisque leurs nombres respectifs sont toujours inférieurs à 80 et 50. Ceci permet d'obtenir des circuits très simples ayant de bonnes fréquences de fonctionnement.

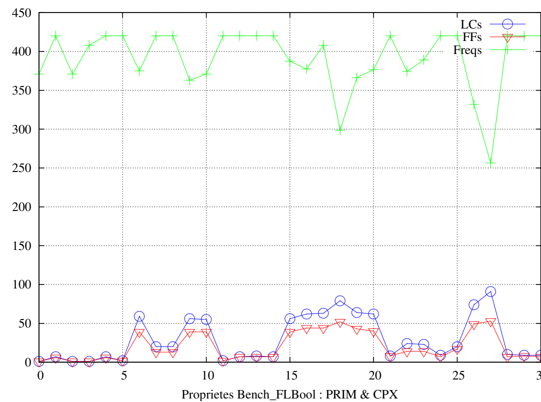

FIGURE 3.10 – Résultats en synthèse pour les générateurs Horus

La majeure partie de la complexité des circuits obtenus est dûe à l'utilisation d'un bloc aléatoire directement embarqué à l'intérieur des générateurs primitifs. Les analyses utilisent un LFSR classique sur 32 bits. Si une source aléatoire est disponible, il est possible de retirer ce bloc et d'obtenir des générateurs ne possédant que quelques registres et cellules logiques.

3.3.2 Générateurs SEREs

3.3.2.1 Introduction

La méthode de construction de générateurs pour les propriétés SEREs est identique à celle pour les FLs si la propriété SERE contient uniquement des répétitions de signaux (**pas de répétitions de séquences**) et ne contient **pas l'opérateur de parallélisation de séquences "&&"**.

Une bibliothèque de générateurs primitifs a été définie dans ce cas. Ils possèdent une interface et une architecture identiques aux générateurs FL. Tous les générateurs SERE sont de type connecteur. La méthode d'interconnexion est identique. Il est alors possible de connecter des générateurs FL à des générateurs SERE sans aucune manipulation particulière.

La figure 3.11 illustre l'architecture du générateur pour l'hypothèse $H2_{arbitre}$ suivante[2] :

$$\text{Property } H2_{arbitre} : \texttt{assume always } \{Ask \mid\rightarrow \{\text{true}[*]\,;Grant;Use[*5]\}\}$$

[2]Pour des raisons de clarté, les indices i de l'hypothèse $H2_{arbitre}$ ont été retirés dans cette section.

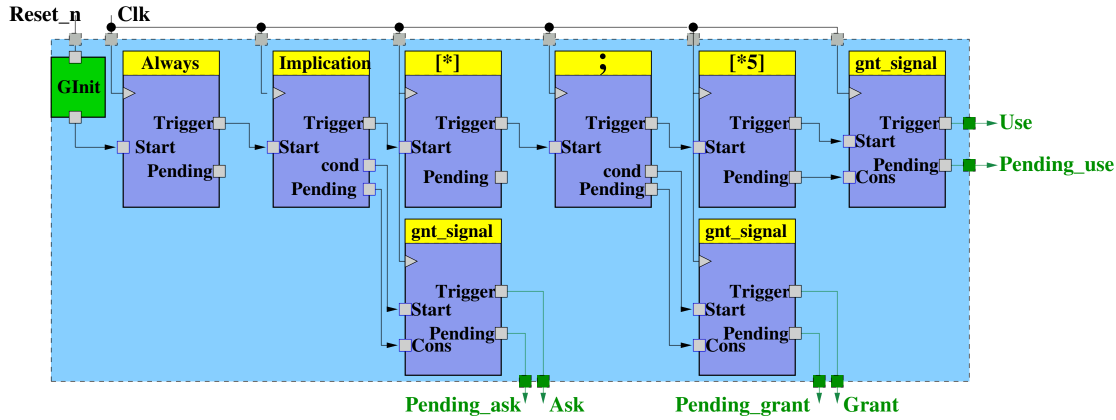

FIGURE 3.11 – Architecture du générateur pour l'hypothèse $H2_{arbitre}$

La propriété $H2_{arbitre}$ énonce qu'à chaque cycle où la i-ème unité demande l'accès à la ressource, un certain nombre de cycles s'écoule, jusqu'à ce que le composant $\mathsf{Arbiter}_P$ valide l'accès en activant *Grant*. Durant les 5 cycles suivants, la ressource est utilisée et la i-ème unité maintient le signal *Use* actif.

Cette propriété est particulière car en plus de contrôler les entrées du circuit $\mathsf{Arbiter}_P$, elle contrôle l'entrée *Use* de l'environnement.

Les deux sections suivantes décrivent les modifications à apporter à notre approche pour traiter d'une part les répétitions de séquences, puis l'opérateur "&&".

3.3.2.2 Générateurs pour répétition de séquences

Jusqu'à présent, la construction des circuits était linéaire et suivait directement le schéma de l'arbre syntaxique de la propriété. Or, dans le cas de la répétition de séquence, une boucle est introduite dans l'architecture du générateur. Celle-ci permet de connecter le dernier élément d'une séquence S sous la portée d'un opérateur de répétition SERE, à cet opérateur. Ce bouclage permet de déclencher la génération de la séquence répétée plusieurs fois. Prenons l'hypothèse $H3_{arbitre}$ suivante :

$$\text{Property } H3_{arbitre} : \text{assume } \{\{Ask;\text{true}[*]\,;Grant;Use[*5]\}[*3]\}$$
$$\text{Sequence S}=\{Ask;\text{true}[*]\,;Grant;Use[*5]\}$$

La propriété $H3_{arbitre}$ reprend le style de scénario de l'hypothèse $H2_{arbitre}$ définie précédemment. Cette fois, celui-ci est répété 3 fois dans son intégralité. La figure 3.12 donne l'architecture du générateur $H3_{arbitre}$. Le générateur [*3] encapsule la séquence répétée. Un nouveau port *Iter* est utilisé par le générateur de répétition de séquences.

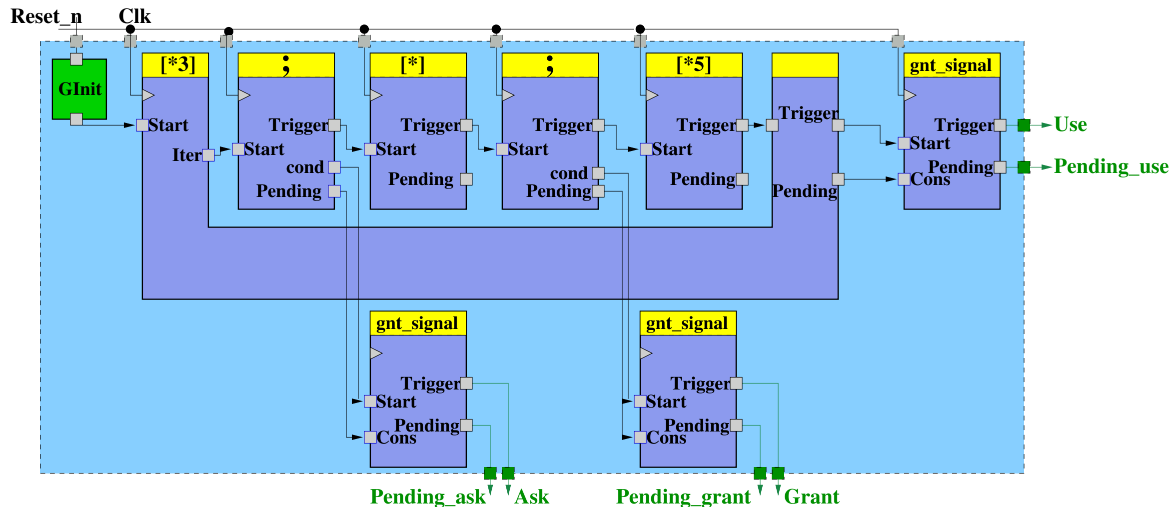

FIGURE 3.12 – Architecture du générateur pour l'hypothèse $H3_{arbitre}$

Le contrôle de la répétition de la séquence S est centralisé dans le générateur [*3] de $H3_{arbitre}$. C'est lui qui lance trois répétitions de la séquence S (3 activations du port *Iter*) et qui activera ensuite le reste de la propriété (activation du port *Trigger*), pour chaque *Start* provenant du Gen_Init.

Tout générateur de répétition possède un tableau à double entrées `Tokens` mémorisant pour chaque *Start* reçu : le numéro du jeton et le nombre de répétitions restantes pour ce jeton.

Alors que jusqu'à maintenant tous les jetons étaient codés par un simple bit à '1', nous utilisons des jetons codés sur N bits dans le cas des répétitions de séquences.

Chaque activation du générateur de répétition (port *Start* actif) déclenche la production d'un nouveau jeton. Celui-ci est placé dans `Tokens` et le nombre de répétitions restantes est initialisé. Ce jeton est envoyé sur le port *Iter* pour activer la première répétition de la séquence.

Chaque générateur composant la séquence effectue la génération et transmet le ou les jetons. Lorsqu'un jeton atteint la fin de la séquence, il est récupéré par le générateur de répétition. Celui-ci décrémente le nombre d'itérations restantes pour ce jeton. Si ce nombre vaut 0, la suite de la propriété est générée en passant le port *Trigger* du générateur de répétition à '1'.

Le tableau Tokens possède une taille fixe et il est donc nécessaire de pouvoir calculer statiquement le nombre de jetons maximal requis pour traiter la propriété.

Ce nombre peut être borné. Pour une séquence de longueur L, le nombre de jetons requis est limité à L. Dans le cas où la séquence est de taille variable[3], définir la borne est bien plus complexe [MAGB07].

Bien que ces considérations limitent fortement l'explosion de la complexité des générateurs pour l'utilisation de séquences SEREs, ceux-ci restent sensibles à la complexité des séquences répétées et au nombre de jetons utilisés.

Notre approche ne prend pas en compte les imbrications de répétitions de séquences. Il suffirait simplement d'augmenter les tailles utilisées dans le tableau Tokens pour coder les jetons et d'améliorer leur traitement dans les générateurs de répétitions de séquences im-

[3] C'est le cas lorsque la séquence contient l'opérateur [*], [->i], ou [=i]

briquées. Ceci augmenterait considérablement la complexité du circuit produit. Le rapport entre utilité et coût ne parait pas suffisamment bon pour implémenter cette technique. Il est en revanche possible d'utiliser des générateurs MyGen (*cf.* 3.4) prenant en compte les imbrications de séquences.

3.3.2.3 Traitement de l'opérateur de parallélisation de séquences : &&

Problème : L'opérateur SERE de parallélisation de séquence && est complexe puisqu'il requiert la génération de deux séquences en parallèle avec la contrainte supplémentaire que ces deux séquences soient de longueurs égales. Considérons une propriété S_{2amp} telle que S_{2amp}={S1&&S2} où S1 et S2 sont des séquences SERE. Trois cas peuvent se produire :

- S1 et S2 sont de même longueur fixe. Il suffit de construire un générateur pour S1, un pour S2 et de déclencher la génération de ces deux composants au même instant.
- S1 et S2 ont chacune une longueur fixe, mais lgr(S1)≠lgr(S2)[4]. Dans ce cas, aucune séquence respectant la sémantique de S_{2amp} ne peut être produite. Ce cas là peut être détecté statiquement en utilisant des outils tels que RAT [BCE+04].
- au moins une séquence possède une longueur variable. Dans ce cas, il est possible de générer des traces correctes à la volée, mais pour cela, une instrumentation doit être ajoutée au générateur global afin de garantir que les deux séquences produites possèdent la même longueur. Parfois aucune solution n'existe dans ce cas. C'est le cas pour la propriété SERE suivante : {a[*2] ;b[*]}&&{c}

L'idée sous-jacente va consister à analyser les deux séquences et à déterminer leurs longueurs minimales ainsi que le nombre d'opérateurs de répétition consécutives dans chacune de celles-ci. Des contraintes vont être extraites à partir de ces informations et le nombre de répétitions consécutives va être contraint pour chacun de ces opérateurs. Nous utiliserons une version modifiée du bloc aléatoire afin d'équilibrer les tailles des deux séquences pour produire des traces correctes.

La solution présentée ici se limite aux opérateurs de répétition consécutive : [*], [*i] et [+]. Traiter les opérateurs de répétitions SERE non consécutifs ([->i] et [=i]) est bien plus complexe l'équilibrage de la longueur des séquences doit tenir compte des nombres de cycles pouvant apparaître entre deux répétitions.

La ré-écriture des opérateurs de répétition non consécutive en opérateur de répétition consécutive peut être effectué de manière automatique en suivant les règles données dans [Bou08]. Cette manière de procéder semble bien plus efficace.

De plus, notre méthode s'applique si et seulement si les deux séquences à traiter contiennent le même nombre d'opérateurs de répétitions.

Formalisation : Le formalisme qui suit sert à extraire les données nécessaires à un équilibrage à la volée des longueurs de séquences, dans le cas où cela est possible. Soit HS1 la formule SERE suivante :

- Property S1 ={a ;b[*] ;c ;d[*] ;e ;f[*5]}
- Property S2 ={m[*] ;n ;o[*]}

[4]La fonction lgr(séquence S) retourne la longueur de la séquence S.

- Property HS1 : {S1 && S2}

Elle servira d'exemple dans la suite de cette section.

- min_i ($i \in [1..2]$) : Le nombre de cycles minimum pour la séquence S_i. Par exemple, pour la séquence S_1, min_1 vaut 8 : 1 cycle obligatoire pour a, c et e, et 5 cycles obligatoires pour f. Les signaux b et d peuvent ne pas être produit et leur nombre minimum de cycles est donc nul.
- var_i ($i \in [1..2]$) : Le nombre d'opérateurs de répétitions de longueur variable ([*], [+]). Pour la séquence S_1, var_1=2. Nous nous limitons aux séquences présentant le même nombre d'opérateurs de répétitions. Nous avons donc toujours var_1=var_2.
- $NR(j)_i, \forall j \in [1..var_i] \wedge \forall i \in [1..2]$: le nombre de répétitions de l'opérateur [*] en j-ème position dans la séquence S_i. Ces nombres constituent les inconnues du système d'équations 3.1. Leur ajustement sera utilisé pour équilibrer les longueurs des séquences.
- $Balance \in \mathbb{N}^+$: Nombre représentant la différence entre la longueur minimale de chaque séquence. On a donc : $Balance = min_1 - min_2$. Dans le cas de HS1, nous avons Balance=8-1=7.
- $Balance(k)_i$ ($i \in [1..2], k \in \aleph^+$) : Nombre entier composant $Balance$.
  On a ainsi : $Balance = \sum_{j=1}^{k}(Balance(j))$. Ceci sera utilisé pour répartir l'équilibrage des séquences sur plusieurs opérateurs de répétitions provenant de chaque séquences.

Nous obtenons les informations suivantes pour la propriété HS1.

- Séquence S1 : $min_1 = 8$ et $var_1 = 2$
- Séquence S2 : $min_2 = 1$ et $var_2 = 2$
- Balance=$min_1 - min_2$=7

Extractions de contraintes : Etant données les définitions ci-dessus, l'extraction de contraintes s'énonce de la façon suivante : La longueur des deux séquences doit être identique et égale à L. Ce qui se traduit par le système 3.1 :

$$\begin{cases} min_1 + \sum_{i=1}^{var_1}(NR(i)_1) = L \\ min_2 + \sum_{i=1}^{var_2}(NR(i)_2) = L \end{cases} \tag{3.1}$$

Le système d'équations 3.1 est produit à partir des données concernant les longueurs minimales et le nombre d'opérateurs [*] des séquences S_1 et S_2. C'est un système de deux équations linéaires à $var_1 + var_2 = N$ inconnues. Ceci donne le système suivant pour la propriété HS1 :

$$\begin{cases} 8 + NR(1)_1 + NR(2)_1 = L \\ 1 + NR(1)_2 + NR(2)_2 = L \end{cases} \tag{3.2}$$

Contenant ainsi N-1 degrés de liberté, la résolution passe par une détermination de la valeur de certaines variables jusqu'à ce que : soit le système possède une solution et un équilibrage est effectué, soit aucune solution n'est trouvée et la propriété ne peut être satisfaite. Le système 3.1 peut se ré-écrire de la manière suivante :

$$\sum_{i=1}^{var_1}(NR(i)_1) = \sum_{i=1}^{var_2}(NR(i)_2) - Balance = 0 \tag{3.3}$$

La méthode de résolution de tels systèmes passe par une détermination arbitraire de la valeur de certaines variables et une déduction de la valeur des autres variables de façon à obtenir un ensemble de solutions cohérent.

Ce qui est intéressant est l'équilibrage de la longueur des deux séquences. Pour cela, deux solutions sont possibles : équilibrer un couple et fixer toutes les autres répétitions à 1, ou alors répartir l'équilibrage sur plusieurs couples.

L'équation 3.4 montre comment l'équilibrage s'effectue sur un seul couple d'opérateurs de répétitions. L'idée est alors d'effectuer l'équilibrage sur les opérateurs de répétitions en j-ème position. Tous les opérateurs de répétitions de position différente seront remplacés par une répétition fixée à une occurrence.

$$\begin{cases} NR(j)_2 = Balance + 1, j \in [1..min(var_1, var_2)] \wedge NR(j)_1 = 1 \\ \forall i \neq j, NR(i)_1 = NR(i)_2 = 1 \end{cases} \tag{3.4}$$

En appliquant cette technique d'équilibrage 3.4, l'équation suivante est obtenue pour HS1 :

$$NR(1)_1 = NR(1)_2 + 7 \text{ et } NR(2)_1 = NR(2)_2 = 1$$

De cette façon, L=7+rand_nb (Balance=7). L'entier rand_nb est produit par le bloc aléatoire. La séquence HS1 s'écrit alors :

Property HS1 : {a ;b[*rand_nb] ;c ;d[*1] ;e ;f[*5]} && {m[*rand_nb+7] ;n,o[*1]}

Toujours dans le cas où il y a le même nombre d'opérateurs de répétitions dans chaque séquence (var_1=var_2), la seconde solution consiste à répartir l'équilibrage sur plusieurs couples d'opérateurs de répétition de longueur variable 3.5. Pour cela l'entier Balance est décomposé selon la somme de k entiers nommés Balance(k). Les k couples d'opérateurs [*] sont alors équilibrés selon les nombres Balance(k) correspondants. L'équation 3.5 donne le système obtenu :

$$\begin{cases} \forall j \in [1..k], k \leq var_1, NR(j)_1 = 1, NR(j)_2 = Balance(k) + 1 \\ \forall i \in [k..var_1], NR(i)_1 = NR(i)_2 = 1 \end{cases} \tag{3.5}$$

En appliquant la méthode d'équilibrage 3.5, le système suivant est obtenu pour HS1 :

$$\begin{cases} NR(1)_1 = NR(1)_2 - Balance(1), NR(1)_1 = 1 \\ NR(2)_1 = NR(2)_2 - Balance(2), NR(2)_1 = 1 \\ Balance(1) + Balance(2) = 7 \end{cases} \tag{3.6}$$

Prenons par exemple Balance(1)=4 et Balance(2)=3 ; Ceci donne alors les valeurs suivantes : $NR(2)_1 = 4$ et $NR(2)_2 = 3$. L'hypothèse HS1 s'écrit alors :

Property HS1 : {a ;b[*rand_nb$_1$] ;c ;d[*rand_nb$_2$] ;e ;f[*5]} &&
{m[*rand_nb$_1$+4] ;n ;o[*rand_nb$_2$+3]}

Si aucune solution n'est trouvée, alors il est impossible de satisfaire la propriété qui est dite "incohérente".

Cette solution ne traite pas le cas général où des répétitions de séquences sont imbriquées, mais seulement le cas où toute imbrication de séquence est composée de séquences de longueurs fixes. Prenons la séquence HS2 suivante :

Property HS2 : {{a ;b[*]}&&{{c ;d[*2]}[*] ;e}}

En appliquant l'équilibrage 3.4, la résolution s'effectue comme suit :

$$\left.\begin{array}{l} 1 + NR(1)_1 = L \\ (1+2) * NR(1)_2 = L \end{array}\right\} \Leftrightarrow NR(1)_1 = 3 * NR(1)_2 \quad (3.7)$$

L'extension de cette méthode pour des répétitions imbriquées de tailles variables est un travail futur.

Cas particuliers :

- Cas où $var_1 = 0$ et $var_2 = 0$: Ce cas se présente si la longueur de chaque séquence S1 et S2 est fixe. Soit les deux séquences ont la même longueur et alors aucun équilibrage, ni aucun traitement particulier n'est à effectuer ; soit leurs longueurs diffèrent et la formule ne possède aucune trace satisfaisant celle-ci. Dans ce dernier cas, le problème est détecté à la compilation et l'utilisateur est informé que la formule passée en entrée ne peut produire un générateur correct puisqu'elle présente une incohérence.
- Cas où $var_1 = var_2$: Il y a autant d'opérateurs [*] dans chacune des séquences ! Dans ce cas, les générateurs sont reliés par couples et il ne reste aucun générateur [*] seul. La méthode décrite dans le cas général peut être appliqué.
- Cas où $var_1 = 0$ et $var_2 > 0$: Une seule séquence possède au moins un opérateur [*]. Supposons que ce soit la séquence S1 qui ait une longueur fixe L. Alors le système 3.1 devient :

  $$\sum_{i=1}^{var_1} (NR(i)_2) = L - min_2 \quad (3.8)$$

  Il suffit de fixer les valeurs des $NR(i)_2$ de façon à satisfaire l'équation 3.8.

Mise en commun de LFSR : Comme il vient d'être présenté dans la partie précédente, l'équilibrage de la longueur des séquences s'effectue sur des couples d'opérateurs de répétition ($NR(i)_1$,$NR(i)_2$) nommés respectivement maître et esclave. L'opérateur $NR(i)_1$ contraint la valeur qui sera produite pour $NR(i)_2$. Prenons la formule {S1 && S2} avec S1={a ;b[*]} et S2={[*] ;c ;d}. La première répétition de taille non bornée, qui est activée, est appelée étoile maître car aucune contrainte n'existe sur le nombre de répétitions. La seconde quant à elle doit respecter ces contraintes et s'appelle répétition esclave.

Ainsi, si l'équilibrage vaut E, lorsque le LFSR produit un entier K pour $NR(i)_1$, alors il devra produire K+E pour l'opérateur esclave $NR(i)_2$. Le LFSR a été modifié en conséquence. Il possède deux ports de sortie $rand_num_1$ et $rand_num_2$ pour déterminer le nombre de répétition pour les opérateurs maître et esclave respectifs. Un simple additionneur a été ajouté afin de prendre en compte l'équilibrage sur le port $rand_num_2$.

Dans ce contexte, le LFSR n'est plus embarqué au sein de chaque générateur primitif, mais mis en commun pour les couples de générateurs de répétition concernés. Cela simplifie l'architecture des générateurs primitifs.

3.3.2.4 Bilan

Le tableau 3.3 regroupe les résultats de synthèse pour les générateurs primitifs SERE. La première partie concerne les générateurs simples, dans le cas où les propriétés ne contiennent ni répétitions de séquences, ni opérateur &&. Leur complexité est faible, ce qui en fait des composants efficaces, dont les caractéristiques sont identiques aux générateurs FL.

La seconde partie contient les générateurs de répétition contrôlant les répétitions de séquences. Leur taille est beaucoup plus importante. Tous les générateurs peuvent prendre en compte $2^{32} - 1$ jetons différents. Ce nombre est conséquent, et en réalité, le nombre de jetons nécessaire est très largement inférieur. Néanmoins, cela donne une bonne idée des limites de la mise en œuvre de la méthode, spécialement dans le cas d'un prototypage sur FPGA.

En comparant les résultats obtenus pour [*] et [*6], il est clair que le nombre de répétition a un impact important sur la complexité du circuit. Alors que [*6] gère 6 répétitions maximum, [*] peut produire un nombre de répétitions compris entre 0 et $2^{32} - 1$.

Concernant les générateurs à l'intérieur de séquences répétées, leur complexité est elle aussi plus importante, bien que d'ordre inférieur aux générateurs précédents. En comparant encore [*], [*6], il est possible d'apercevoir que le nombre de répétitions est encore plus significatif dans ce cas.

Opérateur	**LCs**	**FFs**	**Fréquences**
:	0	0	**420.17**
;	1	1	**420.17**
[*6]	8	6	**420.17**
[*]	69	40	283.45
[=6]	35	24	**420.17**
[->6]	31	25	397.30
contrôleurs de répétitions de séquences			
[*]	1816	1202	82,49
[*6]	640	355	120,7
[=6]	-	-	-
[->6]	-	-	-
générateurs à l'intérieur de répétitions de séquences			
:	0	0	**420.17**
;	45	33	**420.17**
[*6]	214	192	**420.17**
[*]	1305	1039	**228.24**
[=6]	-	-	-
[->6]	-	-	-

Table 3.3 – Résultats en synthèse pour les générateurs SERE

Les générateurs primitifs de répétition non consécutive n'ont pas été implémentés par manque de temps. Toutefois, ceux-ci sont moins fréquents dans les propriétés et des règles

de ré-écriture peuvent permettre de s'affranchir de leur utilisation directe. Des travaux futurs devront définir ceux-ci afin de compléter l'approche.

3.3.3 Générateurs et négation de propriétés

L'utilisation des opérateurs Not et Implication oblige nos générateurs à pouvoir produire des signaux conformes à la négation de la sémantique de l'opérateur correspondant. Prenons par exemple la formule `A -> B` avec A et B des booléens. Alors trois cas conformes à la sémantique se présentent et doivent pouvoir être générés :
- A faux et B faux.
- A faux et B vrai.
- A vrai et B vrai.

Dans les deux premiers cas, il est nécessaire que le signal A soit Faux. Etendons cet exemple à des formules booléennes pour A et B. Alors dans les deux premiers cas, il est nécessaire de générer des signaux respectant la négation de l'expression booléenne A. Une information supplémentaire doit donc être propagée dans l'arbre des générateurs primitifs afin que chaque générateur puisse savoir s'il doit produire des signaux conformes à la sémantique ou à la négation de l'opérateur qui lui est associé.

Dans le sous-ensemble simple PSL_{ss}, la négation n'est autorisée que sur les expressions booléennes, c'est pourquoi nous proposons ici une version différente des générateurs primitifs pour les opérateurs booléens uniquement. Le générateur gnt_Signal est considéré comme un booléen réduit à un simple signal. Lui aussi peut avoir besoin de produire des signaux contraints à '0' ou '1'. Ils possèdent donc une architecture identique aux générateurs booléens.

Pour pouvoir surmonter cette difficulté, deux modifications sont apportées aux générateurs primitifs booléens :

- un nouveau bloc NOT_SEM est ajouté. Il permet de produire des signaux conformes à la négation de l'opérateur booléen concerné.
- L'interface des générateurs a été surchargée avec l'ajout du port d'entrée *Cons* sur tous les générateurs booléens.

La figure 3.13 illustre l'architecture des générateurs primitifs booléens.

Chaque couple de port (*Trigger*, *Pending*) est connecté au couple (*Start*, *Cons*). Dans la suite, nous utiliserons uniquement le premier couple pour exposer le traitement des négations.

Le couple de ports *Trigger* et *Pending* de chaque générateur primitif permet de gérer la négation au niveau de l'opérateur courant, mais permet aussi de propager l'information de négation. Le tableau 3.4 décrit la signification des ports *Trigger* et *Pending* pour les générateurs booléens pouvant produire la négation de l'opérateur associé.

Trigger	*Pending*	**Génération**
0/1	0	aléatoire (bloc ALEA)
0	1	conforme à la négation de la sémantique (bloc NOT_SEM)
1	1	conforme à la sémantique (bloc SEM)

Table 3.4 – Signification du couple de ports (*Trigger*, *Pending*)

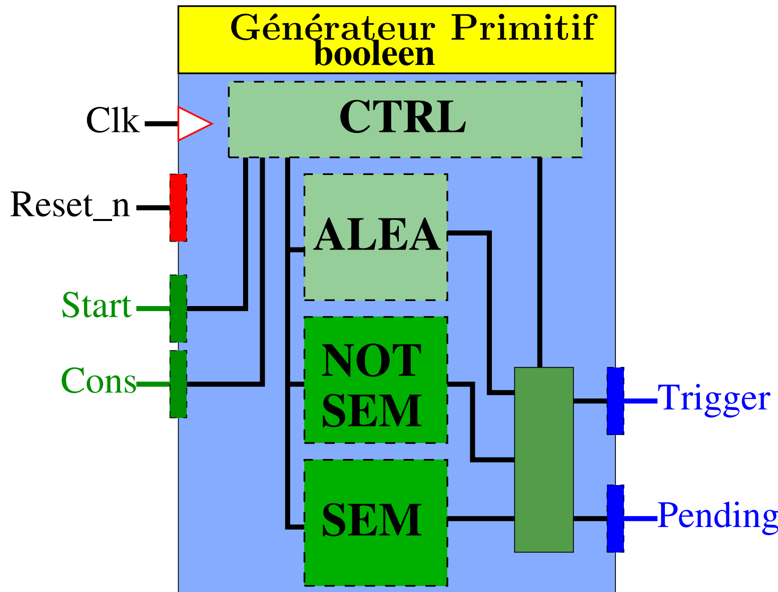

Figure 3.13 – Nouvelle architecture pour les générateurs primitifs booléens

Le chronogramme de la figure 3.14 illustre le fonctionnement des ports (*Trigger*, *Pending*) sur le générateur until. Supposons que celui-ci soit activé au cycle ♯1. La partie aléatoire décide de générer le signal B 7 cycles plus tard. Jusqu'à ce cycle, deux évènements se produisent.

Le signal A est contraint à '1' en passant les ports *Trigger* et *Pending* à '1'. Le générateur gnt_Signal reçoit ('1','1') sur ses ports (*Start*, *Cons*) et active donc le port externe correspondant au signal A.

Mais le signal B ne doit pas être produit avant le cycle ♯8. Jusqu'à ce cycle, il est contraint à '0' en passant le port *Cond* à '0' et *Pending* à '1'. Le générateur gnt_Signal reçoit ('0','1') sur ses ports (*Start*, *Cons*) et fixe à '0' le port externe du signal B.

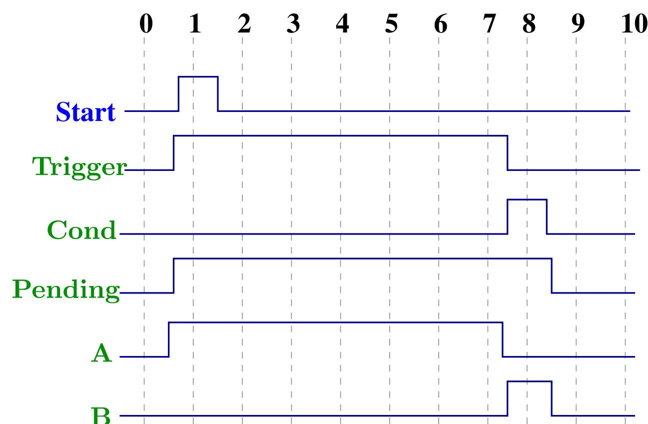

Figure 3.14 – Exemple pour le générateur until

Les circuits booléens sont donc légèrement plus complexes lorsque des négations sont utilisées dans les propriétés. Une solution plus simple consisterait à ré-écrire les expressions

booléennes pour faire descendre les négations aux niveau des signaux et ainsi supprimer le problème de négation.

3.4 Générateurs méthode 2 : MYGEN

Cette section présente les travaux menés dans l'équipe du professeur Zeljko Zilic à l'université de McGill[5]. Marc Boulé a développé, au sein de cette équipe, une méthode et un logiciel de synthèse d'assertions en moniteurs appelé MBAC. Cet outil produit des moniteurs très efficaces. Ils utilisent une surface minimale et possèdent de très bonnes fréquences.

Les résultats sur les moniteurs étant très encourageants, l'idée de base sous-tendant les travaux de cette section était : peut-on adapter l'outil MBAC pour la synthèse d'hypothèses en générateurs ? Une solution a été trouvée et un outil appelé MyGen a été développé pour automatiser le processus de synthèse.

3.4.1 Concepts préliminaires

3.4.1.1 Synthèse d'assertions à l'aide de MBAC

L'outil MBAC décrit dans [BZ06] prend en entrée une propriété temporelle et produit le moniteur correspondant. Ceci est réalisé en construisant l'automate reconnaissant toutes les traces qui violent la propriété. Une description HDL synthétisable est alors extraite de cet automate.

L'ensemble des opérateurs primitifs de PSL est divisé en deux sous-ensembles. Pour le premier, contenant essentiellement les opérateurs booléens, chaque opérateur possède un automate correspondant qui a été prédéfini. La synthèse du second sous-ensemble est effectuée en deux étapes : des règles de ré-écriture sont utilisées pour se ramener à des opérateurs simples du premier sous-ensemble, puis les automates élémentaires sont alors combinés entre eux pour obtenir la propriété finale.

Le principe est le même pour des propriétés complexes. Toutes les règles de ré-écriture ont été prouvées correctes vis-à-vis de la sémantique PSL à l'aide du prouveur de théorème PVS [MABBZ08]. Des optimisations sont effectuées tout au long de l'élaboration de l'automate afin de minimiser la complexité du moniteur final.

La partie droite de la figure 3.15 illustre l'automate produit par MBAC pour la propriété **A2b(i)** (définie section 2.1.1.2) :

$$\textsf{property A2b(i) is assert always}(\{ask(i)\,;\textsf{not}\ grant(i)\}|\!\rightarrow\!\{\text{true}\,;\textsf{not}\ use(i)\})\,;$$

L'automate décrivant le moniteur peut posséder plusieurs états initiaux et plusieurs états finaux. Quand un état est actif, on dit qu'il possède un "jeton". Les automates peuvent posséder plusieurs jetons simultanément ; d'une part à cause de la ré-entrance qui peut activer la propriété plusieurs fois, mais aussi parfois à cause de la structure de l'automate qui peut dupliquer les jetons.

Les transitions sont étiquetées par des conditions observées. Ce sont des expressions booléennes composées de signaux appartenant à la propriété correspondante. Si la condition est satisfaite, la transition est prise : le jeton est passé de l'état courant à l'état

[5] McGill University, Department of Electrical & Computer Engineering McConnell Engineering Building, Montréal, Québec

suivant. Si aucune condition de transition n'est valide, l'état perd son jeton, et aucun état n'est réactivé. La propriété est violée chaque fois qu'au moins un état final possède un jeton.

Le point fondamental de la méthode consiste à traiter différemment les parties gauche et droite d'une implication. Alors qu'il suffit de détecter chaque instant où une partie gauche est valide, il suffit de détecter chaque instant où une partie droite est violée. On distingue alors deux modes d'interprétation d'une propriété :

- **condition** : reconnaissance d'une séquence, ou expression booléenne vraie.
- **obligation** : violation d'une séquence, ou expression booléenne fausse.

Le traitement du mode obligation s'obtient à partir du mode condition en appliquant un algorithme appelé First_Fail. Celui-ci simplifie la partie droite de l'implication en ne gardant que les chemins menant à une violation de la propriété. Cet algorithme est une sorte de négation appliquée sur une partie de l'automate.

La figure 3.15 illustre ceci. La propriété est violée si une unité demande un accès à la ressource à un cycle t, ne l'obtient pas au cycle suivant et l'utilise directement après malgré le refus d'obtention. Dans le cas général, l'application du First_Fail peut modifier la structure de l'automate.

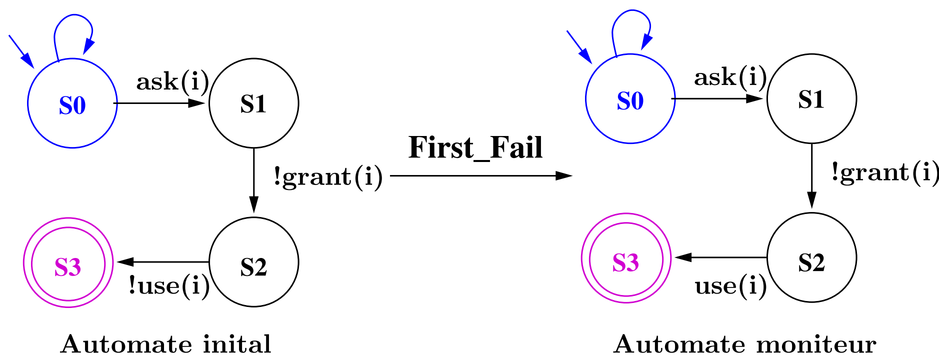

FIGURE 3.15 – Application de l'algorithme First_Fail pour le moniteur A2b(i)

3.4.1.2 Des moniteurs vers les générateurs

Générateurs et moniteurs sont deux concepts symétriques : un moniteur reconnaît le langage d'une propriété (ou plutôt son complément) alors qu'un générateur produit le langage d'une propriété. Cette symétrie se retrouve au niveau des automates, comme le montre le tableau 3.5. Il est alors naturel d'utiliser la même technique de construction d'automates pour synthétiser moniteurs et générateurs.

Puisque les moniteurs produits par MBAC sont très efficaces, nous avons décidé de réutiliser l'approche MBAC et le mécanisme de construction d'automates pour produire des générateurs synthétisables.

3.4.2 Synthèse du générateur

3.4.2.1 Description globale

Comme le montre la figure 3.16, la production de générateurs s'effectue en deux principales étapes. Tout d'abord l'outil MBAC produit l'automate décrivant le moniteur cor-

	Moniteurs	**Générateurs**
conditions de transitions	observées	générées
état finaux	violation	satisfaction sans vacuité

Table 3.5 – Symétrie entre moniteurs et générateurs

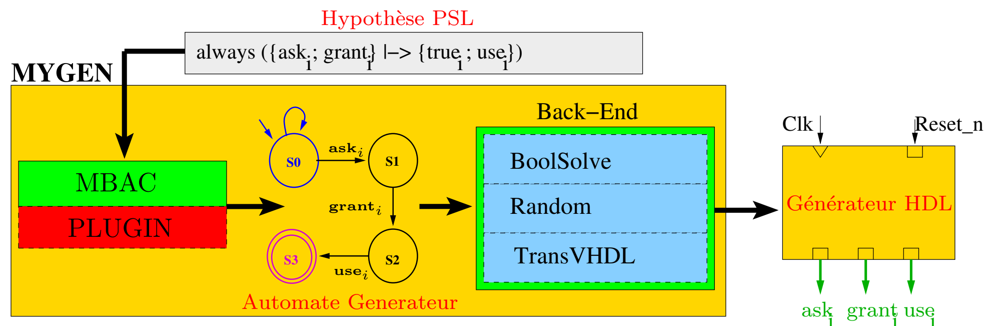

FIGURE 3.16 – Synthèse de générateur à l'aide de MyGen

respondant à la propriété de départ. Puis celui-ci est transformé en un automate décrivant toutes les traces conformes à la propriété de départ. Finalement l'outil back-end extrait le VHDL synthétisable décrivant le générateur.

3.4.2.2 MBAC$_{plugin}$

L'idée consiste à supprimer l'application de l'algorithme `First_Fail`. L'automate résultant décrit alors toute les traces satisfaisants la propriété correspondante. Ceci peut-être observé figure 3.15. En fait, l'algorithme `First_Fail` agit comme une négation sur une partie de l'automate. Cet algorithme donne l'ensemble des traces ne satisfaisant pas la propriété. Supprimer son application permet d'obtenir l'ensemble des traces satisfaisant la propriété.

3.4.3 Le Générateur

3.4.3.1 Description de l'automate du générateur

Tout générateur possède une interface générique prenant en entrée les signaux *clk* et *reset* pour se synchroniser avec le DUT ; et en sortie les signaux impliqués dans la propriété.

La description matérielle du générateur est composée de deux blocs : un pour le codage de l'automate et un pour le bloc aléatoire. Comme plusieurs états de l'automate peuvent être actifs à un instant donné, le codage des états est fait en 1 parmi N. Le processus HDL décrivant l'automate du générateur possède toujours la même structure :

1. sélection des états actifs (ligne 1 du code source figure 3.18)
2. sélection de la transition (lignes 2,4 et 8)
3. sélection d'une valeur pour les signaux de la condition de transition courante (ligne 9, 14, 18)

Dans chaque branche finale du code source (lignes 3,10, 15 et 19), deux types d'opérations sont effectuées : affectation des signaux de sortie et activation/désactivation des états concernés.

3.4.3.2 Bloc aléatoire

Plusieurs, voire une infinité de traces peuvent satisfaire la même propriété. Ceci se traduit au niveau automate par le fait que plusieurs chemins mènent à l'état final. Ainsi il existe des états où un choix doit être fait parmi plusieurs transitions. Ce choix est effectué de manière pseudo-aléatoire à l'aide d'un bloc produisant des nombres pseudo-aléatoires.

Le bloc pseudo-aléatoire n'est pas seulement utilisé pour sélectionner une transition. Etant donné une transition, sa condition associée peut être une expression booléenne complexe contenant plusieurs signaux à générer. Plus d'une valuation pour ces signaux peut satisfaire la condition. Le choix parmi ces valuations est effectué par le bloc pseudo-aléatoire.

Deux types de bloc pseudo-aléatoire ont été testés : Linear Feedback Shift Registers (LFSR) et Automate Cellulaire (CA). Une description détaillée de ces blocs est donnée section 3.5.

Le bloc aléatoire peut être paramétré pour modifier la qualité des traces produites par le générateur. Il est possible d'obtenir différentes instances d'un générateur dans chacun de ces modes. Cela permet de mieux contrôler le type de vecteurs de test qui vont être produits :

- **SimpleRand** : minimise la complexité du générateur produit en partageant un composant aléatoire pour plusieurs tâches. La qualité des traces est faible à cause des dépendances induites par le partage d'un même bloc aléatoire. Cela peut être utile pour un prototypage rapide sur des plateformes simples.
- **DirRand** : permet de sélectionner le type de traces à produire : la plus courte, moyenne, longue etc. Cela peut être utile pour un test en profondeur du circuit.
- **RealRand** : ce mode utilise un bloc aléatoire distinct pour chaque choix effectué à l'intérieur de l'automate. La qualité des vecteurs est maximale et le générateur peut couvrir tout l'ensemble de traces satisfaisant la propriété correspondante. Ce mode est particulièrement adapté pour une vérification complète n'excluant aucun cas limite. La taille du générateur est en revanche plus conséquente.

La flexibilité fournie par ces différents modes est très utile puisqu'il est possible de construire diverses instances d'un même générateur, chacune dédiée à une étape de vérification différente.

Dans la suite, nous appellerons "registre aléatoire" un registre contenant des nombres produits de manière pseudo-aléatoire. Pour la sélection de transitions, les registres aléatoires sont appelés `Trans`. L'algorithme figure 3.18 illustre ceci lignes 2, 4 et 8.

3.4.3.3 L'outil BoolSolve

Pour chaque transition, le générateur doit produire une combinaison de signaux validant la condition de transition. Dans certains cas, il est possible que plusieurs valuations satisfassent la condition. Pour énumérer toutes ces valuations, un solveur booléen est utilisé.

La figure 3.17 illustre ceci. La transitions Trans(2) peut être prise en fixant différentes valeurs (1,0),(0,1) ou (1,1) pour le couple de signaux (A,B). Toute ces solutions sont

statiquement calculées lors de l'analyse de l'automate générateur. Elles sont ensuite codées directement dans la description matérielle.

Notre approche utilise BoolSolve, un outil gratuit disponible sur internet [boo]. Il prend en entrée l'expression booléenne et fournit en sortie toutes les valuations correctes pour cette expression. Le choix d'une valuation est effectuée pseudo-aléatoirement à l'aide de registres aléatoires : `Rand`. L'algorithme figure 3.18 montre ceci lignes 9 et 14. Les signaux non impliqués dans la transition courante ont une valeur pseudo-aléatoire.

3.4.3.4 Gestion des transitions multiples

La dernière subtilité concerne la relation de dépendance que possèdent les transitions sortantes d'un même état. Prenons l'automate figure 3.17 et imaginons que la transition `Trans(2)` soit sélectionnée. Alors deux cas de figures se produisent selon la valeur donnée aux signaux A et B :

- (0,1) : dans ce cas, seule la transition Trans(2) est active et aucune relation de dépendance n'entre en jeu. Au cycle suivant, seul S2 sera actif.
- (1,0) : ici, la transition `Trans(0)` est implicitement activée car le fait d'assigner A à '1' pour `Trans(2)` valide la condition de la transition `Trans(0)`. Ainsi au cycle suivant, les états S2 et S1 seront actifs.

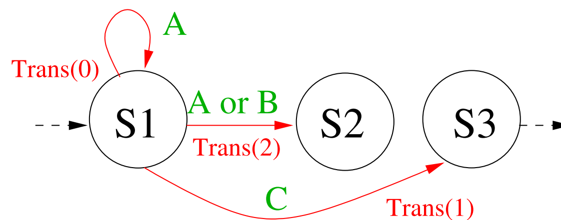

FIGURE 3.17 – Partie d'automate d'un générateur

Ces dépendances sont analysées statiquement par l'outil MyGen avant la production de la description VHDL.

3.4.4 Résultats expérimentaux

3.4.4.1 Analyse de l'outil MYGEN

La complexité des propriétés de l'ensemble LIM est d'un ordre de grandeur supérieur aux autres groupes. Les résultats expérimentaux du groupe LIM sont donnés sur un graphe à part pour obtenir des graphes plus clairs.

Temps d'exécution : Comme le montre la figure 3.19, la plupart des générateurs sont construits en quelques secondes. La majorité du temps utilisé pour produire les générateurs est consommé par BoolSolve pour calculer et lister toutes les valuations correctes pour chaque condition de transition.

Ainsi, plus il y a de transitions, plus le temps de construction augmente. Considérons par exemple la propriété LIM_{40} suivante (appartenant au groupe LIM) :

$$\textbf{Property LIM}_{40} : \mathsf{always}(\mathit{sig1} \rightarrow \mathsf{next_event}(\mathit{sig2})[1:512](\mathit{sig3}))$$

```
1. if S1=1 --Current State=S1
2.     if Trans(0)=1 --Cond=A
3.         A<=1;
4.     elsif Trans(1)=1 --Cond=C
5.         C<=1;
6.         S1<=0;
7.         S3<=1;
8.     elsif Trans(2)=1 --Cond=A or B
9.         if Rand(0)=1 and Rand(1)=1
10.            A<=0;
11.            B<=1;
12.            S1<=0;
13.            S2<=1;
14.        elsif Rand(0)=0 and Rand(1)=1
15.            A<=1;
16.            B<=0;
17.            S2<=1;
18.        else
19.            A<=1;
20.            B<=1;
21.            S2<=1;
22.        end if;
23.    end if;
24.end if;
```

FIGURE 3.18 – Code HDL pour l'automate décrit figure 3.17

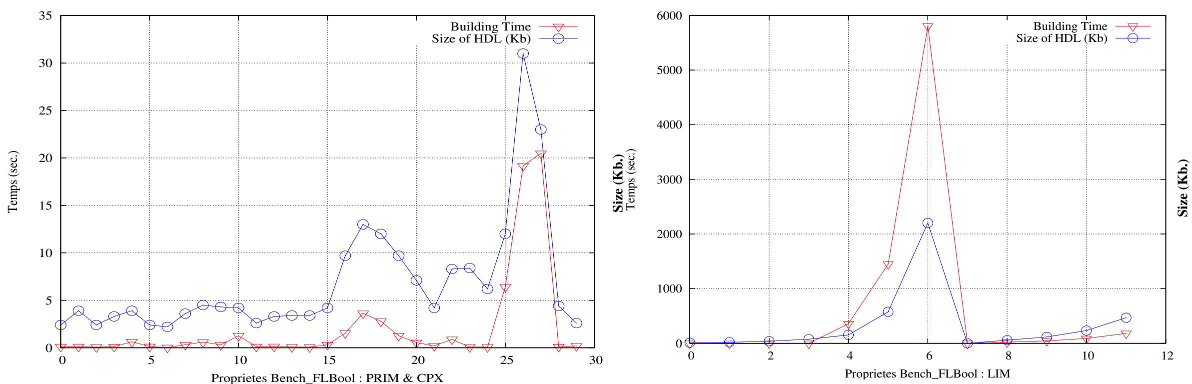

FIGURE 3.19 – MyGen statistiques : temps d'exécution et taille du code HDL des générateurs

L'automate du générateur possède 510 transitions et plusieurs minutes sont requises pour extraire la description matérielle correspondante.

Complexité des générateurs : Alors que le temps de construction est lié au nombre de transitions contenu dans l'automate, la taille de la description HDL dépend essentiellement de la complexité des conditions de transition. Le nombre de valuations correctes pour une condition de transition peut croître très rapidement. Comme la liste de toutes ces valuations est codée directement dans la description HDL, la taille de celle-ci peut aussi exploser. Prenons par exemple la propriété CPX_{29} suivante (appartenant au groupe CPX) :

Property CPX_{29} : always(sig1 or sig2 or sig3 or sig4 or sig5 or sig6 or sig7 or sig8) ;

L'automate pour cette propriété est très simple et contient seulement 2 transitions. Une d'entre elle est étiquetée avec l'expression booléenne contenue dans CPX_{29}. Or pour celle-ci, $2^8 - 1$ valuations correctes existent. La description HDL résultante possède alors 4000 lignes de code.

Ces résultats montrent aussi qu'il est difficile d'établir un lien clair entre la complexité de la propriété et de la description HDL finale. Ceci pour deux raisons. D'une part les traitements faits sur l'automate durant la création du générateur modifient la complexité de la description finale, et l'utilisation de blocs aléatoires influence grandement la complexité du circuit final.

3.4.4.2 Résultats de synthèse sur FPGA

La même plateforme FPGA que pour les expérimentations précédentes a été utilisée. Les générateurs ont été construits en utilisant le mode RealRand de l'outil MyGen.

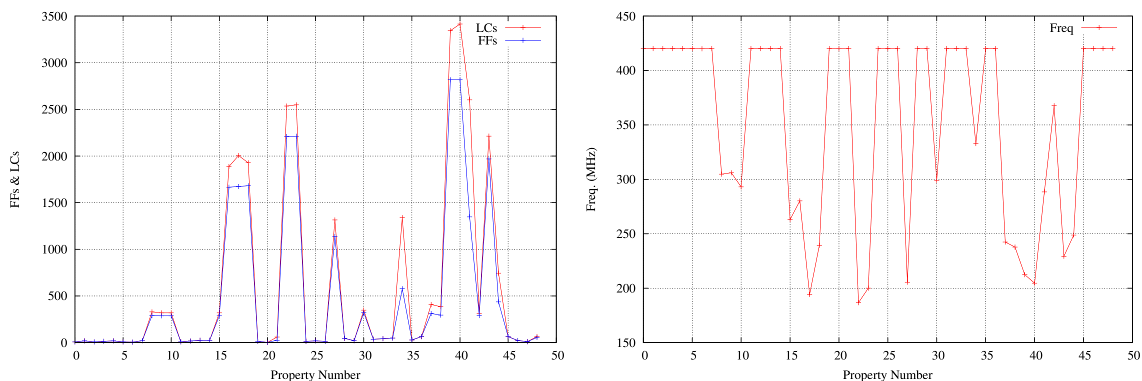

FIGURE 3.20 – Résultats de synthèse : Surface (gauche) et fréquences (droit)

Comme le montrent les courbes figure 3.20, la complexité des générateurs varie de manière significative. Les figures 3.19 et 3.20 permettent de voir que le temps de construction, la taille de la description HDL et la surface utilisée par le générateur, sont liés.

De plus, la surface des générateurs peut aussi croître de manière significative, tout particulièrement pour les propriétés du groupe LIM. Par exemple, le circuit pour la propriété LIM_{40} (dont le VHDL représente 16933 lignes) utilise 122K cellules logiques et 70k registres.

Enfin, il est possible de voir sur la courbe de gauche de la figure 3.20 que la quantité de cellules logiques et de registres est quasiment toujours égal. Ceci permet au circuit d'avoir des chemins critiques très courts, ce qui se traduit par des fréquences toujours très élevées.

La fréquence mimale obtenue durant ces expérimentations se situe autour de 200 MHz, même pour des générateurs complexes. Il est à noter que ce phénomène semble hérité des moniteurs qui possèdent exactement la même caractéristique en fréquence. La plupart des générateurs possèdent la fréquence maximale, imposée par le FPGA Cyclone II, de 420 MHz.

3.4.4.3 Comparaisons HORUS/MYGEN

L'utilisation de l'aléatoire est moins efficace pour l'approche MyGen. Ainsi, comme le montrent les courbes figure 3.21, le nombre de cellules logiques et de registre est globalement plus important pour les générateurs MyGen.

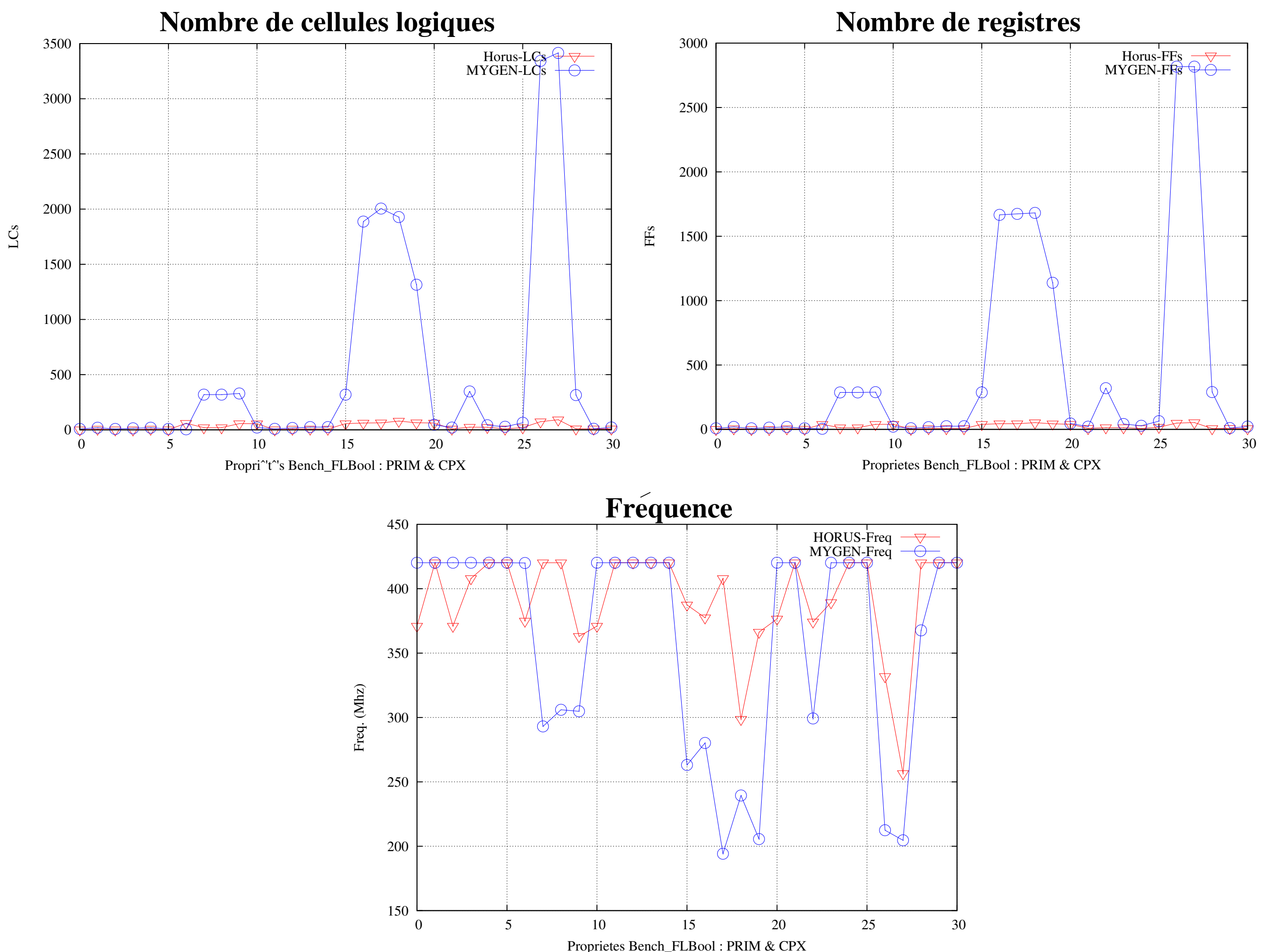

FIGURE 3.21 – Comparaison des résultats de synthèse pour Horus et MyGen

Par contre, on retrouve toujours cet équilibre entre nombre de registres et de cellules

logiques, ce qui fournit plus souvent des fréquences maximales pour MyGen. Même si Horus possède quelques fréquences plus élevées, l'écart n'est pas significatif.

Une remarque concerne les SEREs. L'approche MyGen semble meilleure dans ces cas là puisque la construction est directe et simple. Elle ne requiert pas d'instrumentation particulière, et même si les circuits obtenus pour des répétitions de séquences sont complexes, la méthode les prend en compte sans aucun problème. L'opérateur de parallélisation de séquences && est lui aussi traité sans aucun problème.

Une comparaison sur les outils Horus et MyGen est fournie section 7.1.

3.5 Bloc aléatoire embarqué

Diverses expérimentations ont été conduites afin d'offrir un bloc aléatoire le plus efficace possible. Deux types de composants ont été testés : les LFSRs et les automates cellulaires (CAs). Cette partie donne les détails quant au fonctionnement de chacun, ainsi qu'une comparaison de ceux-ci.

3.5.1 Le composant LFSR

Le document [Ins96] introduit de manière simple le concept de LFSR (Linear Feedback Shift Register). Ce composant permet de générer très simplement des séquences pseudo-aléatoires (pour une graine donnée, les séquences produites seront toujours identiques, d'où le caractère **pseudo**-aléatoire).

Ce composant est un registre à décalage passant d'un état à un autre par une fonction linéaire qui se réduit à XOR ou XNOR dans le cas des bits. Deux grandes classes de LFSR se distinguent : Fibonacci et Galois. Dans le premier cas, les portes XOR (ou XNOR) sont placées en parallèle du registre à décalage à des positions déterminées appelées *Taps*. Un LFSR contenant N Taps formés de portes P aux positions (i_1, ..., i_N) sera noté $LFSR_P[i_1,...,i_N]$.

Dans le cas des LFSR de Galois, les portes sont intercalées directement entre les bits d'états du LFSR. La figure illustre un LFSR de Fibonacci à 3 bits, pouvant donc produire des nombres pseudo-aléatoires entre 1 et 7. Ce LFSR ne contient qu'un seul Tap numéroté [2] qui est une porte XOR. Il sera noté $LFSR_{xor}[2]$.

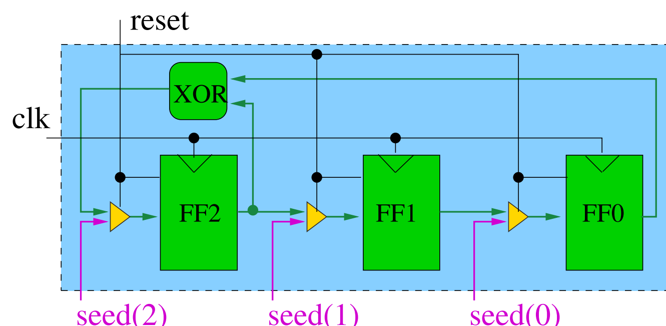

FIGURE 3.22 – $LFSR_{xor}[2]$ 3 bits de Fibonacci

Le LFSR (Fibonacci, ou Galois) est en fait un simple compteur dégénéré produisant les nombres dans un ordre inhabituel. La taille du LFSR définit l'intervalle sur lequel les

entiers seront produits et impose aussi une limite à la périodicité de la production. Un LFSR contenant en effet 3 bits, peut produire 7 nombres différents[6].

Après 7 générations, la graine est à nouveau rencontrée et la séquence suivante est alors identiques aux 7 nombres précédents. Le tableau 3.23 illustre le phénomène sur le $\text{LFSR}_{xor}[2]$ 3-bits ayant pour graine l'entier 5.

t=	0	1	2	3	4	5	6	7	8	9
bits	**101**	010	001	100	110	111	011	**101**	010	001
entier	**5**	2	1	4	6	7	3	**5**	2	1

FIGURE 3.23 – Séquence produite par un $\text{LFSR}_{xor}[2]$ 3-bits

Les Taps d'un LFSR peuvent être représentés sous forme polynomiale. Cette représentation est appelée polynôme caractéristique du LFSR. Par exemple, si les Taps sont [16,14,11], le polynôme caractéristique correspondant est :

$$\text{P}[16,14,11]=x^{16}+x^{14}+x^{11}+1.$$

Le monôme "1" correspond à l'entrée du LFSR. Le document [WM07] fourni une table établissant une correspondance entre les polynômes caractéristiques et la longueur maximale de séquence (ou période) qu'il est possible d'atteindre pour toute une batterie de LFSRs. Néanmoins, quelques règles permettent de caractériser un LFSR :

- La période maximale sera atteinte si et seulement si le nombre de Taps est pair. Deux ou quatre Taps seulement suffisent pour atteindre de très longues périodes.
- L'ensemble des Taps doit être premier et ne pas partager de diviseurs commun.
- Pour une taille de LFSR donnée, il peut exister plus d'une période maximum.

Il existe d'autres types de LFSRs dit "non linéaires" offrants de meilleures qualités aléatoires : période plus longue, fréquence plus élevée etc. Néanmoins, ceux-ci ne sont pas encore complètement caractérisés et leur utilisation est donc plus délicate [DTT08].

3.5.2 Le composant : automate cellulaire

Un automate cellulaire est un assemblage de cellules élémentaires, chacune effectuant une opération simple en fonction de son environnement et pouvant ainsi prendre une valeur en conséquence. Ces automates sont remarquables puisque la simplicité des règles utilisées par les cellules peut se traduire en comportements parfois complexes résultants sur l'automate complet.

Les automates cellulaires peuvent en effet produire des schémas d'une forte complexité et même dans certains cas, peuvent produire des schémas impossible à prévoir, chaotiques. Dans [Wol83a], Stephen Wolfram donne une introduction détaillée sur les automates cellulaires, leur fonctionnement et les limites théoriques quant à leur caractérisation.

Pour illustrer le principe, prenons un automate cellulaire dont les cellules, pouvant prendre uniquement deux valeurs 0 ou 1, sont organisées en ligne. Le plus connu se nomme CA30. Le nombre 30 correspond au numéro de règle et reflète le comportement des cellules vis-à-vis de leur environnement. La règle 30 est détaillée dans le tableau 3.6. Celui-ci donne

[6]Pour N bits, un LFSR peut produire 2^N-1 nombres au lieu de 2^N car un LFSR ne peut produire le nombre 0. Celui-ci mettrait le LFSR dans une situation où il resterait collé à 0.

la valeur de sortie de la cellule au cycle t+1 pour toutes les valeurs possibles de voisins et de la cellule concernée au cycle t.

Le tableau est organisé de manière spécifique : les valeurs courantes de l'environnement sont dans l'ordre décroissant. De cette manière, la chaîne de bits obtenue pour les valeurs suivantes de la cellule donne le numéro de la règle de l'automate. Ici, la chaîne "00011110" représente le nombre 30, d'où le nome de l'automate CA30.

Valeur courante de l'environnement	111	110	101	100	011	010	001	000
Valeur suivante de la cellule	0	0	0	1	1	1	1	0

Table 3.6 – Règle de calcul numéro 30

Chaque cellule prend ainsi en entrée 3 valeurs :

- celle de la cellule à sa gauche
- sa propre valeur
- celle de la cellule à sa droite

Dans le cas où un seul site contient initialement la valeur '1', alors on obtient le comportement de la figure 3.24 où le temps s'écoule de manière discrète du haut vers le bas.

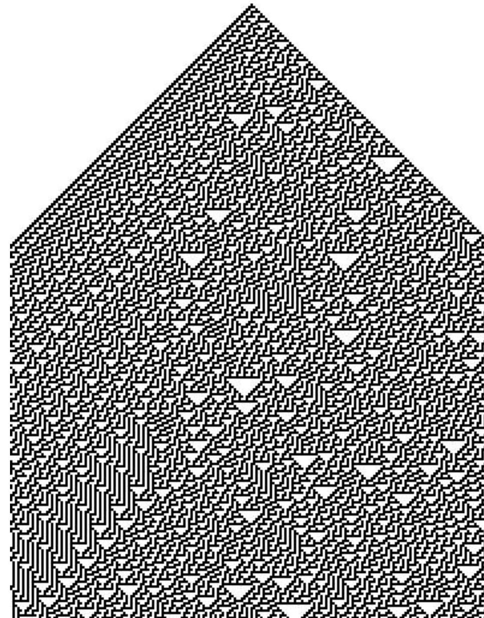

Figure 3.24 – Comportement du CA30 avec un seul site initial actif

Mais ce qui différencie les CAs ne tient pas seulement à la règle utilisée. Les comportements peuvent différer en fonction des éléments suivants :

- règle utilisée : règle de calcul, nombre de voisins et position des voisins utilisés.
- arrangement : 1D, 2D, 3D
- configuration initiale

Même si aucune caractérisation théorique ne permet de décrire l'ensemble des automates cellulaires, il est néanmoins possible d'extraire un certains nombre de propriétés empiriques. Wolfram décrit dans [Wol83b, PW85] de nombreuses caractéristiques statistiques des automates cellulaires pour une et deux dimensions[7]. Un des résultats principal concerne le classement des automates cellulaires en 4 groupes distincts :

- Classe 1 : terminent tous par mourir en ayant toutes leurs cellules à 0.
- Classe 2 : convergent vers un schéma répétitif.
- Classe 3 : présente des schémas chaotiques.
- Classe 4 : présente des schémas très complexes pouvant survivre très longtemps. Les automates de cette classe sont dit universels car à partir d'eux, il est possible d'obtenir et de simuler les automates des classes 1 à 3.

Le CA30 exposé ici appartient à la troisième classe qui est celle qui nous intéresse particulièrement. En effet, nous désirons utiliser les automates cellulaires comme composant aléatoire à l'intérieur des générateurs en exploitant les génération de schémas “chaotiques”. Utiliser les automates cellulaires semble plus intéressant que les simples LFSRs pour deux raisons majeures.

Tout d'abord l'implémentation requiert moins de matériel avec un automate cellulaire. La courbe obtenue figure 3.25 illustre la comparaison entre un CA30 et le $LFSR_{xor}$ générique décrit section 3.1 dont le nombre de bits varie entre 3 et 32.

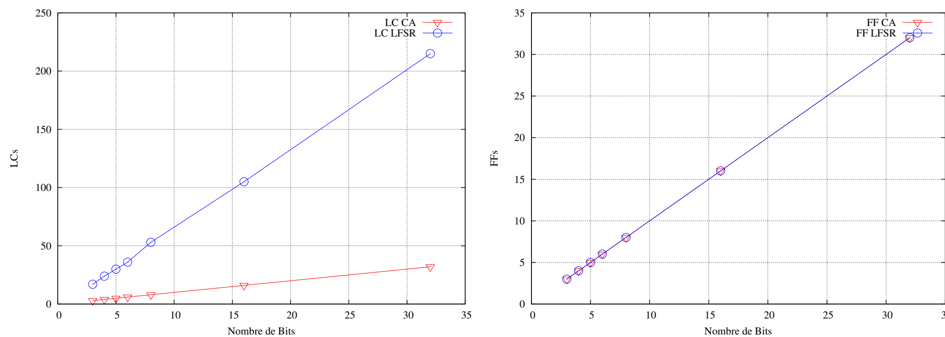

FIGURE 3.25 – Comparaison en surface d'un $LFSR_{xor}$ et d'un CA30

Il est clairement visible que l'automate cellulaire utilise moins de logique combinatoire que le $LFSR_{xor}$, alors que le nombre de registres est identique. Les analyses de fréquences ont montré une quasi-similarité entre les deux composants.

Dans [STJCS02], les auteurs montrent une comparaison intéressante concernant la qualité de l'aléatoire obtenu entre un $LFSR_{XNOR}$[60,61,62,63] et différents automates cellulaires. De plus, des analyses approfondies ont montré qu'alors que le $LFSR_{XNOR}$[60,61,62,63] ne passe que 3 des 18 tests de la suite DieHard [die][8], le CA30 lui en valide 10.

[7] Cellules réparties respectivement sur une ligne ou sur une grille

[8] Les tests DieHard sont un ensemble de tests statistiques permettant de mesurer la qualité d'un ensemble de nombres aléatoires. Cette suite de test a été développée par George Marsaglia durant plusieurs années et publiée pour la première fois en 1995. Cette suite est une référence pour tout générateur aléatoire.

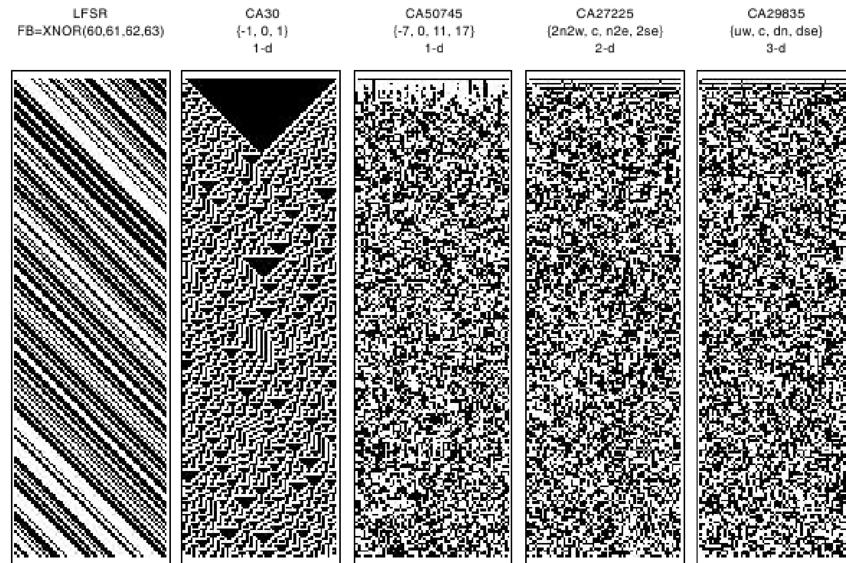

FIGURE 3.26 – Comparaison de la qualité de traces entre un LFSR et différents CA

Mais la principale difficulté est qu'il semble impossible de prévoir à partir d'un automate et de la configuration initiale si celui-ci va exprimer un comportement chaotique possédant les caractéristiques nécessaires pour une génération aléatoire correcte. Dans [Wol86] Wolfram montre qu'il est en effet difficile, voire même impossible, de résoudre ce problème dans le cas général.

De plus, pour une application optimale au sein des générateurs, il faudrait aussi analyser l'impact de l'évolution du nombre de cellules de l'automate sur le comportement obtenu. Par exemple, dans le cas d'un automate cellulaire CA30 avec seulement 5 cellules, les configurations suivantes mènent à un comportement non aléatoire : “00101”, “01010” et “01111”. Il faudrait donc disposer d'une méthode permettant de calculer automatiquement ce genre d'information afin d'introduire seulement des configurations initiales correctes dans les blocs ALEA.

3.5.3 Production de nombres aléatoires sur un intervalle quelconque

3.5.3.1 Méthode

Un composant aléatoire de type LFSR ou CA de N bits produit efficacement des nombres x$\in$ $[1..M]$ où M=$2^N - 1$ de manière équiprobable. Dans notre approche, nous devons pouvoir produire des nombres sur un intervalle quelconque $[P..Q]$ où P et Q sont des entiers naturels quelconques.

Pour réaliser ceci, nous utilisons la méthode décrite par M. Orlov dans [Orl09]. Celle-ci permet de produire, à chaque cycle et de manière équiprobable, des nombres sur tout intervalle [O..m] où m est un entier naturel quelconque.

Se ramener sur un intervalle $[0..m]$ à partir d'une génération aléatoire sur $[0..M]$ s'obtient aisément en appliquant un modulo m au résultat fournit par le bloc aléatoire. Mais cette méthode pose un problème de distribution des nombres aléatoires. Prenons un nombre m environ égal à $3/4$ de M, et appliquons l'opérateur modulo m à tout nombre

RAND_NB produit par un bloc aléatoire classique. Alors le premier quart de l'intervalle $[0..m]$ est constitué des nombres RAND_NB compris dans cet intervalle, mais aussi des nombres RAND_NB compris entre $[m..M]$ auxquels l'opération modulo m a été appliquée. Les nombres obtenus dans le premier quart de l'intervalle [0..m] auront une probabilité d'apparition deux fois supérieure aux nombres apparus dans le reste de l'intervalle.

La méthode d'Orlov utilise l'algorithme du PGCD (Plus petit diviseur commun) afin de trouver le diviseur commun entre m et $M-m$. L'idée consiste à découper les intervalles $[0..m]$ et $[m..M]$ en un même nombre de part. Un mapping est ensuite effectué entre ces deux intervalles pour distribuer les nombres obtenus dans $[m..M]$ en nombres sur $[0..m]$. La distribution ainsi obtenue est uniforme.

Alors que nous voulons produire des nombres sur $[P..Q]$, cet algorithme ne nous permet que de produire des nombres sur [0..m]. Posons $m = Q - P$. Les nombres aléatoires sont alors compris dans l'intervalle $[0..Q - P]$. En ajoutant P à tous les nombres produits par l'algorithme de Orlov, nous obtenons des nombres pseudo-aléatoires répartis de manières équiprobable entre $[P..Q]$.

Si un générateur utilise une génération de nombres aléatoires compris dans un intervalle différent de $[1..2^N]$, alors la construction de Orlov est utilisée. Le composant résultant sera appellé : Rand_Block$_{orlov}$. Dans le cas contraire, un simple LFSR est employé.

3.5.3.2 Application aux générateurs pour les vecteurs de bits

Jusqu'à maintenant les approches Horus et MyGen concernant les générateurs se sont limitées au traitement des bits. Nous présentons dans cette section comment utiliser des LFSRs conçus sur le modèle d'Orlov pour traiter des vecteurs de bits pour ces deux approches.

Comme tout type est encodé à l'aide de vecteurs bits, posséder une solution pour ceux-ci permet de traiter n'importe quel type synthétisable : entier, naturel etc. Notre approche se limite aux opérateurs de comparaison $\{\leq,<,\geq,>,=,\neq\}$, et ne prend pas en compte les opérateurs arithmétiques. Il est alors possible d'écrire des propriétés telles que :

Property H2$_{cdt}$: assume always (*Req* $\rightarrow$ *Data*="00001111" until *Ack*) ;

La propriété H2$_{cdt}$ vérifie que pour chaque requête reçue par le composant CDT, la donnée transférée est égale à une valeur fixe `0x0F`. Ce type de propriété est particulièrement utilisé pour décrire des signaux de contrôle codés sur des vecteurs de bits.

Le cas Horus : Pour synthétiser ce type d'hypothèse, deux générateurs primitifs ont été définis pour chaque opérateur de comparaison : un pour des vecteurs signés, et un autre pour des vecteurs non-signés.

Ces générateurs sont de type producteur et possèdent donc la même interface et la même architecture que le producteur gnt_Signal. Le port de sortie *Trigger* est maintenant un vecteur de N bits. Le port *Pending* associé reste sur 1 seul bit. Lors de la construction d'un générateur complexe, les feuilles de l'arbre sont composées de générateurs gnt_Signal(signal de type bit), générateurs arithmétiques (signal de type vecteur de bits).

La difficulté consiste à produire un nombre respectant la contrainte de comparaison, en un seul cycle, tout en conservant les aspects pseudo-aléatoires. Dans le cas de l'égalité (*Data*="00001111") ou de la différence (*Data*/="00001111"), aucun problème ne se pose et la création des générateurs correspondants fût directe.

Pour les comparaisons arithmétiques d'infériorité et de supériorité, l'idée consiste simplement à embarquer dans le générateur un composant Rand_Block$_{orlov}$ pour produire des nombres sur un intervalle donné, respectant ainsi les contraintes arithmétiques.

Le cas MyGen : Le traitement des vecteurs de bits est possible sans modification particulières dans l'approche. L'outil MyGen a été développé pour prendre en compte les vecteurs. Il suffirait juste de modifier légèrement l'outil MBAC pour que le fichier de sortie renseigne sur les types des signaux utilisés. Enfin, en embarquant dans le générateurs des blocs aléatoires de type Rand_Block$_{orlov}$, le traitement des vecteurs de bits pourrait être effectué.

3.6 Bilan

La plateforme Horus fournit un moyen efficace pour la synthèse de propriétés. Assertions et hypothèses sont transformées en moniteurs et générateurs et permettent une instrumentation efficace du circuit à vérifier en contrôlant à la fois la génération de stimuli et l'analyse des réponses fournies par le circuit.

La combinaison de l'approche Horus et MyGen peut être intéressante pour deux raisons. Horus est plus efficace que MyGen pour la synthèse d'hypothèses FLs et fournit un outil possédant plus de fonctionnalités d'aide à l'instrumentation que MBAC ou MyGen. Mais MyGen est bien plus approprié à la production de générateurs SERE. Il possède de plus plusieurs modes de génération aléatoire qui permettent d'adresser efficacement différentes phases de test.

La prise en compte de vecteurs de bits est indispensable pour traiter des cas réels et apporte ainsi un pouvoir d'expression fondamental pour la synthèse de générateurs. La solution présentée se limite cependant aux opérateurs de comparaison à une valeur constante. Elle ne prend pas en compte les opérateurs arithmétiques (addition, soustraction etc...). Malgré tout, nous pensons que l'utilisation d'opérations arithmétiques est beaucoup plus rarement utilisé que les opérations de comparaison lors de l'écriture de propriétés logico-temporelles. Même si des travaux futurs s'orientent naturellement dans cette direction, cette limitation n'a jamais posé problème jusqu'à maintenant, même pour des circuits complexes issus du monde industriel.

Le chapitre suivant détaille comment combiner les générateurs pour obtenir un composant global modélisant le comportement d'un environnement décrit par un ensemble d'hypothèses logico-temporelles.

Chapitre 4

Modélisation de l'environnement

Sommaire

4.1 Incohérences 78

4.1.1 Cohérence d'une propriété 78

4.1.2 Cohérence d'une spécification 79

4.2 Résolution d'incohérences : Description mathématique du problème 80

4.3 Algorithme de résolution pour les systèmes Sys_1 81

4.3.1 Circuit de Résolution pour des signaux de type Bit : $Solver_bit_{sys1}$ 83

4.3.2 Circuit de Résolution pour des signaux de type vecteur de Bits : $Solver_vect_{sys1}$ 85

4.4 Algorithme de résolution pour les systèmes Sys_2 87

4.4.1 Analyse statique d'un système Sys_2 87

4.4.2 Solver Sys_2 89

4.4.3 Résultats expérimentaux 89

4.4.4 Comparaisons $Solver_{sys2}$ et $Solver_vect_{sys1}$ pour des systèmes Sys_1 89

4.4.5 Analyse de couverture d'un ensemble de propriétés 90

4.5 Bilan 91

4.1 Incohérences

Considérons un environnement décrit par un ensemble de propriétés logico-temporelles. La construction d'un modèle d'environnement consiste à combiner les générateurs de ces propriétés. Or cette composition n'est pas triviale pour deux raisons.

Le même signal peut être présent plusieurs fois dans une hypothèse, on dit qu'il est dupliqué. Il est produit par différents générateurs gnt_Signal et ce signal possède donc plusieurs sources. Une résolution doit être effectuée pour produire une valeur finale correcte pour le signal dupliqué.

Le même signal peut aussi être présent plusieurs fois dans diverses propriétés composants la spécification de l'environnement. Une fois encore, une résolution est nécessaire afin de déterminer la valeur finale du signal produit par plusieurs générateurs complexes.

Enfin, nous verrons comment effectuer la résolution dans des cas plus complexes où chaque duplication d'un signal est décrite par un ensemble de valeurs correctes (et non pas par une seule valeur).

4.1.1 Cohérence d'une propriété

Lorsqu'un même signal est présent plusieurs fois dans une propriété, plusieurs générateurs gnt_Signal pilotent ce même signal qui a donc plusieurs sources. Un mécanisme de résolution est requis pour déterminer la valeur finale à donner au signal.

Deux concepts distincts doivent être différenciés : *incohérence* et *réalisabilité* d'une propriété.

4.1.1.1 Incohérence

Dans ce cas, aucune trace ne peut satisfaire la propriété donnée. La propriété $\mathrm{H2}_{arbitre}$ suivante illustre ceci :

$$\textsf{property } \mathrm{H2}_{arbitre} \textsf{ is } \text{assume } \textsf{always}(\mathrm{use}_i \textsf{ and } \textsf{not}(\mathrm{use}_i))\,;$$

La propriété $\mathrm{H2}_{arbitre}$ est incohérente car à chaque instant le signal use_i est contraint à deux valeurs différentes.

Une propriété sera dite incohérente si, quelque soit la combinaison des signaux la composant, aucune trace ne satisfait la propriété. Dans ces cas là, le générateur ne peut produire de valeur valide.

4.1.1.2 Réalisabilité

D'autres propriétés peuvent présenter des incohérences sous certaines conditions, et posséder des traces satisfaisant la propriété dans les autres cas. On dit que la propriété est partiellement réalisable. C'est le cas de la propriété $\mathrm{H3}_{arbitre}$ suivante :

$$\textsf{property } \mathrm{H3}_{arbitre} \textsf{ is } \text{assume } \textsf{always}(\mathrm{use}_i \rightarrow (\textsf{next!}[1]\textsf{not}(\mathrm{use}_i)))$$

Si le signal use_i est actif durant deux cycles consécutifs, alors au second cycle, la propriété présente une incohérence puisque use_i est contraint à deux valeurs différentes. Mais si le signal use_i n'est jamais actif durant deux cycles consécutifs, alors toutes les traces respecteront la propriété $\mathrm{H3}_{arbitre}$.

Si aucune trace ne produit d'incohérence, la propriété est dite totalement réalisable. Aucune incohérence n'est possible. C'est le cas pour des propriétés possédant des instances de signaux tous différents entre-eux.

L'incohérence d'une propriété peut être analysée par des outils spécifiques tels que RAT [BCE+04]. Cet outil d'aide à l'écriture de propriétés peut fournir à l'utilisateur un rapport sur l'incohérence d'une ou plusieurs propriétés.

Les propriétés incohérentes doivent être soit supprimées, soit ré-écrites. Celles partiellement réalisables, peuvent être utilisées puisque valides sous certaines conditions. Il est alors du devoir du générateur d'analyser à tout moment que la séquence de signaux produite, respecte la propriété, et ne produit aucune incohérence.

4.1.2 Cohérence d'une spécification

Un environnement peut être spécifié sous la forme d'un ensemble de propriétés. Le modèle sera alors construit en combinant les générateurs obtenus pour chaque propriété de la spécification. Ceci pose un nouveau problème : plusieurs sources pour un même signal peuvent exister au travers de plusieurs propriétés. Chaque source contraint le signal à un ensemble de valeurs. Prenons les deux propriétés Arith_1 et Arith_2 suivantes :

- property Arith_1 is always($cond1 \rightarrow sig \geq 3$) ;
- property Arith_2 is always($cond2 \rightarrow sig = 4$) ;

Si le signal *cond1* est actif, alors l'ensemble de valeurs correctes pour *sig* est tout entier supérieur à 3. Si *cond2* est actif, alors l'ensemble de valeurs correctes pour *sig* est réduit au seul entier 4.

Dans le cas où un signal possède plusieurs sources actives à un instant donné, la valeur pour ce signal appartiendra à l'intersection de tous les ensembles pour chacune des instances du signal.

Il faut alors s'assurer qu'à un instant donné, les propriétés contraignent les instances d'un même signal à un ensemble de valeurs non vide. On parle alors de vérification de la cohérence d'une spécification. De même que pour les incohérences au niveau des propriétés, deux concepts distincts doivent être différenciés pour un ensemble de propriétés : incohérence et réalisabilité d'une spécification.

4.1.2.1 Incohérence

Un ensemble de propriétés est incohérent si l'ensemble des traces satisfaisant cet ensemble est vide. Par exemple, les deux propriétés suivantes sont incohérentes :

- property Inc1 is assert always (a and c)
- property Inc2 is assert always (a and not(c))

Des outils tels que RAT (Requirement Analysis Tool) [BCE+04] ou la méthode utilisée pour construire les Cando Objects [SNBE07b] peuvent être utilisés pour détecter statiquement les incohérences.

Un ensemble de propriétés sera dit incohérent s'il n'existe aucune trace satisfaisant celui-ci.

4.1.2.2 Réalisabilité

Il est possible qu'une ou plusieurs propriétés soient satisfaites sous certaines contraintes, mais possèdent des incohérences si ces contraintes ne sont pas respectées. Le couple de propriétés suivant illustre ceci :

- assert always $(a \Rightarrow c)$
- assert always $(b \Rightarrow \text{not}(c))$

Si a et b possèdent des valeurs différentes, alors la spécification est réalisable. Mais si ces deux signaux sont actifs au même cycle, le signal c est contraint à deux valeurs différentes.

Il est alors possible de résoudre ce problème en ajoutant des hypothèses à la spécification. Pour la spécification précédente, l'ajout de l'hypothèse suivante résout le problème : assume never $(a \text{ and } b)$. Ceci contraint l'environnement du circuit à synthétiser à ne jamais activer les signaux a et b en même temps.

Si la spécification est cohérente, alors quelque soit la combinaison des entrées respectant les hypothèses, il existe toujours une valeur correcte pour toutes les sorties.

La détection des incohérences et la résolution des signaux dupliqués s'effectue à l'aide de deux mécanismes :

- un *signal de contrainte* : associé à chaque signal à produire, il indique si au cycle courant, le signal correspondant est contraint par le générateur, ou produit par le bloc aléatoire.
- un *composant de résolution* : il prend en entrée l'ensemble des duplications d'un signal ainsi que leurs contraintes associées. Il fournit en sortie la valeur du signal final, ou détecte une erreur le cas échéant.

4.2 Résolution d'incohérences : Description mathématique du problème

Le problème de résolution d'incohérences peut-être modélisé par un système linéaire de N équations (si il y a N duplications) à une seule inconnue (le signal à résoudre). Ce système possède un ensemble de solutions, éventuellement vide.

De nombreuses techniques permettent de résoudre ce type de problème. L'algorithme du Simplex [Dan98] est le plus connu. Malgré une complexité exponentielle dans le pire cas, une réponse est fournie en temps polynomial dans la plupart des cas, ce qui l'a rendu populaire.

Depuis quelques années, les outils SAT (outils résolvant des problèmes de satisfaisabilité d'ensembles d'équations booléennes) ont été adaptés à la résolution de problèmes plus complexes prenant en considération des variables entières, réelles, des tableaux etc. La résolution de ce type de problème fait appel à des technique SMT (SAT Modulo Theory) [Cim08]. Ceci consiste à résoudre des équations logiques étant donné une théorie spécifique. Typiquement, les théories utilisées sont celles des entiers ou des réels. Les outils SMT ont connu récemment des améliorations conséquentes augmentant considérablement leur pouvoir [MB08, JLS09, WHM09].

Alors que ces techniques analysent statiquement les équations afin de fournir une réponse au problème, nous avons besoin de résoudre des systèmes linéaires de contraintes dynamiquement. De plus la résolution de ces systèmes doit être réalisée de manière matérielle, puisqu'embarquée au sein des générateurs. Bayliss *et al.* ont porté l'algorithme du

Simplex sur FPGA [BBCL06], augmentant ainsi l'efficacité de l'algorithme d'un facteur 100. Malheureusement, cette technique n'est pas utilisable dans notre cas puisque nous sommes soumis aux contraintes suivantes :

- la résolution doit s'effectuer en un seul cycle d'horloge.
- le circuit doit être minimal et doit représenter une part non significative de la surface totale utilisée par un ensemble de générateurs.
- si le système de contraintes possède plusieurs solutions, chacun d'elle doit avoir la même probabilité d'apparition.

Nous donnons ici un composant respectant ces critères, capable de fournir en temps réel une solution à certains types de systèmes. Pour réaliser ceci, nous avons restreint le type de système de contraintes pris en compte à des systèmes très simples où une seule variable entre en jeu.

La résolution est implémentée sous forme d'un composant matériel fournissant une solution au problème de résolution à la volée. En contrepartie, nous simplifions le problème initial en se restreignant dans un premier temps aux systèmes linéaires de la forme :

$$Sys_1 \begin{cases} x = \alpha_1 \\ ... \\ x = \alpha_n \end{cases} \tag{4.1}$$

Ce type de système, noté dans la suite $\textsc{Sys}_1$, possède soit une unique solution α_1, soit aucune solution et une erreur est levée pour rapporter l'incohérence détectée.

Ensuite, nous élargirons le problème aux systèmes que nous appellerons $\textsc{Sys}_2$. Ceux-ci sont de la forme :

$$Sys_2 \begin{cases} x \mathcal{R} \alpha_1 \\ ... \\ x \mathcal{R} \alpha_n \end{cases} \tag{4.2}$$

Où $\mathcal{R}$ est une des relations arithmétiques suivantes :$\{<,>,\leq,\geq,=,\neq\}$. Dans ce cas la solution est un ensemble de valeurs qui peut être éventuellement vide dans le cas d'une incohérence. Encore une fois nous proposons une approche basée sur un composant matériel résolvant à la volée des systèmes linéaires d'équations de type $\textsc{Sys}_2$.

4.3 Algorithme de résolution pour les systèmes Sys_1

Afin de résoudre les signaux possédant plusieurs sources, nous avons choisi d'ajouter un composant de résolution de signaux. Ce dernier prend en entrée les différentes instances d'un même signal, analyse leurs valeurs, et retourne une réponse pouvant prendre deux types de valeur différentes. Dans le premier cas, aucune incohérence n'est détectée. Une sortie de ce circuit renvoie la valeur correspondant à la résolution du signal.

Dans le cas où une incohérence est détectée, une sortie d'erreur passe à '1' et la valeur du signal qui a été résolue n'est pas pertinente. Avec l'implémentation actuelle, la génération de tests continue même si le signal d'erreur passe à '1' durant un cycle.

La solution présentée ici amène donc une réponse partielle au problème cité en introduction puisque nous détectons, mais dans les cas où la résolution ne peut être effectuée

(deux signaux contraints par des valeurs différentes au même cycle), notre méthode signale simplement à l'utilisateur l'incohérence. Aucune correction à la volée de la valeur n'est effectuée.

Un composant de résolution possède deux ports d'entrée *SIG* et *Pending* permettant de résoudre la valeur d'un signal *sig* dupliqué N fois. Le premier port prend en entrée les N duplications *sig*(i) de *sig* tandis que le second prend les N duplications *Pending*(i) associées. Il possède un port de sortie *Val* fournissant la valeur résolue pour *sig*.

Dans le cas d'une spécification cohérente, si plusieurs duplications sont contraintes à un instant donné, alors elles possèdent toutes la même valeur pour *sig*. Dans ce cas, la résolution s'effectue selon les trois règles suivantes.

La règle 4.3 s'applique dans le cas où aucune duplication n'est contrainte. La résolution fournit '0' par défaut. Une valeur aléatoire est produite sur *Val* si le mode RANDOM est sélectionné.

$$\forall k \in [1..N], Pending(k) = 0 \Rightarrow Val = 0 \tag{4.3}$$

La règle 4.4 s'applique si seule la *i-ème* duplication est contrainte, on transmet *SIG*(i) sur *Val*.

$$\exists! k \in [1..N], Pending(k) = 1 \Rightarrow Val = SIG(k) \tag{4.4}$$

La règle 4.5 s'applique si plusieurs duplications sont contraintes, alors la valeur de la première duplication contrainte est transmise sur *Val*.

$$\begin{cases} \exists J/(J \subseteq N), \forall j1, j2 \in J \\ Pending(j1) = Pending(j2) = 1 \Rightarrow Val = SIG(min(j1)) \end{cases} \tag{4.5}$$

Pour une spécification dont la cohérence n'a pas été vérifiée, une version améliorée des composants de résolution doit être utilisée. Il est possible qu'à un instant donné , il existe deux duplications i et j contraignant un signal à des valeurs différentes. Dans ce cas, une erreur doit être détectée. Le composant de résolution possède alors un port de sortie *Err* signalant les problèmes de cohérence. Dans ce contexte, la règle 4.5 est remplacée par les deux règles 4.6 et 4.7.

La règle 4.6 est utilisée si plusieurs duplications sont contraintes, alors la valeur de la première duplication contrainte est transmise sur *Val*.

$$\left\{ \begin{array}{l} \exists J \subseteq N, \forall d1, d2 \in J \\ Pending(d1) = Pending(d2) = 1 \\ Val(d1) = Val(d2)) \end{array} \right\} \Rightarrow Val = SIG(min(d1)) \wedge Err = 0 \tag{4.6}$$

La règle 4.7 est appliquée si plusieurs duplications sont contraintes, et si au moins l'une d'entre elle possède une valeur différente des autres. Dans ce cas, le port de sortie *Err* est passé à '1' pour signaler l'incohérence. La valeur présente sur *Val* n'est pas significative.

$$\left\{ \begin{array}{l} \exists J \subseteq N, \exists d1, d2 \in J \\ Pending(d1) = Pending(d2) = 1 \\ Val(d1) \neq Val(d2) \end{array} \right\} \Rightarrow Val = 0 \wedge Err = 1 \tag{4.7}$$

La suite de ce chapitre donne les détails des circuits de résolution pour des signaux dupliqués de type bit (composant SOLVER_BIT_{sys1}) et vecteur de bits ($\text{SOLVER_VECT}_{sys1}$).

Ces composants effectuent les résolutions si les signaux sont sous le contrôle d'un opérateur d'égalité (Ex : *sig*="0101"). Dans le cas plus général d'opérateurs de comparaisons (Ex : *sig*>"011"), un composant de résolution plus évolué est utilisé : SOLVER_{sys2}. Enfin, nous montrerons comment, à l'aide de ces composants de résolution, le modèle d'environnement peut être construit.

4.3.1 Circuit de Résolution pour des signaux de type Bit : $\textbf{Solver_bit}_{sys1}$

Le composant SOLVER_BIT_{sys1} a été conçu de façon modulaire : l'un pour détecter les erreurs (SOLVER_{err}), et un autre pour la résolution de valeurs de signaux (SOLVER_{sig}).

4.3.1.1 Module de détection d'incohérence : $\textbf{Solver}_{err}$

Un premier composant a été défini afin de détecter les incohérences susceptibles d'apparaître lors de la génération de signaux pour une formule donnée : SOLVER_{err}. Considérons N duplications d'un signal donné. Le composant SOLVER_{err} correspondant possède l'interface suivante :

- *Pending* (N bits) : un bit *Pending*(i) correspond au port de sortie *Pending* associé à la i-ème occurrence d'un signal A.
- *SIG* (N bits) : un bit *SIG*(i) correspond au port de sortie *Trigger* associé à la i-ème occurrence d'un signal A.
- *Err* (1 bit) : cette sortie est active si une contradiction est détectée. Sinon elle vaut '0' ;

Supposons qu'un signal A soit répété N fois dans la formule. Le générateur possède alors N composants gnt_Signal pour la génération d'un même signal. Les ports *Trigger* et *Valid* de la i-ème occurrence du composant gnt_Signal seront connectés respectivement aux bits *SIG*(i) et *Pending*(i) du composant SOLVER_{err}.

Le circuit SOLVER_{err}est basé sur les équations booléennes 4.8.

$$Solver_{err} \begin{cases} \forall i, j \in [1..N] \ : \\ Err = \bigvee (Pending(i) \wedge Pending(j) \wedge (SIG(i) \oplus SIG(j))) \end{cases} \tag{4.8}$$

Cette formulation simple permet de définir précisément le nombre de portes utilisées par ce circuit pour traiter les conflits éventuels sur un nombre N de signaux :

- nombre de portes `xor` : N*(N-1)/2
- nombre de portes `and` : N*(N-1)
- nombre de portes `or` : N-1

Le point critique pour les performances du circuit est largement influencé par le nombre de portes `xor`. Néanmoins, il est rare que le nombre de signaux identiques utilisés dans une même formule dépasse quelques unités, notre solution est largement réalisable dans la majorité des cas.

Ce composant est basé sur des cellules de bases pour traiter chaque bit. La création d'un composant pour N duplications et faite de manière automatique à l'aide de paramètres génériques.

4.3.1.2 Module de résolution : $\textbf{Solver}_{sig}$

Il effectue la résolution des signaux présents en entrée. Il dispose des même entrées que SOLVER$_{err}$. Sa sortie *Val* représente la valeur du signal résolu (si *Err*='0'). Nous appelons résolution le calcul de la valeur d'un signal A à partir des signaux *SIG*(i) représentant les duplications de ce signal, et des *Pending*(i) associés. Considérons deux signaux dupliqués. Plusieurs cas peuvent se produire :

- *Err*=1 : La valeur qui est sortie sur *Val* n'est pas pertinente.
- *Pending*(1) et *Pending*(2) sont à '1' et *Err*='0' : *SIG*(1) et *SIG*(2) ont la même valeur et *Val* prend cette valeur. Les deux signaux sont contraints et ont la même valeur.
- *Pending*(1)='0' et *Pending*(2)='1' : *Val* prend la valeur de *SIG*(2) puisque seul ce signal est contraint.
- *Pending*(1) et *Pending*(2) sont à '0' : si le mode RANDOM est actif, *Val* prend une valeur pseudo-aléatoire, sinon elle est fixée à '0'.

Ainsi l'architecture de notre circuit pour N signaux est basée sur les équations booléennes 4.9 :

$$Solver_{sig}\begin{cases}\forall i \in [1..N] \\ Val = \bigvee(Pending(i) \wedge SIG(i))\end{cases} \tag{4.9}$$

Nous pouvons encore une fois donner une estimation précise du nombre de portes logiques utilisées pour N signaux en entrée :
- nombre de portes `and` : N
- nombre de portes `or` : N-1

De la même manière que pour le composant précédent, le code VHDL est complètement générique.

4.3.1.3 Le composant global : $\textbf{Solver_bit}_{sys1}$

La description VHDL de ce circuit correspond simplement à l'association des deux composants précédents SOLVER$_{err}$ et SOLVER$_{sig}$. Une instance de ce circuit pour un signal dupliqué 3 fois est fourni figure 4.1.

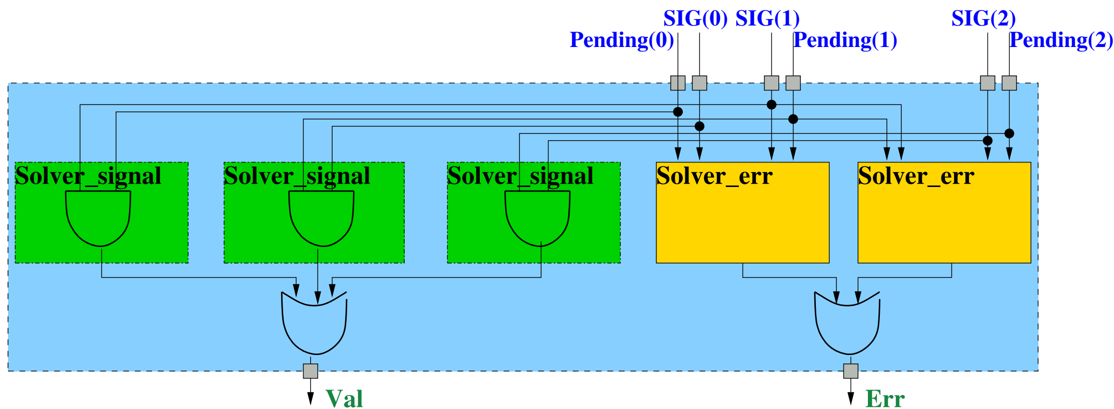

FIGURE 4.1 – Instance du composant SOLVER_BIT$_{sys1}$ pour 3 signaux dupliqués

4.3.2 Circuit de Résolution pour des signaux de type vecteur de Bits : Solver_vect$_{sys1}$

Le traitement des vecteurs, bien que reposant sur les même principes que pour les résolution de signaux de type bits, est légèrement plus complexe à mettre en oeuvre. Le composant SOLVER_VECT$_{sys1}$ possède la même interface que SOLVER_BIT$_{sys1}$.

Il possède aussi une architecture modulaire, composée de briques élémentaires PRIM_SOLVER interconnectées entre elles. Chaque élément PRIM_SOLVER prend en entrée deux occurrence du signal de type vecteur et les 2 signaux de contrainte associés. Il possède en sortie un port d'erreur indiquant si une cohérence a été levée et un port *Val* fournissant la valeur résolue du vecteur.

Alors que pour le traitement de cohérence des signaux de type bit, le composant SOLVER_BIT$_{sys1}$ traite toutes les occurrences en parallèle, le calcul pour les vecteurs est formé d'une chaîne combinatoire de PRIM_SOLVER traitant l'une après l'autre 2 duplications d'un signal et fournissant en sortie finale la valeur résolue du vecteur dupliqué. Nous nommons cette structure "chaîne de résolution". Une instance de circuit SOLVER_VECT$_{sys1}$ pour un signal dupliqué 5 fois est fourni figure 4.2. La chaîne de résolution est composée de 4 PRIM_SOLVER.

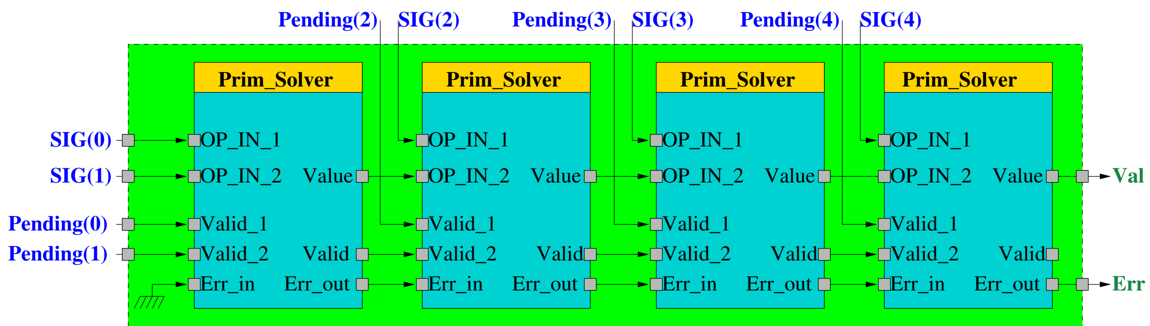

FIGURE 4.2 – Instance du composant SOLVER_VECT$_{sys1}$ pour un signal dupliqué 5 fois

Soit un signal *sig* dupliqué N fois, et *sig*(i) une duplication de ce signal. La résolution s'effectue s'effectue en comparant les valeurs de tous les couples (*sig*(i),*sig*(i+1)) avec $i \in [1..N]$ ayant au moins une des deux duplications contraintes. Cette comparaison est effectuée par le composant PRIM_SOLVER.

Si toutes les valeurs contraintes sont égales deux à deux, alors toutes les valeurs sont égales et la résolution peut être effectuée. Si au moins une duplication possède une valeur qui diffère des autres, alors une incosistence est détectée.

Le composant PRIM_SOLVER possède l'interface suivante :

- *SIG*(0) : valeur courante d'une duplication du signal considéré.
- *SIG*(1) : pour le premier composant PRIM_SOLVER de la chaîne, une duplication du signal considéré est passée sur ce port. Dans les autres cas, la valeur de résolution temporaire provenant du circuit PRIM_SOLVER précédent est transmise sur ce port.
- *Pending*(0) : contrainte associée au port *SIG*(0).
- *Pending*(1) : contrainte associée au port *SIG*(1) provenant soit d'une duplication du signal considéré (cas où le PRIM_SOLVER est en tête de la chaîne de résolution),

soit indiquant que le PRIM_SOLVER précédent est actif est que la valeur passée sur le port *SIG*(1) est contrainte.

- *Err_in* : si ce port est à '1', alors une erreur de résolution a été detectée par un des composants PRIM_SOLVER précédent.
- *Value* : valeur de résolution temporaire (resp. finale) du signal pour un PRIM_SOLVER au coeur de la chaîne (resp. en queue de chaîne).
- *Valid* : indique si le PRIM_SOLVER courant est actif. Si c'est le cas, alors une résolution est en cours et la valeur présente sur *Value* est contrainte.
- *Err_out* : ce port est actif si une erreur de résolution a été détectée par le PRIM_SOLVER courant, ou par un PRIM_SOLVER précédent.

Comme l'illustre le code source de la figure 4.3, le composant PRIM_SOLVER est totalement combinatoire.

```
1  ELT_COMP:process(OP_IN_1,OP_IN_2,Pending_1,Pending_2,Err_in)
2  begin
3      if Err_in='1' then -- error : propagate Err signal
4          Err_out<=Err_in;
5          Value<=(others=>'0');
6          Valid<='0';
7      elsif Pending_1='0' and Pending_2='0' then -- no constraint
8          Err_out<='0';
9          Value<=rand_nb;
10         Valid<='0';
11     elsif Pending_1='0' and Pending_2='1' then -- one constraint
12         Err_out<='0';
13         Value<=OP_IN_2;
14         Valid<='1';
15     elsif Pending_1='1' and Pending_2='0' then -- one constraint
16         Err_out<='0';
17         Value<=OP_IN_1;
18         Valid<='1';
19     elsif Pending_1='1' and Pending_2='1' then -- two constraints
20         if(OP_IN_1=OP_IN_2) then -- consistency => resolution OK
21             Err_out<='0';
22             Value<=OP_IN_1;
23             Valid<='1';
24         else -- inconsistency => resolution KO
25             Err_out<='1';
26             Value<=(others=>'0');
27             Valid<='0';
28         end if;
29     end if;
30 end process;
```

FIGURE 4.3 – Process VHDL du composant PRIM_SOLVER

Si aucun des ports *Pending*(0) et *Pending*(1) ne sont actifs, alors le composant est inactif (lignes 11 à 14), la valeur produite sur la sortie *Value* est soit aléatoire, soit fixée à zéro selon la valeur du paramètre générique RANDOM. Toutes les autres combinaisons de valeurs pour les ports *Pending*(0) et *Pending*(1) activent le composant PRIM_SOLVER, ce qui active implicitement le reste de la chaîne de résolution.

Si un seul signal est contraint (lignes 15 à 22 figure 4.3), sa valeur est passée directement sur le port de sortie *Value* et la sortie *Valid* est activée. Ceci active de proche en proche tout le reste des composants PRIM_SOLVER de la chaîne de résolution. La résolution globale réussie si tout autre signal contraint possède la valeur passée sur le port *Value*.

Si les deux signaux sont contraints (lignes 23 à 32), la sortie *Valid* est aussi passée à '1' pour activer le reste de la chaîne. Si les duplications ont toutes la même valeur (lignes 24 à 27), celle-ci est passée sur le port de sortie *Value*.

Mais dans le cas où les valeurs sur $SIG(0)$ et $SIG(1)$ sont différentes, une incohérence est détectée (lignes 28 à 31). La chaîne des ports *Err_out* transmet l'information d'erreur jusqu'à la sortie *Err* du solveur global. Les composants suivant n'effectuent plus aucune vérification puisque la résolution est alors impossible. Si aucune erreur n'est levée, la sortie *Value* permet de faire passer au composant suivant la valeur résolue pour le couple de duplication courant. Cette valeur est transmise de composant en composant et fournit la valeur finale du signal résolu.

4.4 Algorithme de résolution pour les systèmes Sys_2

Alors que pour les systèmes de contraintes de type SYS_1, l'ensemble des solution était réduit à une seule, voire aucune solution, ce n'est plus le cas pour les systèmes SYS_2. L'ensemble des solutions peut contenir un nombre quelconque de solutions.

Dans la suite de cette section, nous nous intéressons à des contraintes arithmétiques portant sur des signaux. Mais jusqu'à maintenant, nous parlions de “contrainte” pour exprimer le fait que la valeur d'un signal n'est pas aléatoire, mais fixée par un générateur afin de satisfaire la propriété associée. Pour clarifier la lecture de cette section, un signal contraint sera dit actif.

La construction de composants de résolution pour ce type de systèmes passe par une analyse statique de l'ensemble de propriétés concernées. Toutes les contraintes arithmétiques pour chaque signal sont analysées et les systèmes d'équations sont construits.

A partir de ce système, une analyse statique est effectuée et le composant de résolution final est construit : le SOLVER_{sys2}.

Prenons par exemple la spécification S1 suivante :

- property Arith_1 is always(*cond(1)* $\Rightarrow x \geq 3$) ;
- property Arith_2 is always(*cond(2)* $\Rightarrow x = 4$) ;
- property Arith_3 is always(*cond(3)* $\Rightarrow x \leq 8$) ;
- property Arith_4 is always(*cond(4)* $\Rightarrow x \geq 10$) ;

Chaque signal de la spécification est analysé. Pour chaque signal dupliqué, l'ensemble des contraintes arithmétiques associées aux duplications est calculé. Pour la spécification S1, nous obtenons l'ensemble de contraintes arithmétiques C1 suivant pour le signal x :$\{x \geq 3, x = 4, x \leq 8, x \geq 10\}$. Nous notons $Pending(0)..Pending(3)$ les informations d'activité des duplications x de S1.

4.4.1 Analyse statique d'un système Sys_2

L'ensemble C1 donne une représentation statique de toutes les contraintes arithmétiques pouvant être actives à un instant donné pour un signal x. En pratique, seul un

sous ensemble de ces contraintes arithmétiques doit être résolu à un cycle donné puisque toutes les duplications d'un signal x peuvent ne pas être actives à un instant précis.

Supposons que le signal *cond* soit égal à "1100" à un cycle t, alors seules les deux premières contraintes arithmétiques de C1 sont actives, puisque seules les deux premières duplications de x sont actives dans S1.

Chaque combinaison possible de contraintes arithmétiques est analysée et les ensembles solutions sont calculés. Tout ensemble solution possède l'une des 5 cinq formes suivantes : {k, [k..], [..k],[k..l], ⊘}, où ⊘ dénote l'ensemble vide, dans le cas où aucune solution n'est possible.

Soit P l'ensemble décrivant les contraintes arithmétiques portant sur un signal x. L'analyse de toutes les combinaisons possibles de duplications actives s'effectue en analysant chaque partie de l'ensemble P. Pour chaque partie, une système d'équation de type $\text{S}{\scriptstyle\text{YS}}_2$ est obtenu. Nous obtenons l'ensemble de systèmes 4.10 pour les parties de C1 à trois éléments :

$$\begin{cases} x \geq 3 \\ x = 4 \\ x \leq 8 \end{cases} \quad \begin{cases} x \geq 3 \\ x = 4 \\ x \geq 10 \end{cases} \quad \begin{cases} x = 4 \\ x \leq 8 \\ x \geq 10 \end{cases} \quad \begin{cases} x \geq 3 \\ x \leq 8 \\ x \geq 10 \end{cases} \tag{4.10}$$

L'ensemble solution est ensuite calculé pour chacun des systèmes obtenus. Il peut être éventuellement vide. A chaque ensemble solution est associé l'information d'activité des duplications. Autrement dit, nous enregistrons l'ensemble des duplications actives pour chaque ensemble solution. Pour le premier système de l'équation 4.10, nous sauvegardons le fait que les trois premières duplications sont actives. Ceci est modélisé sous forme d'équations booléennes sur les signux *Pending* associées aux duplications. Pour le premier système d'équation de l'équation 4.10, nous avons *Pending*(0)='1', *Pending*(1)='1', *Pending*(2)='1' et *Pending*(3)='0'.

A la fin du calcul de tous les ensembles solutions, des optimisations sont effectuées sur ces équations dans le cas où plusieurs combinaisons différentes de duplications actives, mènent au même ensemble solution.

Comme le nombre de parties dans un ensemble de N élément s'élève à 2^N, le nombre de systèmes à résoudre explose rapidement avec le nombre de contraintes. Heureusement, le nombre d'ensembles solution croît généralement moins vite que le nombre de systèmes obtenus pour plusieurs raisons.

Premièrement, tout ensemble de contraintes contenant au moins un opérateur d'égalité produit un ensemble solution qui se réduit soit à un nombre fixe, soit à l'ensemble vide. Par exemple, l'ensemble de contraintes $\{x \geq \ 3, x = 4\}$ se réduit à l'ensemble solution suivant $\{4\}$.

De plus de nombreuses combinaisons de contraintes produisent le même ensemble solution, le nombre d'ensembles solution est en pratique proportionnel au nombre de contraintes. Nous obtenons la liste d'ensembles solution suivante pour l'ensemble de contraintes C1 :{4, [3..8], [10..],[3..],[v..8], ⊘}.

Finalement, un composant Rand_Block$_{orlov}$ est créé pour chaque ensemble solution. Celui-ci permet de produire des nombres sur l'intervalle solution approprié. Pour les ensembles du type [k..], l'ensemble possède en pratique une borne supérieure fixée par le bloc aléatoire utilisé. Pour les ensembles réduits à une unique solution k, le générateur

gnt_eq est utilisé. Lorsqu'il est actif, il produit le nombre k, sinon il produit une valeur aléatoire (ou une valeur nulle si le mode RANDOM est inactif).

En combinant ces composants entre eux, nous obtenons un nouveau composant de résolution permettant de résoudre la valeur d'un signal dont les duplications sont contraintes par des inégalités arithmétiques : le SOLVER_{sys2}. La suite de cette section détaille l'architecture de ce composant.

4.4.2 Solver Sys_2

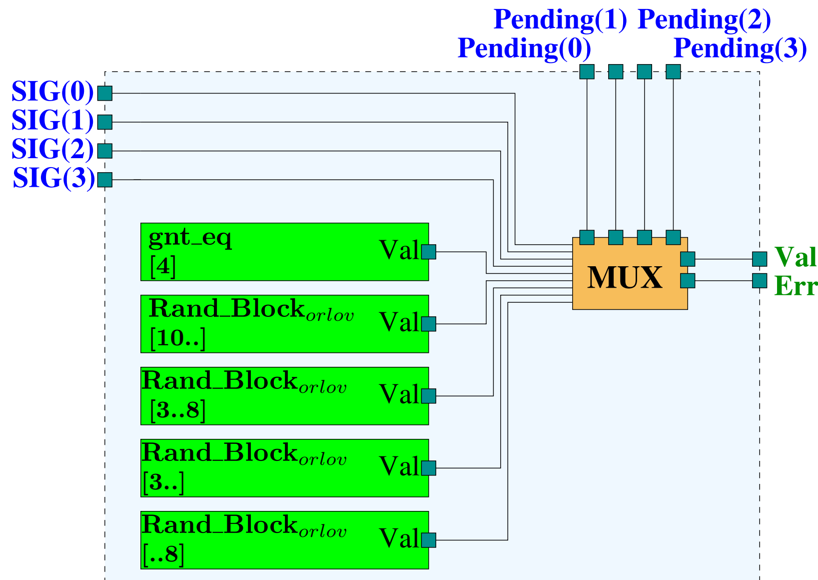

FIGURE 4.4 – Architecture du SOLVER_{sys2}pour la spécification S1

La figure 4.4 donne l'architecture du SOLVER_{sys2} pour le signal x de la spécification S1. Celui-ci est composé des blocs aléatoires fournissant des valeurs sur les ensembles solution des systèmes obtenus à partir de l'ensemble de contraintes C1. Le port d'entrée *SIG*(i-1) (resp. *Pending*(i-1)) correspond à la i-ème duplication (resp. la i-ème information d'activation) du signal x de S1.

Le composant MUX établit le lien entre les blocs aléatoires (en fait les ensembles solutions) et l'état d'activité des duplications. Si les trois premières duplications de x sont actives (ceci correspond au premier système SYS_2 de l'équation 4.10), alors l'ensemble solution est [=4], et le composant gnt_eq[4] est selectionné. Le composant MUX pilote aussi la sortie d'erreur dans le cas où l'ensemble solution est réduit à l'ensemble vide.

4.4.3 Résultats expérimentaux

4.4.4 Comparaisons Solver_{sys2} et $\text{Solver_vect}_{sys1}$ pour des systèmes Sys_1

Nous nous intéressons dans cette section à l'efficacité des composants SOLVER_{sys2} dans le cas où toutes les contraintes sont des égalités. Dans ce cas, les systèmes d'équations sont de type SYS_1. Nous effectuons alors une comparaison entre les solveurs SOLVER_{sys2} et

SOLVER_VECT$_{sys1}$ afin de voir lequel de ces deux types de composants est le plus efficace dans ce cas particulier.

Le tableau 4.1 donne les résultats obtenus pour un solveur traitant 9 duplications d'un signal de 32 bits. Le composant de référence de type SOLVER_VECT$_{sys1}$ se nomme SOLVER_VECT$_{sys19/32}$. Il est construit sur l'ancienne architecture qui était utilisée jusqu'à maintenant et peut être utilisé "quelque soit" le contexte. Les autres composants concernent des solveurs construit selon la méthode présentée dans ce document. Ils sont dépendants du contexte et on trouve donc différents solveurs : SOLVER$_{sys29/32}$_eqJ est un solver pour 9 signaux de 32 bits, avec J opérations arithmétiques identiques.

	LCs	**FFs**	**Chemin critique (ns)**
SOLVER_VECT$_{sys19/32}$	479	0	40
SOLVER$_{sys29/32}$	61	0	16
SOLVER$_{sys29/32}$_eq2	54	0	16.920
SOLVER$_{sys29/32}$_eq3	35	0	13.910
SOLVER$_{sys29/32}$_eq4	30	0	14.762
SOLVER$_{sys29/32}$_eq5	24	0	14.966
SOLVER$_{sys29/32}$_eq6	13	0	14.806
SOLVER$_{sys29/32}$_eq7	11	0	19.904
SOLVER$_{sys29/32}$_eq8	9	0	16.242

Table 4.1 – Résultats en synthèse pour un composant de résolution gérant 9 duplications de 32 bits

Le nouveau solveur SOLVER$_{sys29/32}$ est bien plus efficace que SOLVER_VECT$_{sys19/32}$. La surface est divisée par 7 environ, alors que le chemin critique est divisé par 2. Cette nouvelle architecture est donc bien plus efficace que l'ancienne. Plus il y a d'opérations arithmétiques identiques, plus le solver est simple, ce qui est naturel puisque la résolution est simplifiée.

En contrepartie, le temps de production des solveurs de type SOLVER$_{sys2}$ peut être limitant pour un nombre de duplications élevé, ce qui justifie la conservation du composant SOLVER_VECT$_{sys1}$.

4.4.5 Analyse de couverture d'un ensemble de propriétés

Dans [Cla07], Claessen détaille une analyse statique permettant de connaître si un signal peut-être non contraint à un instant donné. Si c'est le cas, alors la spécification peut présenter un manque de précision. Il se restreint dans son étude aux propriétés de sûreté. Grâce à notre approche, nous pouvons effectuer de manière dynamique le même type d'analyse, sur toute propriété de PSL$_{ss}$.

Il suffit d'ajouter un module Behav_Free au modèle de l'environnement. Un module *Behav_Free* est utilisé pour chaque signal de la spécification. Celui-ci prend en entrée l'ensemble des signaux de contrainte et fournit une sortie *Free*. S'il existe un cycle où tous les signaux de contrainte pour un signal, sont inactifs, alors le signal *Free* est passé à '1', indiquant un manque de précision dans la spécification.

Ceci ne constitue pas une erreur, puisqu'il se peut que dans certains cas, un signal ne soit fixé à aucune valeur particulière. Mais le signal *Free* fournit une information impor-

tante à l'utilisateur en l'informant que des signaux peuvent prendre des valeurs aléatoires sous certaines conditions.

Il est aussi possible d'utiliser un model-checker pour vérifier que tous les signaux sont toujours correctement spécifiés. La propriété consiste alors à vérifier qu'aucun des signaux *Free* ne sont jamais actifs.

4.5 Bilan

Le SOLVER_{sys2} permet de résoudre les valeurs pour des signaux dont les valeurs sont contraintes par des comparaisons arithmétiques. Contrairement aux solveurs classiques (SOLVER_BIT_{sys1}, $\text{SOLVER_VECT}_{sys1}$), ils ne sont pas utilisables quelque soit les signaux dupliqués. Une analyse statique des propriétés est effectuée et le solveur SOLVER_{sys2} est construit pour un ensemble de duplications donné.

L'analyse statique est basée sur une exploration de toutes les parties d'un ensemble pour déterminer les blocs aléatoires à utiliser. Pour un ensemble de N duplications, 2^N parties doivent être analysées. Le temps de l'algorithme peut donc exploser. Théoriquement le nombre de composants peut lui aussi augmenter très rapidement avec le nombre de duplications. Nous avons montré qu'en général, ceci ne se produit pas car les opérateurs d'égalité réduisent considérablement le nombre d'ensembles solutions, donc de blocs aléatoires.

Enfin, dans le cas d'un ensemble de duplications réduit à des opérateurs d'égalité seulement, il est possible d'utiliser le composant SOLVER_{sys2} au lieu du composant $\text{SOLVER_VECT}_{sys1}$ décrit précédemment. Le circuit résultant est bien plus efficace. Néanmoins, le temps de l'analyse statique peut toujours être un facteur pénalisant.

Notre approche ne prend pas en compte les opérateurs arithmétiques de calcul tels que l'addition, la soustraction etc. Des travaux futurs devront porter sur ce point.

La modélisation de l'environnement à l'aide d'hypothèses s'effectue en deux étapes :

- création d'un générateur pour chaque hypothèse.
- assemblage des générateurs à l'aide de composants de résolution (si nécessaire).

Une fois cela effectué, l'utilisateur n'a plus qu'à lancer les générateurs aux instants voulus afin de fournir des vecteurs de test appropriés. Une automatisation est possible afin de construire des scénarios complexes permettant un test en profondeur et une couverture maximale des fonctionnalités critiques.

Chapitre 5

Circuits corrects par construction

Sommaire

5.1 Introduction **94**

5.2 Annotations de propriétés PSL **95**

5.2.1 Problème 95

5.2.2 Vers une annotation automatique 96

5.3 Synthèse de spécifications **99**

5.3.1 Le générateur-étendu 99

5.3.2 Construction du circuit final 102

5.4 Générateurs-étendus à l'aide de MyGen **103**

5.4.1 Exemple 105

5.5 Résultats expérimentaux **106**

5.5.1 Comparaison SyntHorus/MyGen 106

5.6 Conclusions **108**

5.1 Introduction

Générateurs et composants de résolution permettent de construire un modèle d'environnement à partir d'un ensemble de propriétés décrivant le comportement de celui-ci. Ceci permet un certain degré d'automatisation lors de la production de scénarios complexes utilisés pour la vérification fonctionnelle. Dans ce chapitre, nous proposons d'aller au delà de l'écriture de scénarios afin de construire un circuit complet à partir de sa spécification. Ce circuit n'est plus seulement composé de générateurs activés manuellement par l'utilisateur. L'idée consiste à produire une trace spécifique lorsqu'une séquence de signaux particulière a été reconnue. En remarquant que ce type d'action est à la base de tout contrôleur, il est possible de décrire n'importe quelle partie contrôle à partir de composants spécifiques mélangeant cet aspect reconnaissance de séquences et production de vecteurs particuliers. Ces composants se nomment générateurs-étendus.

L'idée consiste à transformer une spécification temporelle en un circuit correct par construction. Comme le montre la figure 5.1, les avantages sont multiples. D'une part, l'implémentation n'est plus nécessaire puisque la description matérielle est produite automatiquement, et l'étape de vérification fonctionnelle n'a plus lieu d'être puisque le circuit est correct par construction. Le coût de conception est alors fortement diminué.

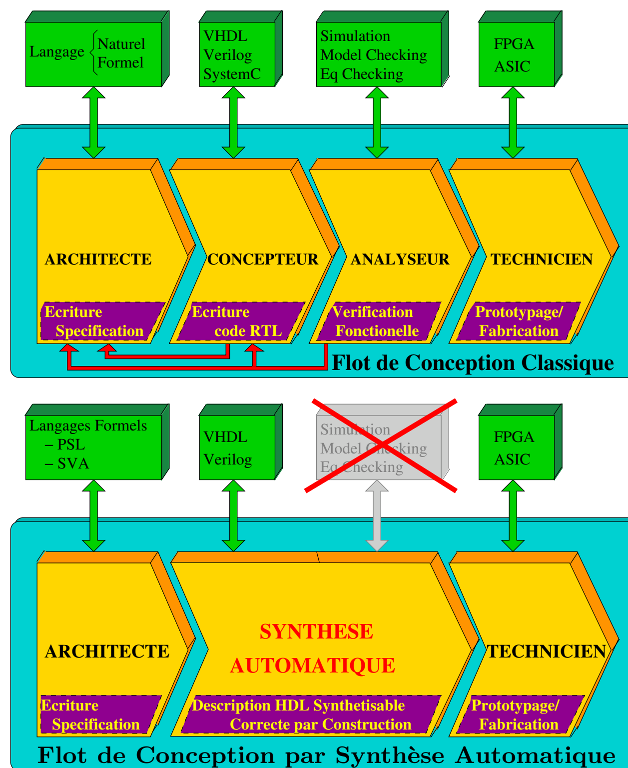

FIGURE 5.1 – Flots de conception classique et supporté par la synthèse automatique de spécifications

Bien qu'il existe des méthodes de complexité linéaire pour synthétiser des spécifications booléennes en circuits, il n'existe pas à notre connaissance de telles approches pour des spécifications temporelles.

Nous proposons une méthode de synthèse de spécification temporelles dont la complexité (temps de production et complexité du circuit résultat) est linéaire en fonction de

la taille de la spécification (plus exactement en fonction du nombre d'opérateurs contenus dans la spécification). Notre approche, appelée SyntHorus, synthétise automatiquement des spécifications temporelles écrites en PSL_{ss} en descriptions HDL correctes par construction. Pour notre approche, assertions et hypothèses sont combinées et transformées en composants fusionnant les caractéristiques des moniteurs et des générateurs : les générateurs-étendus.

Ces composants sont la base de la méthode de synthèse SyntHorus. Ils sont prouvés corrects par rapport à la sémantique de PSL. Combinés entre-eux via des composants de résolution (*cf.* chapitre 4), le circuit résultant possède une fonctionnalité correspondant à la spécification de départ.

Le flot SyntHorus peut traiter des spécifications complexes de plusieurs centaines de propriétés temporelles, et produit en quelques secondes le circuit final correspondant. La taille du circuit est proportionnelle à celle de la spécification.

5.2 Annotations de propriétés PSL

5.2.1 Problème

La spécification d'un circuit met en jeu des signaux à observer (entrées du circuit) et à générer (sorties du circuit). Il est nécessaire de distinguer ces deux types de signaux directement dans la spécification. Pour cela, tout signal *sig* sera annoté de la manière suivante :
- sig_o : signal *sig* de type observé
- sig_g : signal *sig* de type généré

Prenons la spécification Spec_cdt qui est celle du contrôleur CDT présenté chapitre 1, section 1.3 :

- entrées={*Init*, *Req*, *Ack*, *Ressource*}, sorties={*Data*, *Busy*, *Send*}
- property $\mathbf{F0}_{cdt}$ is always($Init_o \rightarrow$ (not($Send_g$) and not($Busy_g$) and$Data_g$="00000000")) ;
- property $\mathbf{F1}_{cdt}$ is always(not($Init_o$) and Req_o)$\rightarrow$ (($Send_g$ and $Data_g$=$Ressource_o$)until Ack_o) ;
- property $\mathbf{F2}_{cdt}$ is always((not($Init_o$) and $Send_o$)$\rightarrow$ next_a[1 to 4]($Busy_g$)) ;

La propriété $F0_{cdt}$ décrit l'état du circuit lors d'une réinitialisation. Si le signal *Init* est actif, toutes les sorties doivent être passées à '0' (*Send*, *Busy* et *Data*). Les deux propriétés suivantes sont utilisées lorsque le circuit n'est pas en réinitialisation.

Propriété $F1_{cdt}$: si le composant CDT reçoit une requête, la valeur *Ressource* est passée sur le port de sortie *Data* jusqu'à ce que le transfert se termine par la réception d'un '1' sur le port *Ack*. Durant tous les cycles que dure le transfert, la sortie *Send* est maintenue active jusqu'à réception du signal *Ack*. Le signal *Send* est utilisé pour informer le composant récepteur que le transfert est en cours.

Propriété $F2_{cdt}$: pour chaque requête, le contrôleur CDT est occupé durant 4 cycles et ne peut donc recevoir d'autres requêtes.

Notre exemple met en jeu uniquement les signaux externes du composant a synthétiser. Il est aussi possible d'utiliser des signaux internes au composant final et de mixer les deux types de signaux.

Un signal annoté “g” dans plusieurs propriétés, ou plusieurs fois dans une même propriété est dit *dupliqué*. Les signaux *Send*, *Busy* et *Data* sont des signaux de Spec_{ctrl} qui sont dupliqués.

Actuellement, l'annotation peut être effectuée manuellement ou de manière partiellement automatique.

5.2.2 Vers une annotation automatique

L'annotation automatique des propriétés s'effectue en trois étapes. Supposons que l'annotation de la spécification S_{annote} suivante doit être effectuée :

- entrées=$\{a, d\}$, sorties=$\{b, c\}$
- property $\mathrm{F1}_{annote}$ is $(a \rightarrow b)$
- property $\mathrm{F2}_{annote}$ is $(d \rightarrow (c$ until $b))$

Annotation étape 1 : Toutes les entrées sont annotées avec “o”. Seule l'annotation des signaux de sortie ou internes pose problème. À la fin de cette étape, l'annotation suivante est obtenue pour S_{annote} :

- entrées=$\{a, d\}$, sorties=$\{b, c\}$
- property $\mathrm{F1}_{annote}$ is $(a_o \rightarrow b)$
- property $\mathrm{F2}_{annote}$ is $(d_o \rightarrow (c$ until $b))$

Il reste à annoter les signaux de sortie b et c.

Annotation étape 2 : Nous définissons le type d'une propriété, assertion ou hypothèse, si elle exprime des contraintes respectivement sur le circuit ou l'environnement. Pour la synthèse automatique, chaque propriété décrivant le circuit doit être de type assertion. En effet, si une propriété est de type hypothèse, alors la propriété porte sur l'environnement et ne décrit donc aucun comportement du circuit lui-même. La seconde étape d'annotation consiste alors à analyser chaque propriété et à utiliser un ensemble de règles définissant le type assertion d'une propriété pour annoter les signaux.

Pour cela, nous utilisons l'ensemble de règles défini par Jin *et al.* dans [JNZ08]. Celui-ci permet de définir le type d'une propriété (assertion ou hypothèse) en fonction du type des signaux utilisés : entrées, sorties etc.

Soient F une propriété FL, B une expression booléenne, et S une propriété SERE. Une propriété F est notée F_g (resp. F_o) si tous ses signaux sont générés (resp. observés). Si une propriété F possède des signaux générés et observés, elle est notée $\mathrm{F}_{g \vee o}$. Ce principe s'applique aussi aux propriétés booléennes et SERE.

Pour les propriétés du type “$P1 \rightarrow P2$”, c'est la nature de $P2$ qui défini si la propriété porte sur le circuit ou l'environnement. Supposons que $P2$ porte sur l'environnement. Alors si $P1$ porte aussi sur l'environnement, la propriété exprime que lorsque l'environnement est dans un certain état satisfaisant $P1$, il doit aussi satisfaire les contraintes $P2$. Dans le cas où $P1$ porte sur le circuit, la propriété exprime que lorsque le circuit est dans un certain état, l'environnement doit satisfaire $P2$ et la propriété porte encore une fois sur l'environnement. Ce raisonnement s'étend aux opérateurs $|\rightarrow$ et $|\Rightarrow$.

Pour les opérateurs de la famille until* (until, until!, until_ et until_!), c'est lopérande gauche qui définit le type de la propriété. Supposons que l'opérande droit du until* porte sur l'environnement. Si l'opérande gauche porte sur le circuit, la propriété exprime des contraintes sur le circuit jusqu'à ce que l'environnement satisfasse une certaine condition.

Si l'opérande droit du until* porte sur le circuit, alors les contraintes exprimées par l'opérande gauche seront relachées lorsque le circuit lui-même satisfera une certaine condition. Avec le même raisonnement, on montre que si l'opérande gauche est de type hypothèse, alors la propriété est de type hypothèse. La même règle s'applique pour la famille before* (before, before !, before_, before_ !).

Pour la famille des opérateurs next* (next![k], next[k], next_a[j..k], next_a ![j..k], next_e[j..k], next_e ![j..k]), la définition est évidente. L'opérande définit le type de la propriété. Il en est de même pour never, always et eventually!.

Enfin, nous étendons les règles proposées dans [JNZ08] pour traiter la famille des opérateurs next_event* (next_event[k], next_event[k] !, next_event_a[j..k], next_event_a[j..k] !, next_event_e[j..k] et next_event_e[j..k] !) ainsi que les opérateurs rose et fell). Pour ces derniers, seule l'observation d'un front montant ou descendant d'une expression booléenne n'a de sens. Produire un tel front dans le contexte des circuits synchrones n'aurait aucune utilité. Les opérandes sont donc de type assertion.

C'est l'opérande droit du next_event* qui définit le type de la propriété. L'opérande gauche exprime la condition sous laquelle l'opérande droit doit être satsfait. Si l'opérande droit est de type assertion, alors il y a deux possibilités : l'opérande gauche est de type hypothèse et la propriété exprime que sous certaines conditions de l'environnement le circuit doit vérifier l'opérande droit ; ou alors de type assertion et la propriété exprime que dans un certain état du circuit, celui-ci doit vérifier l'opérande droit.

En appliquant le raisonnement ci-dessus, l'ensemble de règles $\text{Rule}_{Assertion}$ définissant une propriété de type assertion est obtenu :

$\text{Rule}_{Assertion}$: :
- | F_g
- | $\text{B}_{g \vee o} \rightarrow \text{Rule}_{Assertion}$
- | $\text{Rule}_{Assertion}$ until* $\text{B}_{g \vee o}$
- | B_g before* $\text{B}_{g \vee o}$
- | $\{\text{S}_{g \vee o}\} | \Rightarrow \text{Rule}_{Assertion}$
- | $\{\text{S}_{g \vee o}\} | \rightarrow \text{Rule}_{Assertion}$
- | next*($\text{Rule}_{Assertion}$)
- | next_event* ($\text{B}_{g \vee o}$)($\text{Rule}_{Assertion}$)
- | never $\{\text{S}_g\}$
- | never $\{\text{B}_g\}$
- | always $\text{Rule}_{Assertion}$
- | eventually! $\text{Rule}_{Assertion}$
- | rose(B_o)
- | fell(B_o)

Si cet ensemble de règles est appliqué à la spécification S_{annote}, l'annotation suivante est obtenue :
- entrées=$\{a, d\}$, sorties=$\{b, c\}$
- property F1_{annote} is $(a_o \rightarrow b_g)$
- property F2_{annote} is $(d_o \rightarrow (c_g$ until $b))$

Cette étape a permis d'annoter le signal b de la première propriété, ainsi que le signal c de la propriété F2. Si b avait été annoté "o", la propriété aurait été une hypothèse et n'aurait pas décrit un aspect du comportement du circuit à synthétiser.

Annotation étape 3 : La troisième étape d'annotation analyse chaque propriété et annote les signaux grâce à des règles supplémentaires définies dans le cadre des générateurs-étendus.

Seul un opérande binaire peut posséder à la fois un signal observé et généré. Ainsi, seuls les opérandes binaires de PSL possèdent des générateurs-étendus primitifs correspondants. Pour un opérateur binaire OPb de PSL correspond deux générateurs-étendus où les opérandes gauche et droit sont respectivement (observés, générés) ou (générés, observés). Dans le premier cas, le Extended-generator sera noté OPb_{og} et dans le second cas OPb_{go}.

Le sous-ensemble simple de PSL définit des règles sur l'écriture des propriétés pour que la vérification dynamique soit possible. Pour cela, la contrainte consiste à ce que la vérification d'une propriété à un cycle t dépende uniquement de l'état de la propriété aux cycles $j \leq t$. Ceci se traduit de la manière suivante pour les :

- moniteurs : l'état de vérification d'un moniteur au cycle t ne dépend que des signaux observés à l'instant courant et aux instants précédents. Un moniteur peut être créé à partir de n'importe quelle propriété de PSL_{ss}.
- générateurs : un générateur est conçu pour respecter la propriété dés le cycle de son activation. Celui-ci ne viole donc jamais la propriété quel que soit le cycle t où l'on se place. Un générateur peut donc être produit à partir de n'importe quelle propriété PSL de type LTL.

A partir des considérations ci-dessus, nous obtenons la règle suivante pour les générateurs-étendus : il est nécessaire et suffisant que l'état de validité du générateur-étendu à un cycle t ne dépende que de l'état des signaux observés aux cycles $j \leq t$. Aucune restriction n'est appliquée pour les signaux produits puisque, sous cette condition, les signaux générés produiront toujours des traces valides.

Nous imposons donc la restriction suivante aux propriétés PSL utilisées dans le cadre de la synthèse de spécifications : *tout opérande observé d'un générateur-étendu doit être booléen.* Cette limitation est simplement un ajustement du sous ensemble simple de PSL dans le contexte des générateurs-étendus.

Afin de clarifier le problème qui se pose lors de l'observation d'un opérande FL, prenons l'exemple suivant :

$$\mathsf{property}\ \text{P2}\ \mathsf{is}\ (sig_o \rightarrow (\mathsf{next!}[4]A_o)\ \mathsf{until}\ B_g)$$

Supposons que *sig* est actif au cycle 0 et que la production du signal B='1' est prévue au cycle *t*. Pour tous les cycles j<t, la sous-propriété $\mathsf{next!}[4]A_o$ doit être vérifiée. La vérification se terminera au cycle *t+3*. Il est donc impossible de garantir au cycle t que le générateur-étendu satisfait la propriété correspondante puisque nous n'avons aucune garantie que l'environnement produira le signal A à $t+3$.

L'observation d'une propriété FL qui requiert une quelconque connaissance du futur pour déterminer la production de signaux au cycle courant est contraire aux principes du sous-ensemble simple de PSL. La restriction présentée ici applique simplement les principes de PSL_{ss} dans le nouveau contexte des générateurs-étendus qui combinent les concepts d'observation et de génération de signaux.

Il découle de cette restriction que tous les opérateurs binaires doivent avoir au moins un opérande booléen (celui qui est observé). Ceci supprime toute ambiguïté potentielle concernant le choix d'un générateur-étendu primitif (OPb_{og} ou OPb_{go}) pour l'opérateur binaire correspondant.

Dans le cas général, il est par exemple possible d'utiliser until_{go} puisque l'opérande droit est booléen, mais pas until_{og} puisque l'opérande gauche peut être de type FL. Si ce dernier est de type booléen, alors les deux versions peuvent-être utilisées. L'utilisateur devra explicitement spécifier quelle version du générateur-étendu employé. Par défaut, la version où l'opérande booléen est observée est sélectionnée par l'outil.

L'annotation suivante est obtenue pour la spécification S_{annote} :

- entrées=$\{a, d\}$, sorties=$\{b, c\}$
- property $\mathrm{F1}_{annote}$ is $(a_o \rightarrow b)$
- property $\mathrm{F2}_{annote}$ is $(d_o \rightarrow (c_g \text{ until } b_o))$

Dans ce cas précis, l'annotation est complète, mais il existe des cas pour lesquels l'annotation ne peut être finalisée et quelques signaux doivent être annotés à la main. Un tel exemple d'annotation partielle est détaillé dans la section 7.3. Cet exemple montre que le taux d'annotation est élevé, plus de 80% des signaux ont pu être annotés dans les exemples utilisés au cours des travaux rapportés ici (en appliquant l'hypothèse minimisant les comportements aléatoires). Une analyse plus poussée doit être menée afin d'améliorer le processus d'annotation automatique et tendre vers une annotation complète dans tous les cas.

5.3 Synthèse de spécifications

Toute synthèse de spécification doit être précédée d'une étape consistant à vérifier si la spécification est cohérente. Notre approche utilise RAT pour analyser la cohérence de la spécification. Des hypothèses peuvent être ajoutées pour rendre la spécification réalisable.

Les méthodes décrites dans l'état de l'art sont principalement fondées sur une analyse statique de la spécification afin de modéliser (de différentes manières) les comportements du circuit sous toutes les actions possibles de l'environnement. Ces solutions sont codées directement dans la description matérielle.

Nous prenons une approche du problème complètement différente puisque les circuits produits par SyntHorus possèdent des circuits spécifiques ($\mathrm{SOLVER_BIT}_{sys1}$, $\mathrm{SOLVER_VECT}_{sys1}$ et SOLVER_{sys2}) calculant les signaux de sortie à la volée.

5.3.1 Le générateur-étendu

Un générateur-étendu résulte de la combinaison d'un moniteur et d'un générateur. L'idée consiste à produire une séquence de signaux prédéfinie (partie générateur) lorsqu'une séquence spéciale est reconnue (partie moniteur).

5.3.1.1 Générateur-étendu primitif

Les générateurs-étendus primitifs possèdent une interface générique sur le même modèle que les moniteurs (*cf.* figure 5.2). La seule différence est que le port d'entrée *Expr* du moniteur n'est pas présent sur les générateurs-étendus primitifs. Ces derniers sont tous de type connecteur.

Ils possèdent une architecture sur le même modèle que les moniteurs. Leur code source est globalement plus simple d'une part car l'observation de signaux supprime l'utilisation du bloc ALEA et d'autre part car le bloc SEM est lui aussi simplifié. La génération est en effet contrôlée directement par le *Start* et le signal observé.

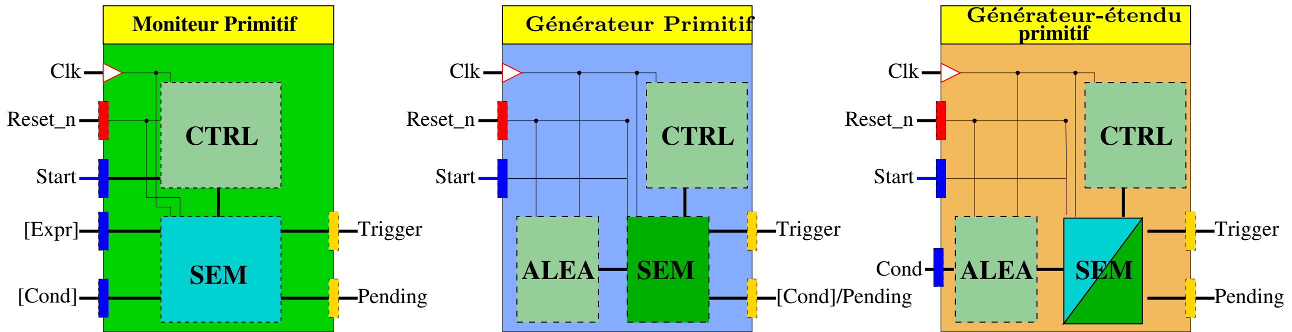

FIGURE 5.2 – Architectures des moniteurs, générateurs et générateurs-étendus

Comme pour les moniteurs et les générateurs, le générateur-étendu possède une interface générique comprenant deux signaux de synchronisation *Clk* et *Reset_n*, un port d'entrée *Cond* pour le signal observé et un port de sortie *Trigger* pour le signal produit. La figure 5.2 illustre ceci.

Prenons le générateur-étendu primitif until_{go}. Dés qu'il est activé, l'opérande est passé à '1' (port *Trigger* actif). Il est maintenu à '1' jusqu'à ce que l'opérande droit soit actif (port *Cond* actif).

5.3.1.2 Générateur-étendu complexe

Alors qu'un générateur-étendu primitif correspond à un *opérateur binaire de* PSL, un générateur-étendu complexe correspond à une *propriété* PSL. Il est composé de trois types de composants primitifs : moniteurs, générateurs et générateurs-étendus.

Cette caractéristique pose un problème quant à l'utilisation du composant primitif approprié, et justifie le besoin d'annoter les propriétés (*cf.* section 5.2). La propriété (A until B) peut être synthétisée en moniteur (A et B observés), en générateur (A et B générés) ou en un générateur-étendu (A généré, B observé). Toutes ces ambiguïtés sont résolues en annotant les propriétés avant de construire le générateur-étendu complexe.

Un générateur-étendu complexe possède une interface générique prenant en entrée les signaux de synchronisation (*Clk* et *Reset_n*) et les signaux observés. Il produit en sortie les signaux à générer ainsi que leurs signaux de contrainte *Pending* associés.

La construction d'un générateur-étendu s'effectue en deux étapes. Premièrement, l'arbre syntaxique est analysé, et pour chaque opérateur, le type de composant à utiliser est calculé (moniteur, générateur ou générateur-étendu). Ensuite, ceux-ci sont interconnectés entre-eux en suivant l'arbre syntaxique de la propriété.

Etape 1 - Sélection des composants primitifs : Le calcul des types pour chaque composant élémentaire est effectué de manière récursive sur l'arbre syntaxique de la propriété PSL. Un composant ayant des opérandes uniquement de type générateurs, ou générateurs-étendus sera un générateur. S'il possède un opérande de type générateur et un de type moniteur alors ce sera un générateur-étendu. Enfin si les opérandes sont uniquement observés, le composant sera un moniteur. L'algorithme `DEF_TYPE` 3 effectue le calcul des types.

Chaque noeud n peut avoir au plus 2 fils notés $n.left$ et $n.right$, correspondants aux opérandes gauche et droit de l'opérateur courant. La notation $\mathrm{T}(n)$ donne le type du noeud

Algorithme 3 Sélection des composants primitifs

```
1:  DEF_TYPE(node : n)
2:  if n.left !=null then
3:     DEF_TYPE(n.left)
4:  end if
5:  if n.right !=null then
6:     DEF_TYPE(n.right)
7:  end if
8:  if Is_Binary(n) then
9:     if T(n.left)=mon and T(n.right)=mon then
10:       T(n)←mon
11:    else if (T(n.left)=mon && T(n.right)=gen) || (T(n.left)=gen && T(n.right)=mon)
       then
12:       T(n)←ext_gen
13:    else
14:       T(n)←gen
15:    end if
16: else if Is_Unary(n) then
17:    if T(n.left)=mon then
18:       T(n)←mon
19:    else
20:       T(n)←gen
21:    end if
22: else if Is_leaf(n) then
23:    if Observed(n) then
24:       T(n)←mon
25:    else
26:       T(n)←gen
27:    end if
28: end if
```

n. Si ce noeud ne possède qu'un fils, alors $n.right$ est nul. Chaque noeud peut avoir 3 types différents. Pour un signal observé ou un moniteur, T(n)=mon. Pour un signal généré, ou un générateur, T(n)=gen. Finalement pour un générateur-étendu, T(n)=ext_gen.

Les feuilles correspondent aux signaux observés et générés. Les noeuds restants correspondent à des opérateurs PSL. Le type des feuilles est obtenus directement en analysant le type du signal : observé ou généré (lignes 24 et 26 de l'algorithme `DEF_TYPE` 3).

Ensuite, l'algorithme `DEF_TYPE` calcule récursivement le type de chaque noeud en utilisant les types de noeuds fils. Pour cela, les cinq règles suivantes sont utilisées :

- (T($n.left$)=mon,T($n.right$=null)) :T(n)=mon (ligne 18 algorithme 3)
- (T($n.left$)=gen,T($n.right$=null)) :T(n)=gen (ligne 20)
- (T($n.left$)=mon,T($n.right$)=mon) :T(n)=mon (ligne 10)
- (T($n.left$)=gen,T($n.right$)=gen) :T(n)=gen (ligne 14)
- (T($n.left$)=mon,T($n.right$)=gen) :T(n)=ext_gen (ligne 12)
- (T($n.left$)=gen,T($n.right$)=mon) :T(n)=ext_gen (ligne 12)
- autres cas : T(n)=ext_gen (ligne 14)

Etape 2 - Interconnexion : Le schéma d'interconnexion est identique à celui utilisé pour les générateurs. Le générateur-étendu pour la propriété $F1_{cdt}$ est donné figure 5.3.

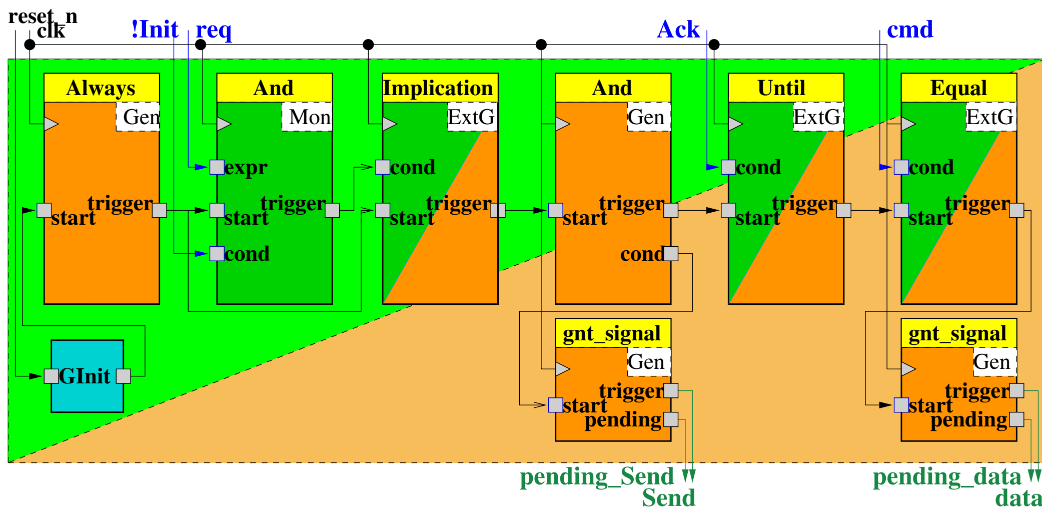

FIGURE 5.3 – Architecture du générateur-étendu complexe pour $F1_{cdt}$

Il suffit alors de construire les générateurs-étendus pour chaque propriété et de les assembler afin d'obtenir la description matérielle du circuit final.

5.3.2 Construction du circuit final

Un générateur-étendu est produit pour chaque propriété de la spécification. L'obtention du circuit final s'effectue en connectant les générateurs-étendus ayant des signaux générés en commun à des composants de résolution : les solvers(*Cf.* section 4.1.1).

Il y en a un par signal dupliqué. La figure 5.4 illustre ceci. Les propriétés $F0_{cdt}$ et $F1_{cdt}$ génèrent le signal *Send* et sont ainsi connectées au SOLVER_BIT$_{sys1}$ Solver$_{send}$ qui produit la valeur finale pour *Send*.

Une fois tous les générateurs-étendus connectés aux solvers éventuels, il suffit d'encapsuler l'ensemble de circuits obtenu dans une entité top-level. Celle-ci prend en entrée les signaux de synchronisation *Clk* et *Reset_n* ainsi que les signaux observés. Elle fournit en sortie les signaux à générer ainsi que les signaux *Err* des composants de résolution dans le cas où la cohérence de la spécification n'a pas été vérifiée. La figure 5.4 illustre la structure du circuit Spec_{ctrl} synthétisé automatiquement par SyntHorus.

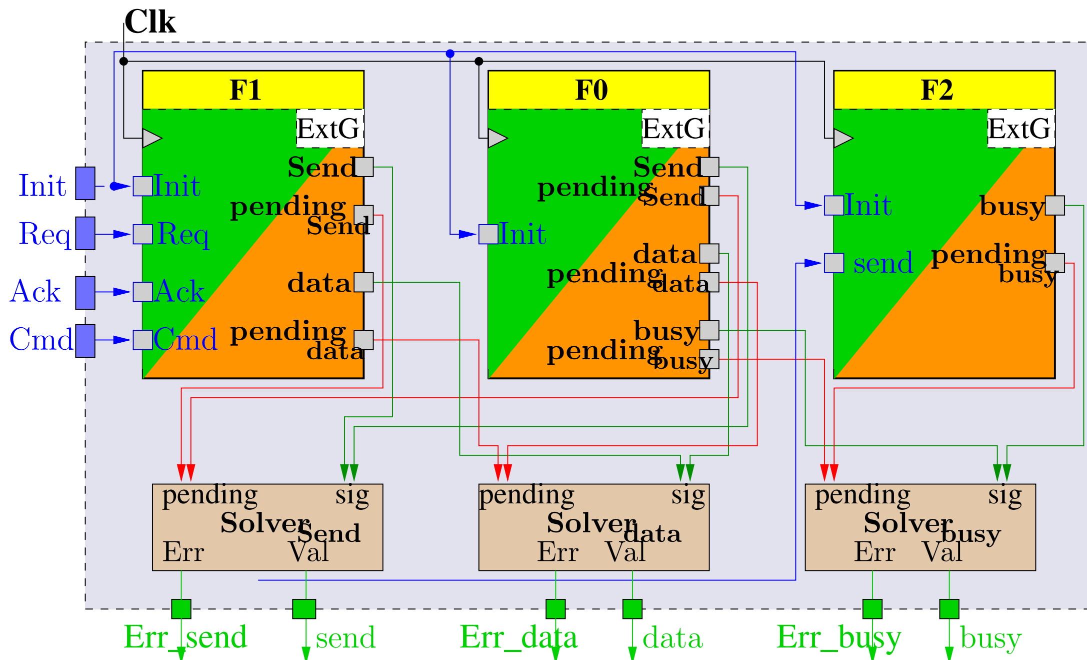

FIGURE 5.4 – Résultat de la synthèse de la spécification Spec_cdt : Le contrôleur CDT

5.4 Générateurs-étendus à l'aide de MyGen

L'idée consiste à adapter l'outil MyGen pour la production de générateurs-étendus. Que ce soit MBAC ou MyGen, les deux outils travaillent sur des modèles à base d'automates afin de synthétiser les propriétés temporelles en moniteurs ou générateurs.

Dans le cas des moniteurs, toutes les étiquettes associées aux transitions sont des expressions booléennes ne contenant que des signaux observés. Une transition d'un état à l'autre s'effectue si la combinaison des entrées à l'instant considéré satisfait l'expression booléenne associée à la transition courante.

Pour les générateurs, les expressions booléennes ne contiennent que des signaux produits. Le fonctionnement relatif aux transitions est plus complexe car :

- à chaque cycle une transition doit être prise sous peine de violer la propriété associée.
- une combinaison de signaux satisfaisant la condition de transition doit être calculée à la volée afin d'emprunter la transition de manière correcte.
- si plusieurs transitions peuvent être prises, un choix aléatoire doit être effectué afin de décider équitablement quelle(s) transition(s) prendre au cycle courant.

Nous proposons ici une approche pour synthétiser un générateur étendu à l'aide de MyGen. Dans ce cas, les conditions de transitions sont des expressions booléennes contenant à la fois des signaux d'entrée et de sortie. L'idée consiste à mettre cette expression booléenne sous forme normale disjonctive et à analyser les clauses ainsi obtenues une par une.

Prenons la propriété A3b(i) suivante (dérivée de la propriété A2b(i) utilisée section 2.1.1.2) :

property A3b(i) is assert always($\{ask(i)_g$;(not $grant(i)_o$ or not $ask(i)_g$)}
$|\rightarrow$\{true ;not $use(i)_g$\}) ;

La propriété A3b(i) stipule : si une unité demande un accès à la ressource au cycle courant et qu'elle n'obtient pas de jeton ou stoppe sa demande au cycle suivant, alors elle n'utilise pas la ressource deux cycles plus tard.

La figure 5.5 illustre l'automate obtenu pour la propriété A2b(i) dans le cas d'un moniteur, d'un générateur et d'un générateur-étendu. Les transition en pointillés sont empruntées si la condition de transition est observée, celles avec un trait plein représentent les conditions générées et celles avec un trait triple les conditions combinant signaux observés et générés. Pour l'automate du générateur-étendu, si le signal $ask(i)$ est produit, et qu'au cycle suivant le signal $grant(i)$ est inactif, ou la demande a été remise à zéro (not $ask(i)$), alors l'unité n'utilise pas la ressource au cycle suivant (signal $use(i)$ à '0').

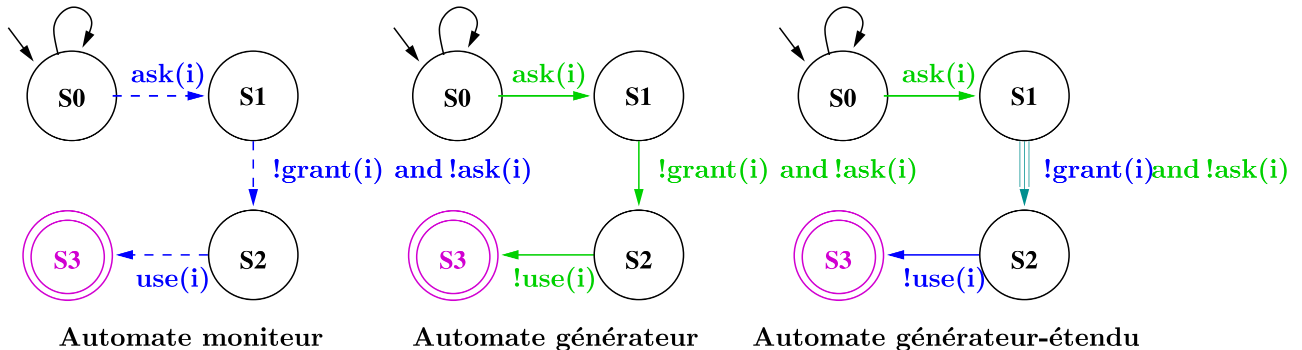

FIGURE 5.5 – Automate obtenu pour la propriété A2b(i)

5.4.0.1 Condition booléenne complexe

Considérons un ensemble de transitions allant d'un état S0 à un état S1. De même que pour les générateurs, le bloc aléatoire sélectionne une transition T. La condition associée à T contient des signaux observés et générés. La méthode appliquée pour prendre une telle transition hybride est décrite dans cette section.

La propriété booléenne B associée à la condition T est mise sous forme normale disjonctive. C'est la première étape effectuée par l'outil MyGen pour traiter les générateurs-étendus. La seconde étape consiste à produire le code VHDL.

Notons a_o^{ij} un signal à observer et b_g^{ij} un signal à générer. Une fois mise sous forme normale disjonctive, l'expression booléenne B de la condition de transition a la forme

suivante :

$$\begin{cases} B = Clause(1) \vee ... \vee Clause(n), n \geqslant 1 \\ Clause(i) = (a_o^{i_1} \wedge ... \wedge a_o^{i_N}) \wedge (b_g^{i_1} \wedge ... \wedge b_g^{i_P}) \end{cases} \tag{5.1}$$

Le traitement d'une telle condition s'effectue clause par clause. Nous distinguons trois types de clauses :

- *clause observée* : tous les signaux sont observés.
- *clause générée* : tous les signaux sont générés.
- *clause hybride* : signaux observés et générés.

Une fois la propriété sous forme clausale, le principe de prise de transitions est simple. Pour une clause observée, il suffit d'analyser la valeur de l'expression booléenne au cycle présent. Si elle est vérifiée, la transition est prise sans aucune analyse supplémentaire des autres clauses. Le principe est donc complètement identique à celui utilisé par les moniteurs.

Pour une clause hybride, si la partie observée est vérifiée, alors il y a production de la partie générée pour satisfaire la condition de transition.

Pour une clause générée, une combinaison valide est calculée pour satisfaire l'expression booléenne. Le principe est le même que pour les générateurs.

5.4.0.2 Couverture de l'espace de traces valides

Il faut toujours s'assurer qu'au moins une transition est prise à chaque cycle d'horloge. Si ce n'est pas le cas, un jeton est perdu et ceci signifie une violation de la propriété !

Dans le cas d'un état contenant uniquement des transitions observées, aucun contrôle n'est possible et on se retrouve avec le même problème d'incohérence que celui décrit dans le chapitre 4. Celui-ci est résolu en appliquant à l'ensemble de propriétés une analyse statique à l'aide de RAT. Ceci supprime tout problème éventuel de cohérence.

Dans le cas où aucune analyse de la spécification n'a été effectuée, la détection d'incohérence au niveau de la propriété (plusieurs instances d'un même signal au sein d'une même propriété) est très simple. Si aucune transition n'est prise, une sortie d'erreur passe à '1'. Chaque état est donc équipé d'une variable indiquant si aucune transition n'a été prise durant un cycle. Ceci permet la détection d'incohérences.

Si au moins une transition observée est prise, il n'est pas nécessaire de regarder les autres transitions. Par contre, si aucune transition observée n'est prise, le générateur-étendu analyse chaque clause mixte. Si la partie observée de l'une d'elle est vérifiée, la transition est prise (une combinaison valide pour les signaux générés de cette clause est calculée) et aucune analyse supplémentaire n'est effectuée.

Enfin, dans le cas où aucune clause observée ou mixte n'est prise, alors au moins une transition générée doit être prise.

5.4.1 Exemple

Prenons l'exemple d'une transition entre un état S0 et S1 (notés STATE(0) et STATE(1) dans la description VHDL) possédant la condition de transition Trans1 suivante :

$$\text{Trans1}=(\text{a1}_o \wedge \text{a2}_o) \vee (\text{a1}_o \wedge \text{b1}_g) \vee (\text{b1}_g \vee \text{b2}_g)$$

```
1  if Trans1='1' then
2      no_trans:='1';
3      if a1='1' and a2='1' then
4          no_trans:='0';
5      elsif a1='1' then
6          no_trans:='0';
7          b1<='1';
8      end if;
9      if no_trans='1' and rand_nb(0)='0' and rand_nb(1)='1' then
10         b1<='0';
11         b2<='1';
12     elsif no_trans='1' and rand_nb(0)='0' and rand_nb(1)='1' then
13         b1<='1';
14         b2<='0';
15     elsif no_trans='1' then
16         b1<='1';
17         b2<='1';
18     end if;
19     STATE(0)<='0';
20     STATE(1)<='1';
21 end if;
```

FIGURE 5.6 – Code VHDL produit pour la condition Trans1

La figure 5.6 illustre le code VHDL pour le traitement de la transition possèdant la condition Trans1 et allant de l'état S0 vers S1. D'autres transitions peuvent être présentes dans un état et sont toutes traitées en parallèle, de la même manière que pour les générateurs. La variable `no_trans` vaut '1' si aucune clause n'a permis de prendre la transition.

Le code est séquentiel. Les conditions observées sont analysées en premier (lignes 3 à 7). Si les signaux a1 et a2 sont tous deux actifs, alors la première clause de Trans1 est vérifiée et la transition vers S1 est prise. Si ce n'est pas le cas, la clause mixte est analysée (lignes 8 à 13). Si a1 set actif alors le signal est placé à '1' pour satisfaire la clause. Le jeton est déplacé dans S1.

Enfin, si aucune des transitions observées et mixtes n'a été prise, alors la clause générée doit être validée pour prendre la transition (lignes 14 à 23). Ceci est indiquée dans la description VHDL puisque la variable `no_trans` vaut '1'. La dernière clause est alors validée en activant au moins un des deux signaux b1 et b2.

5.5 Résultats expérimentaux

5.5.1 Comparaison SyntHorus/MyGen

L'ensemble des propriétés de l'annexe B.1[1] a été utilisé pour évaluer l'efficacité des générateurs-étendus produits par SyntHorus et MyGen. La figure 5.7 regroupe les résultats obtenus.

Comme le montrent les courbes, les circuits produits par SyntHorus sont globalement plus simples que ceux obtenus par MyGen. Ils utilisent généralement moins de ressources sur le FPGA. Les fréquences sont très bonnes dans les deux cas avec un maximum de

[1]Les opérateurs unaires de PSL n'apparaissent pas car ils ne possèdent pas de générateur-étendu.

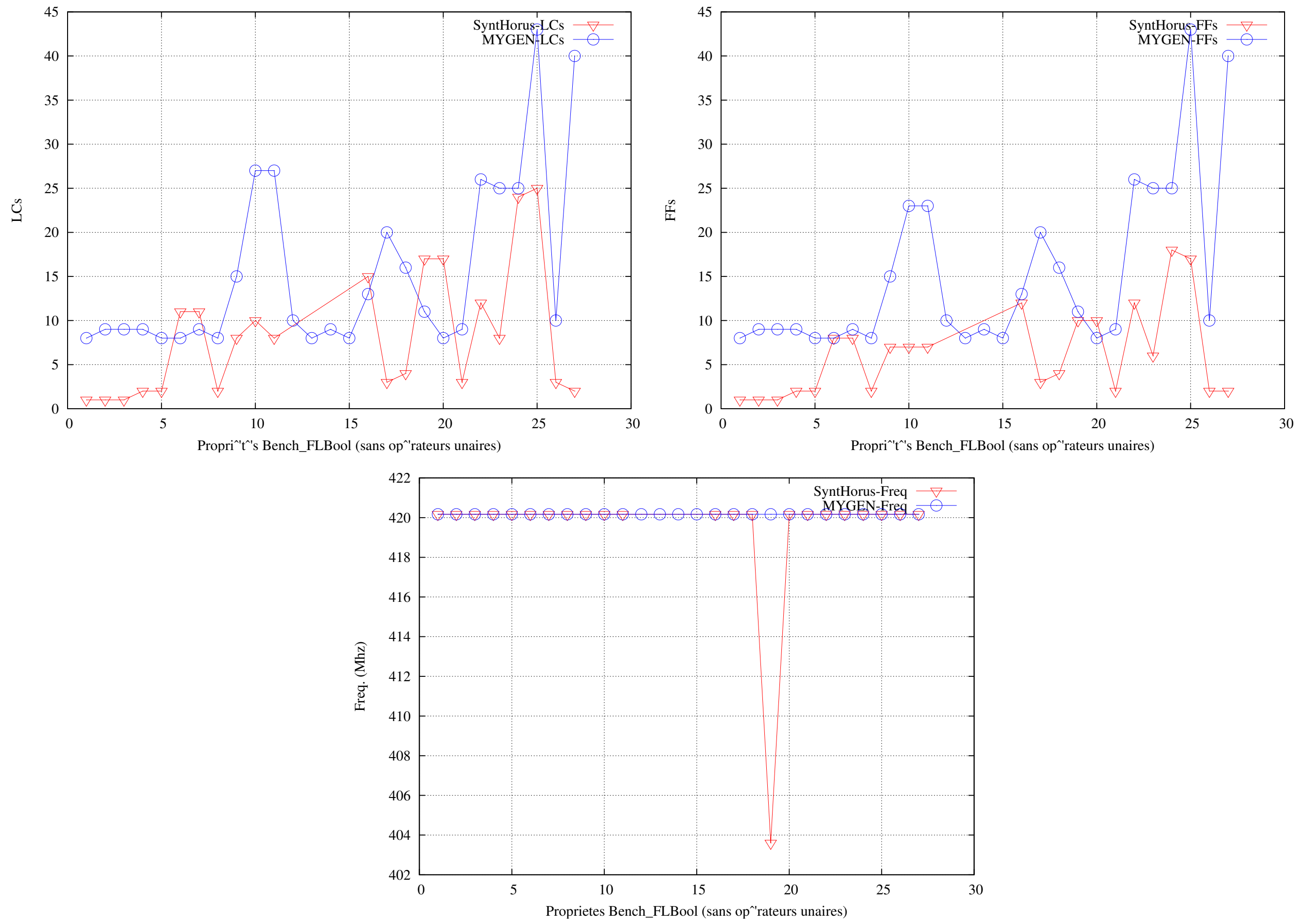

FIGURE 5.7 – Comparaisons des résultats de synthèse Synthorus/MYGEN

420Mhz obtenu pour quasiment toutes les propriétés. Ceci permet la construction de circuits efficaces, aussi bien en terme de surface que de fréquence.

De plus, que ce soit pour les composants primitifs, ou les propriétés complexes, les générateurs-étendus sont plus efficace que les générateurs(*cf.* figure 3.10).

5.6 Conclusions

Une méthode de synthèse automatique à partir de composants issus de l'ABV a été présentée. À notre connaissance, c'est la première méthode de complexité linéaire permettant de synthétiser automatiquement un circuit à partir de spécifications du sous ensemble simple de PSL.

Notre approche est modulaire, encapsulant ainsi la complexité de la spécification dans une hiérarchie de composants vérifiés par preuve de théorème (*cf.* chapitre 6). La preuve des générateurs-étendus primitifs et de la méthode d'interconnexion garantit que le circuit produit par SyntHorus est correct vis-à-vis de la spécification. Actuellement, l'annotation de la spécification n'est pas complètement automatique. Alors que l'annotation des ports d'entrées avec “o” est évidente, les ports de sortie peuvent être annotés “o” ou “g” (*cf.* signal *Send* dans Spec_cdt). Un travail plus approfondi est nécessaire pour compléter l'ensemble des règles d'annotation définies jusqu'à présent.

Deux circuits complexes ont été étudiés et illustrent l'efficacité de l'approche sur des spécifications contenant jusqu'à plusieurs centaines de propriétés temporelles (*cf.* chapitre 7).

Chapitre 6

Preuve des moniteurs, générateurs et générateurs-étendus

Sommaire

6.1 Modélisation en PVS 110

6.1.1 Modélisation des composants de vérification 110

6.2 Sémantique des opérateurs temporels de PSL 112

6.3 Modélisation de la sémantique de PSL en PVS 113

6.4 Modélisation de l'équivalence 113

6.4.1 Définition formelle de $\mathrm{H}(\varphi, t_0, T)$ 114

6.4.2 Définition formelle de $\mathrm{V}(\varphi, t_0, T)$ 114

6.4.3 Définition formelle de $\mathrm{P}(\varphi, t_0, T)$ 115

6.4.4 Définition formelle de $\mathrm{S}(\varphi, t_0, T)$ 115

6.5 Preuve de correction des moniteurs 115

6.5.1 Cas de base 116

6.5.2 Etape d'induction 117

6.6 Preuve des générateurs et générateurs-étendus 119

PSL est un langage possédant une sémantique formellement définie. M. Gordon [Gor03] a joué un rôle prépondérant dans la validation de cette sémantique. Il a embarqué PSL dans l'assistant de preuves HOL, et a utilisé ce système pour démontrer divers théorèmes portant sur la sémantique de PSL. À partir de ces travaux, Tuerk a mis en place une technique permettant de transformer un sous-ensemble non clocké d'expressions FL en propriétés LTL, toujours en utilisant HOL [TS05]. Claessen et Martensson ont proposé dans [CM04] une définition opérationnelle de la sémantique de PSL, guidée par la structure de la propriété PSL. Ils ont pu mettre en évidence des incohérences dans l'interprétation de certains types d'expressions régulières et ce travail fût extrêmement utile afin de définir les bases théoriques permettant de prouver l'équivalence entre la sémantique de PSL et les résultats fournis par l'implémentation des composants de vérification : moniteurs, générateurs et générateurs-étendus.

6.1 Modélisation en PVS

La preuve de la bibliothèque de moniteurs ainsi que de la méthode d'interconnexion a été effectuée par Katell Morin-Allory [MAB06] à l'aide du système PVS [SORSC01]. Celui-ci fournit un environnement supportant la vérification formelle qui comprend : un langage de spécification, un ensemble de théories prédéfinies et un prouveur de théorèmes. Il est basé sur une logique d'ordre supérieur.

La sémantique de PSL s'exprimant dans une logique du second ordre, il était possible de représenter celle-ci directement en PVS. Ajouté à cela que PVS possède de nombreuses stratégies automatiques de preuves, il faisait un candidat idéal pour la preuve des composants de vérification. La preuve se décompose en plusieurs parties :

- modélisation du code VHDL en PVS
- modélisation de la sémantique de PSL
- modélisation de l'équivalence entre les deux

Mes travaux ont consisté à reprendre les preuves effectuées sur l'ancienne bibliothèque de moniteurs et à les adapter pour la nouvelle version de la bibliothèque. Nous verrons plus loin que grâce à la dernière version des moniteurs, la preuve des générateurs et des générateurs-étendus est quasi identique à celle des moniteurs.

6.1.1 Modélisation des composants de vérification

Pour prouver que chaque moniteur est correct vis-à-vis de la sémantique de PSL, la machine d'états finis correspondante est extraite et traduite dans le langage de modélisation PVS. Ceci s'effectue automatiquement grâce à une technique originellement développée pour automatiser la traduction de code HDL en modèles pour ACL2 [TBS04]. L'idée consiste à utiliser une étape de simulation symbolique pour calculer les fonctions de transition et de sortie de la machine à états.

6.1.1.1 Simulation Symbolique

Le simulateur symbolique VSYML [OBMAP09], développé par Florent Ouchet, a été utilisé. Il prend en entrée un circuit séquentiel synchronisé par une horloge globale *Clk*, et fournit en sortie une machine d'états fini où les fonctions de transition et de sortie sont

écrites sous forme conditionnelle normalisée et où l'unité de temps est le cycle d'horloge. Cet outil effectue une stabilisation statique des blocs combinatoires entre chaque cycle d'horloge si aucune boucle combinatoire n'est présente dans le circuit. Le simulateur définit la valeur symbolique d'un signal comme une fonction des valeurs précédentes de tous les signaux susceptibles de modifier la valeur du signal en question.

Le code VHDL de la figure 6.1 contient le code source du moniteur primitif always. Le port *Trigger* déclenche le moniteur primitif suivant. Il est fixé à '1' dès que le moniteur always reçoit un *Start*, et reste actif jusqu'à la fin de la vérification. La valeur du port *Trigger* se calcule de la manière suivante (ligne 1) : soit le port *Start* est actif au cycle courant et *Trigger* est activé, soit le port *Start* a été activé à un cycle antérieur et cette information est enregistrée via le signal *start_t1*. Le signal *start_t1* mémorise le signal *Start* (lignes 7-8). Il est réinitialisé par le signal *Reset_n* (lignes 5-6).

Ceci déclenche une vérification continue de la propriété connectée au moniteur always. La simulation symbolique produit une sortie sous format XML et peut-être alors facilement traduite dans différents langages tels que PVS ou encore ACL2.

```
1  trigger <= start or start_t1;
2  evaluate_start: process (clk)
3  begin
4      if clk'event and clk='1' then
5          if reset_n='0' then
6              start_t1  <= '0';
7          elsif start = '1' then
8              start_t1 <= '1';
9          end if;
10     end if;
11 end process;
```

FIGURE 6.1 – Code VHDL du moniteur always

6.1.1.2 Traduction dans le format PVS

La traduction de XML vers PVS est entièrement automatique et s'effectue à l'aide d'un fichier de style XSLT (Xml Extensible Stylesheet Language) permettant de formater aisément le fichier XML dans la syntaxe PVS. Les objets VHDL sont considérés comme des fonctions récursives en fonction du temps dans le modèle PVS. La figure 6.2 illustre la fonction PVS générée pour la modélisation du calcul du signal *Trigger* pour le moniteur always.

```
1  TRIGGER(t:nat):boolean =
2  (IF t=0
3  THEN FALSE
4  ELSE ( START(t-1) ) OR ( START_T1(t-1) )
5  ENDIF)
```

FIGURE 6.2 – Fonction PVS pour le calcul du signal *Trigger* du moniteur always

6.2 Sémantique des opérateurs temporels de PSL

Cette partie présente la modélisation des opérateurs temporels PSL en PVS. La sémantique des opérateurs booléens est définie comme suit. Soit P un ensemble non vide de propositions atomiques. En pratique, P est l'ensemble des noms des signaux observés ou générés. L'ensemble de toutes les valuations possibles de P est noté 2^P. Soit $\mathcal{B}$ l'ensemble des expressions booléennes de P. L'alphabet Σ est défini comme l'union de 2^P et $\{\top, \bot\}$. Une *lettre* est un élément de Σ.

La *sémantique des expressions booléennes* de $\mathcal{B}$ se définit comme une relation notée $\vdash$ entre Σ et l'ensemble des valeurs booléennes B. On dit que $\ell \in \Sigma$ satisfait *exp* ($\ell \vdash exp$) si et seulement si l'évaluation de *exp* est vraie lorsque les expressions atomiques prennent leurs valeurs dans ℓ.

Exemple 4 : Soit P = {Req, Busy, Ack} un ensemble de propositions atomiques, Σ est défini par $\{ <Req>, <Busy>, <Ack>, <Req, Busy>, <Req, Ack>, <Busy, Ack>, <Req, Busy, Ack>, <> \} \cup \{ \top, \bot \}$. Soit ℓ égal à $<Req, Busy>$, ce qui correspond à Req='1', Busy='1' et Ack='0', ou encore à l'expression booléenne $Req \wedge Busy \wedge \neg Ack$, on a alors :

$$\ell \vdash \text{Req} \qquad \ell \vdash Req \wedge Busy \qquad \ell \vdash \neg Ack$$

Soit $\mathcal{FL}$ l'ensemble des expressions construites à partir des opérateurs FL de PSL. La *sémantique d'une expression* $\varphi \in \mathcal{FL}$ pour un mot $v \in \Sigma$ est donnée sous forme d'une relation de satisfaction $v \Vert\!\equiv \varphi$ (v satisfait φ).

En pratique, un mot correspond à une trace de tous les signaux observés (ou générés) sur un intervalle de temps [t_0, T]. Soit v une trace et $|v|$ sa longueur. Nous notons v^i le i-ème élément de la trace en partant de l'instant 1 et $v^{i..}$ la sous-trace commençant au cycle i. La sémantique des opérateurs PSL est définie sur ces traces.

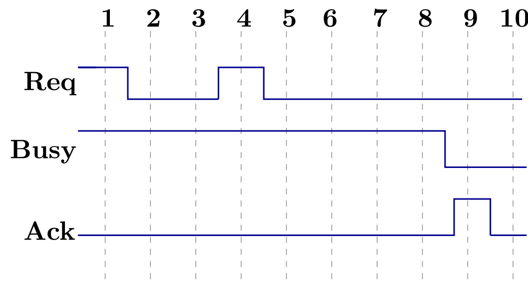

FIGURE 6.3 – Exemple 5 : trace v satisfaisant la propriété $A1_{cdt}$

Exemple 5 : L'alphabet Σ de l'exemple 4 est réutilisé. Nous reprenons la propriété $A1_{cdt}$ définie section 3.2.2 :

$$\text{property } A1_{cdt} \text{ is assert always } (Req \rightarrow (Busy \text{ until! } Ack))$$

La propriété $A1_{cdt}$ permet de vérifier que lorsqu'une requête de transmission de données est reçue par le composant CDT, alors celui-ci passe en mode transmission (signal *Busy* activé) jusqu'à ce que le composant externe termine le transfert en passant le signal *Ack* à '1'.

Soit v la séquence $\{ <Req, Busy>, <Busy>, <Busy>, <Req, Busy>, <Busy>, <Busy>, <Busy>, <Busy>, <Ack>, <> \}$ illustrée figure 6.3. Puisque *Req* est vrai aux cycles $\sharp 1$ et $\sharp 4$, nous avons :

$$v^{1..} \| \equiv Req \qquad\qquad v^{4..} \| \equiv Req$$

De plus, la propriété $A1_{cdt}$ est satisfaite (sur la trace allant des cycles ♯1 à ♯9) car à chaque occurence du signal *Req*, *Busy* vaut '1' jusqu'à ce que le signal *Ack* soit actif (♯9).

$$v \| \equiv \mathsf{always}\ (Req \rightarrow (Busy\ \mathsf{until!}\ Ack)))$$

6.3 Modélisation de la sémantique de PSL en PVS

La sémantique est modélisée grâce à une fonction Sem allant de $\mathcal{FL} \times \mathbb{N} \times \mathbb{N}$ vers l'ensemble $\mathcal{B}$. Soit φ appartenant à $\mathcal{FL}$, alors Sem(φ,t_0,T) est définie de manière inductive sur l'arbre syntaxique de φ et sur l'intervalle de temps [t_0, T]. Pour chaque opérateur Ω, une fonction Sem_Ω implémente la sémantique de Ω et dépend de la fonction globale Sem. Les fonctions Sem et Sem_Ω sont mutuellement dépendantes. Si φ est réduite à une expression booléenne, la fonction Sem(φ,t_0,T) est simplement définie par la valeur de φ à l'instant t_0.

Exemple 6 (Fonction $\mathsf{Sem}_{\mathsf{next_a}}$) : la sémantique des opérateurs faibles de la famille next est :

- $v \| \equiv$ b $\Longleftrightarrow |v|$=0 $\vee$ $v^1 \vdash$ b
- $v \| \equiv$ next(e) $\Longleftrightarrow |v| \leqslant 1 \vee v^{2..} \models$ e
- next[k](e)=$\overbrace{\mathsf{next}\ldots\mathsf{next}(e)}^{k\ times}$
- next_a[i..j](e)=$\forall k \in [i..j]$, next[k](e)
- next_e[i..j](e)=$\exists k \in [i..j]$, next[k](e)

Les opérateurs next[k], next_a et next_e peuvent tous être réécrits à l'aide de l'opérateur next. Leur transformation en PVS nous donne :

$$\mathsf{Sem}_{next}(\varphi, k, t_0, T) = \begin{cases} \mathsf{Sem}(\varphi, t_0, T) & si k = 0 \\ (T - t_0 + 1) \leq 1 \vee \mathsf{Sem}(\varphi, t_0 + 1, T) & si k = 1 \\ \mathsf{Sem}_{next}(\varphi, k - 1, t_0 + 1, T) & si k > 1 \end{cases} \tag{6.1}$$

$$\mathsf{Sem}_{\mathsf{next_a}}(\varphi, i, j, t_0, T) = \forall k \in [i..j], \mathsf{Sem}(\varphi, k, t_0, T) \tag{6.2}$$

$$\mathsf{Sem}_{\mathsf{next_e}}(\varphi, i, j, t_0, T) = \exists k \in [i..j], \mathsf{Sem}(\varphi, k, t_0, T) \tag{6.3}$$

6.4 Modélisation de l'équivalence

Soit M un moniteur implémentant une propriété $\varphi \in \mathcal{FL}$. Prouver l'équivalence entre φ et M revient à prouver l'équivalence entre la sémantique de φ et les signaux $Valid_M$ et $Pending_M$ sur un intervalle [0..T]. Plus précisément, il faut prouver une équation de la forme 6.4.

$$\forall \varphi, \neg Reset_n(0) \Rightarrow (S(\varphi, 1, T) \Leftrightarrow V(\varphi, 1, T) \wedge P(\varphi, 1, T)) \tag{6.4}$$

La fonction S dépend de Sem(φ,t_0,T). Les fonctions V et P dépendent respectivement des signaux $Valid_M$ et $Pending_M$ du moniteur global.

La preuve de l'équation 6.4 s'effectue par récurrence sur la structure de la propriété φ. L'induction va déplacer l'intervalle temporel sur lequel la preuve s'effectue[1] et nous amène à généraliser l'équation 6.4 à un intervalle quelconque [t_0..T] et à renforcer les hypothèses d'induction. Ceci nous donne l'équation 6.5.

$$\forall \varphi, \forall t_0 \in \mathbb{N}, \forall T \geq t_0, H(\varphi, t_0, T) \Rightarrow (S(\varphi, t_0, T) \Leftrightarrow V(\varphi, t_0, T) \wedge P(\varphi, t_0, T)) \quad (6.5)$$

6.4.1 Définition formelle de H(φ, t_0, T)

Sur l'intervalle de vérification [t_0..T], les sorties du moniteur M peuvent être influencées par un *Start* qui aurait été actif avant l'instant t_0. C'est le cas sur l'exemple 6.4. Supposons que l'intervalle de vérification soit [2..7]. Sur cet intervalle, le signal *Valid* doit toujours être actif pour valider la vérification. Or ce n'est pas le cas car celui-ci vaut '0' au cycle ♯3. Ceci est due à l'activation du signal *Start* au cycle ♯1, juste avant l'intervalle de vérification [2..7].

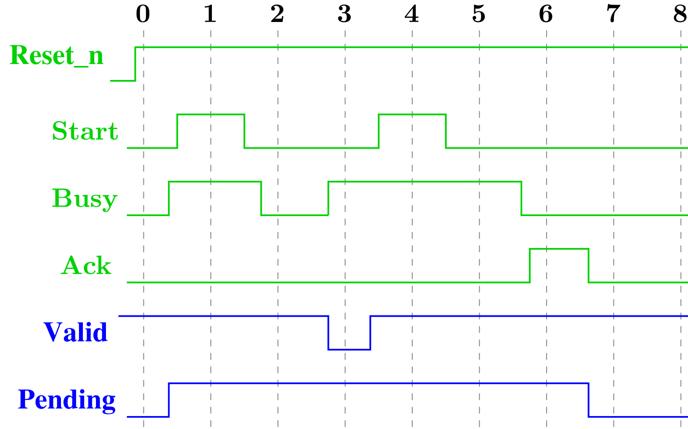

FIGURE 6.4 – Trace pour laquelle Sem((*Busy* until *Ack*),2,7) est fausse.

Il faut donc supposer qu'il n'y a pas eu de *Start* actif à partir du dernier *Reset_n* actif et jusqu'au début de la vérification (instant t_0). Notons t' l'instant du dernier *Reset_n* actif, alors pour tout instant $t > t'$, *Reset_n* et *Start* sont inactifs. De plus, nous supposons que *Reset_n* n'est pas actif sur [t_0..T], sinon le moniteur n'évalue pas la propriété sur [t_0..T] alors que la sémantique serait calculée sur [t_0..T]. Ceci s'exprime par l'équation 6.6.

$$H(\varphi, t_0, T) = \begin{cases} \exists t' \in [0..t_0], \neg Reset_n_M(t') \\ \wedge \forall t_1 \in [t'..T], Reset_n_M(t_1) \\ \wedge \forall t_2 \in [t'+1..t_0-1], \neg Start_M(t_2) \end{cases} \quad (6.6)$$

6.4.2 Définition formelle de V(φ, t_0, T)

Le signal *Valid* est actif par défaut. Si la propriété est vérifiée, alors le signal *Valid* du moniteur global doit toujours rester à '1'. Le signal *Valid* possède un cycle de retard et

[1]Pour une propriété de type alwaysφ, la preuve s'effectue d'abord sur [0..T], puis sur [1..T] etc.

si l'intervalle de vérification est [t_0..T], alors le signal *Valid* sera effectif sur [$t_0 + 1$..T+1]. Ceci justifie l'expression $t + 1$ de $Valid_M$ dans l'équation 6.7 de V :

$$V(\varphi, t_0, T) = \forall t \in [t_0..T],\ Valid_M(t + 1) \quad (6.7)$$

Chaque moniteur possède un seul signal *Valid* fourni par le moniteur primitif le plus en bas à droite de l'arbre syntaxique de la propriété. Ainsi, l'expression $V(\varphi, t_0, T)$ dépend uniquement de la valeur de ce signal $Valid_M$ global, et pas de signaux intermédiaires.

L'exemple 6.4 illustre le fonctionnement de V. Au cycle ♯2, la propriété (*Busy* until *Ack*) est violée, ce qui est reporté au cycle suivant par le moniteur : signal *Valid* à '0' au cycle ♯3.

6.4.3 Définition formelle de P(φ, t_0, T)

Le signal *Pending* est actif lorsque des obligations futures sont en cours de vérification. Si la propriété doit être dans l'état Holds ou Holds Strongly à la fin de la trace (instant T), alors *Pending* doit être inactif à T+1. Ceci est modélisé par la fonction P définie par l'équation 6.8.

Il n'existe aucune contrainte sur le signal *Pending* avant la fin de la vérification puisque selon la propriété il peut être actif ou non. Seule sa valeur à la fin de la vérification est pertinente puisque si celui-ci est actif à T+1, alors la propriété n'a pas été vérifiée. Dans le cas où des opérateurs constituant la propriété sont faibles, la valeur du signal *Pending* est à '0' par défaut.

$$P(\varphi, t_0, T) = \neg Pending(T + 1) \quad (6.8)$$

Sur l'exemple 6.4, la vérification de la propriété s'achève au cycle ♯7 grâce à l'activation du signal *Ack* au cycle précédent. Durant toute la vérification, le signal *Pending* est actif.

6.4.4 Définition formelle de S(φ, t_0, T)

Si un moniteur est activé, alors la propriété φ associée doit être vérifiée. Il est possible que plusieurs *Start* soit actifs sur [t_0..T] à cause de la réentrance potentielle de toute propriété, ce qui justifie l'opérateur ∀ employé dans l'équation de S.

$$S(\varphi, t_0, T) = \forall t \in [t_0..T], Start_M(t) \Rightarrow \mathsf{Sem}(\varphi, t, T) \quad (6.9)$$

Sur l'exemple 6.4, nous avons Sem((*Busy* until *Ack*),1,7) qui est faux, alors que Sem((*Busy* until *Ack*),4,7) est vraie.

6.5 Preuve de correction des moniteurs

Soient Ω_1, ... , Ω_n n opérateurs FL, et $\varphi_n = \Omega_n ... \Omega_1 op_1 ... op_n$ une expression $\mathcal{FL}$ où op_i est la liste des opérandes de Ω_i. L'indice n représente la profondeur de la formule (qui est notée $|\varphi_n|$).

Notons $(M_j)_{j \in \mathbb{N}}$ la suite de moniteurs tels que pour tout j, M_j implémente la propriété φ_j. Autrement dit, pour tout j, M_j est l'interconnexion du moniteur M_{j-1} et du moniteur primitif O_j implémentant l'opérateur Ω_j. Le port d'entrée $Start_M$ correspond au port $Start_n$. L'entrée *Reset_n* est connectée à tous les moniteurs primitifs via le port d'entrée $Reset_n_M$. La figure 6.5 illustre ceci pour la propriété P1 suivante :

property P1 : always $(Req \rightarrow$ next[4]$(Busy$ until $Ack))$;

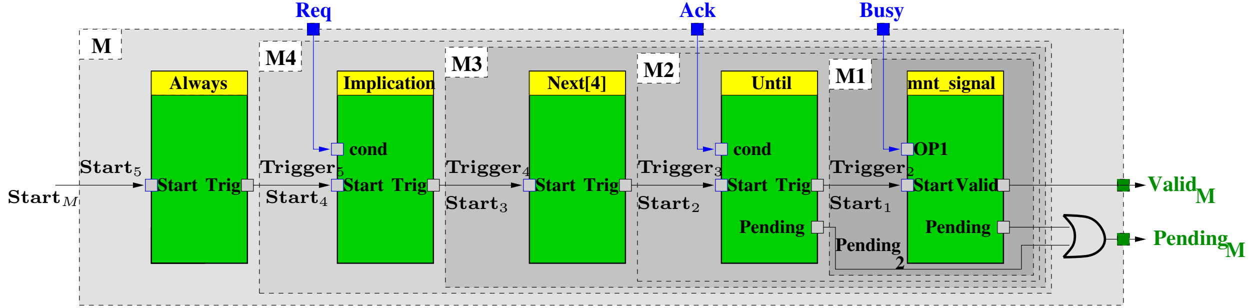

FIGURE 6.5 – Moniteur pour la propriété P1

Les expressions H, S et P de l'équation 6.4 peuvent être réécrites comme le montre l'équation 6.10.

$$\begin{cases} H(\varphi_n, t_0, T) = & \exists t' \in [0..t_0], \neg Reset_n_M(t') \land \forall t_1 \in [t'..T], Reset_n_M(t_1) \\ & \land \forall t_2 \in [t'+1..t_0-1], \neg Start_n(t_2) \\ S(\varphi_n, t_0, T) = & \forall t \in [t_0..T], Start_n(t) \Rightarrow \mathsf{Sem}(\varphi_n, t, T) \\ V(\varphi_n, t_0, T) = & \forall t \in [t_0..T], Valid_1(t+1) \\ P(\varphi_n, t_0, T) = & \forall t \in [t_0..T] P(\varphi_{n-1}, t_0, T) \land \neg Pending_n(T+1) \end{cases} \tag{6.10}$$

Notre hypothèse d'induction est donnée par l'équation 6.11.

$$\begin{array}{l} \forall t_0 \in \mathbb{N}, \forall T \geq t_0, \mathcal{I}(n) = \forall n \in \mathbb{N}, \forall \varphi_n \in \mathcal{FL}, \\ \quad H(\varphi_n, t_0, T) \Rightarrow (S(\varphi_n, t_0, T) \land P(\varphi_n, t_0, T) \land V(\varphi_n, t_0, T)) \end{array} \tag{6.11}$$

6.5.1 Cas de base

Comme il a été dit dans le chapitre 3, les moniteurs primitifs sont divisés en deux catégories : connecteurs et observateurs. La preuve du cas de base concerne les observateurs, alors que la preuve de l'étape d'induction met en jeu les moniteurs de type connecteur.

La preuve du cas de base est effectuée à l'aide de l'équation 6.12 où φ est soit une expression booléenne, soit une propriété FL réduite à un opérateur de type observateur n'ayant pour opérande que des booléens.

$$\begin{array}{l} \forall t_0 \in \mathbb{N}, \forall T \geq t_0, \mathcal{I}(1) = \forall \varphi, H(\varphi, t_0, T) \Rightarrow \\ \quad (S(\varphi, t_0, T) \Leftrightarrow V(\varphi, t_0, T) \land P(\varphi, t_0, T)) \end{array} \tag{6.12}$$

Cette preuve a été effectuée à l'aide PVS pour chaque observateur :mnt_Signal, next_event_e, eventually! et before.

Un théorème PVS est généré pour chaque observateur. La figure 6.6 décrit le théorème produit pour l'opérateur FL next_e.

```
1    FORALL (t0:above(0), T:upfrom(t0)):
2      (EXISTS(t:subrange(0,t0-1)):
3          Not RESET_N_(t)
4          AND (FORALL (t2:subrange(t+1,t0-1)): NOT START_(t2))
5          AND (FORALL (t3:subrange(t+1,T)):RESET_N_(t3))
6      )
7      IMPLIES
8      (
9              (FORALL (t:subrange(t0,T+1)):
10                 START_(t) IMPLIES Nexte(EXPR_,t,T,low_clk,high_clk))
11         IFF
12             ((FORALL (t:subrange(t0,T+1)):
13                 TRIGGER(t) IMPLIES EXPR_(t)) AND NOT PENDING(T+1))
14     )
```

FIGURE 6.6 – Théorème d'équivalence pour l'opérateur next_e

6.5.2 Etape d'induction

Le but est de prouver l'équation 6.13 :

$$(\forall t_0', \forall T'\ \mathcal{I}(n-1)) => (\forall t_0, \forall T\ \mathcal{I}(n)) \tag{6.13}$$

D'aprés la règle de logique suivante : $\forall p(G \Rightarrow F) \Leftrightarrow G \Rightarrow (\forall p F)$, l'équation 6.13 se réécrit alors :

$$(\forall t_0, \forall T)((\forall t_0', \forall T'\ \mathcal{I}(n-1)) => \mathcal{I}(n)) \tag{6.14}$$

En utilisant la règle $\exists p(F \Rightarrow G) \Leftrightarrow (\forall p F) \Rightarrow G$, nous obtenons :

$$(\forall t_0, \forall T, \exists t_0', \exists T')(\mathcal{I}(n-1) => \mathcal{I}(n)) \tag{6.15}$$

Soient $t_0 \in \mathcal{N}$ et $T \geq t_0$, nous prenons $t_0' = t_0$ et $T' = T$ et l'équation d'induction se réduit alors à la formule 6.16 :

$$\mathcal{I}(n-1) => \mathcal{I}(n) \tag{6.16}$$

Grâce à cette simplification, l'induction s'effectue seulement sur la profondeur n de la propriété. Les bornes t_0 et T de l'intervalle de vérification ne font pas partie de la récursion. Tout d'abord nous avons la réécriture suivante :

$$\varphi_n = \Omega_n(\varphi_{n-1}, op_n)$$

Préservation des hypothèses La première étape consiste à montrer que les hypothèses (H) sont préservées lors de l'induction effectuée dans la propriété. Autrement dit, il faut prouver l'équation 6.17.

$$H(\varphi_n, t_0, T) \Rightarrow H(\varphi_{n-1}, t_0, T) \tag{6.17}$$

Comme $Start_{n-1}$ équivaut à $Trigger_n$ (*cf.* figure 6.5), l'équation 6.17 se réécrit en 6.18.

$$\begin{cases} \exists t' \in [0..t_0], \neg Reset_n_M(t') \\ \wedge \forall t_1 \in [t'..T], Reset_n_M(t_1) \\ \wedge \forall t_2 \in [t'+1..t_0-1], \neg Start_n(t_2) \wedge \neg Trigger_n(t_2) \end{cases} \tag{6.18}$$

Cette équation ne dépend que de l'opérateur de tête de φ_n, et il suffit de générer un théorème par connecteur.

Equivalence de la sémantique Comme les hypothèses sont préservées, il suffit alors de prouver l'équation 6.19.

$$(V(\varphi_{n-1}, t_0, T) \Leftrightarrow S(\varphi_{n-1}, t_0, T)) \Rightarrow (V(\varphi_n, t_0, T) \Leftrightarrow S(\varphi_n, t_0, T)) \tag{6.19}$$

La définition de $V(\varphi, t_0, T)$ ne dépend pas de la structure de la propriété et fait seulement intervenir le signal $Valid_M$ commun à tout le générateur (*cf.* équation 6.7). Nous avons donc $V(\varphi_{n-1}, t_0, T)$=$V(\varphi_n, t_0, T)$. De plus, $P(\varphi_n, t_0, T) = P(\varphi_{n-1}, t_0, T) \wedge \neg Pending_n(T+1)$. Ainsi, pour prouver 6.19 il suffit de prouver 6.20.

$$S(\varphi_{n-1}, t_0, T) \wedge \neg Pending_n(T+1) \Leftrightarrow S(\varphi_n, t_0, T) \tag{6.20}$$

De plus $S(\varphi_{n-1}, t_0, T)$ est définie à l'aide de la fonction $\mathsf{Sem}(\varphi_n, t, T)$, et par définition de Sem, nous avons :

$$\mathsf{Sem}(\varphi_n, t_0, T) = \mathsf{Sem}_{\Omega_n}(\varphi_{n-1}, op_n, t_0, T)$$

Dans la fonction définissant Sem_{Ω_n}, φ_{n-1} est toujours utilisé comme paramètre de la fonction Sem. Ainsi, l'opérande φ_{n-1} de la fonction Sem_{Ω_n} peut-être considéré comme une fonction booléenne $\lambda t.\mathsf{Sem}(\varphi_{n-1}, op_n, t_0, T)$ et par abus de notation, $S(\varphi_n, t_0, T)$ peut être réécrite de la manière suivante :

$$S(\varphi_n, t_0, T) = \forall t \in [t_0, T], Start_n(t) \Rightarrow \mathsf{Sem}_{\Omega_n}(\lambda t.\mathsf{Sem}(\varphi_{n-1}, t, T), op_n, t_0, T)$$

Puisque $V(\varphi_{n-1}, t_0, T)$=$V(\varphi_n, t_0, T)$, l'hypothèse d'induction appliquée à φ_{n-1} sur [t_0..T] est donnée par l'équation 6.21.

$$H(\varphi_{n-1}, t_0, T) \Rightarrow (V(\varphi_n, t_0, T) \Leftrightarrow S(\varphi_{n-1}, t_0, T)) \tag{6.21}$$

Afin de prouver l'équation 6.19, la structure de φ_n et φ_{n-1} doit être prise en compte. Celle-ci est contraire aux principes de la preuve par induction où seuls l'opérateur Ω_n et l'hypothèse d'induction doivent être pris en compte. Cette difficulté est contournée en prouvant les deux équations 6.17 et 6.20. Ces deux équations impliquent l'équation 6.19.

Preuve de l'équation 6.17 En substituant $H(\varphi_n, t_0, T)$ et $H(\varphi_{n-1}, t_0, T)$ par leurs définition complète (*cf.* équation 6.5), les expressions φ_n et φ_{n-1} disparaissent. Le théorème 6.18 a été généré et prouvé à l'aide PVS. La preuve s'est déroulée à l'aide d'une trentaine de commandes PVS et a utilisé moins d'une seconde de temps système.

Preuve de l'équation 6.20 La substitution $S(\varphi_n, t_0, T)$ et $S(\varphi_{n-1}, t_0, T)$ dans l'équation 6.20 supprime seulement l'occurrence de φ_n (*cf.* 6.22).

$$\begin{aligned}\forall \varphi_{n-1}, t_0, T: \quad & ((\forall t' \in [t_0..T], Trigger(t') \Rightarrow \mathsf{Sem}(\varphi_{n-1}, t', T)) \\ & \Leftrightarrow (\forall t \in [t_0..T], Start(t) \Rightarrow \mathsf{Sem}_{\Omega_n}(\lambda t.\mathsf{Sem}(\varphi_{n-1}, t, T), op_n, t_0, T), t_0, T)))\end{aligned} \tag{6.22}$$

La preuve se termine en instanciant e par φ_{n-1}.

$$\begin{aligned}\forall e, t_0, T: \quad & ((\forall t' \in [t_0..T], Trigger(t') \Rightarrow e(t')) \\ & \Leftrightarrow (\forall t \in [t_0..T], Start(t) \Rightarrow \mathsf{Sem}_{\Omega_n}(e, t_0, T)))\end{aligned} \tag{6.23}$$

Tout le raisonnement a été modélisé et prouvé en PVS. Les fichiers incluant les modélisations et les théorèmes font de quelques dizaines à quelques centaines de lignes. Les preuves, ont été effectuées avec peu d'automatisation et ont requis plusieurs dizaines de commandes en moyenne. Le temps de preuve utilisé par l'outil reste négligeable (quelques secondes pour chaque moniteur primitif.)

6.6 Preuve des générateurs et générateurs-étendus

La preuve des générateurs et des générateurs-étendus se base sur celle des moniteurs. Tout d'abord, le bloc aléatoire peut être supprimé de la modélisation PVS en modélisant sa sortie comme une fonction quelconque. De plus, le bloc sémantique de ces trois types de composants est quasiment identique. Nous montrons que malgré quelques changements dans l'interface du bloc SEM, le code source est quasiment inchangé, et ceci n'entraine alors aucune modification sur les théorèmes utilisés. Les preuves ont dû être ré-effectuées, et les résolutions des preuves à l'aide de PVS ressemblent à celles qui sont utilisées pour les moniteurs.

Pour illustrer ceci, prenons l'opérateur PSL until. La figure 6.7 contient le code source produisant le signal *Trigger* pour le moniteur et le générateur-étendu until. Ce code est identique pour les deux types de composant. Comme le montre la figure 6.8, le code de production du *Trigger* diffère légèrement pour le générateur. Mais la seule différence tient dans la première ligne. Alors que pour le moniteur et le générateur-étendu, le signal *Cond* est une entrée du composant, dans le cas du générateur c'est une sortie. Celle-ci est produite à l'aide du bloc aléatoire embarqué dans le générateur (signal RAND_NB).

```
1  evaluate_expr: process (clk)
2  begin
3      if clk'event and clk='1' then
4          if reset_n='0' then start_t1 <= '0';
5          elsif cond = '1'  then start_t1 <= '0';
6          elsif start = '1' then start_t1 <= '1';
7          end if;
8      end if;
9  end process;
10 start_t2 <= start or start_t1;
11 start_until<=start_t2 and  not cond when OP_TYPE < 2 else start_t2;
12 trigger <= start_until;
```

FIGURE 6.7 – Code source du moniteur et générateur-étendu until pour la production de l'opérande gauche (*Trigger*)

```
1  cond <= RAND_NB(0);
2  generate_expr: process (clk)
3  begin
4      if clk'event and clk='1' then
5          if reset_n='0' then start_t1 <= '0';
6          elsif RAND_NB(0) = '1'  then start_t1 <= '0';
7          elsif start = '1' then start_t1 <= '1';
8          end if;
9      end if;
10 end process;
11 start_t2    <=  start or start_t1;
12 start_until<=start_t2 and not RAND_NB(0)when OP_TYPE<2 else start_t2;
13 trigger <= start_until;
```

FIGURE 6.8 – Code source du générateur until pour la production de l'opérande gauche (*Trigger*)

Cette différence n'a aucune incidence sur la modélisation des théorèmes et sur la preuve pour la raison suivante. Dans le cas du générateur, le signal *Cond* prend toujours pour valeur RAND_NB(0), qui est actif de manière pseudo-aléatoire. Ce signal RAND_NB(0) possède les même caractéristiques que le signal *Cond* pour le cas du moniteur. Au niveau du bloc SEM, *Cond* et RAND_NB sont deux entrées ayant exactement les même propriétés. Ainsi, les deux modélisations PVS concernant le calcul du *Trigger* sont équivalentes.

Les figures 6.9 et 6.10 montrent que le même phénomène se produit pour le code source responsable de la production du signal *Pending*. En appliquant ce raisonnement à tous les opérateurs de PSL, il est possible de prouver tous les générateurs et générateurs-étendus primitifs en se basant sur les théorèmes et les techniques de preuve utilisées pour les moniteurs. La modélisation PVS change peu, et est en fait équivalente à celle des moniteurs.

```
1  gen_pending: process (clk)
2  begin
3          if clk'event and clk='1' then
4                  if reset_n='0' then pending_t <= '0';
5                  elsif cond = '1' then pending_t <= '0';
6                  elsif start = '1' then pending_t <= '1';
7                  end if;
8          end if;
9  end process;
10 pending <= (pending_t or start) when
11         ((OP_TYPE = STRONG_EXCL_OP) or (OP_TYPE = STRONG_INCL_OP))
12         else '0';
```

FIGURE 6.9 – Code source du moniteur et générateur-étendu until pour la production du *Pending*

```
1  gen_pending: process (clk)
2  begin
3              if clk'event and clk='1' then
4                      if reset_n='0' then pending_t <= '0';
5                      elsif RAND_NB(0) = '1' then pending_t <= '0';
6                      elsif start = '1' then pending_t <= '1';
7                      end if;
8              end if;
9  end process;
10 pending <= (pending_t or start) when
11         ((OP_TYPE = STRONG_EXCL_OP) or (OP_TYPE = STRONG_INCL_OP))
12         else '0';
```

FIGURE 6.10 – Code source du générateur until pour la production du *Pending*

Chapitre 7

Mise en oeuvre et applications

Sommaire

7.1 La plateforme Horus 122
7.1.1 Le programme psl-bin 123
7.1.2 Intrumentation de circuits à l'aide d'Horus 123
7.1.3 SyntHorus 124
7.2 Exemple illustratif : Arbiter$_P$ et CDT 126
7.2.1 Instrumentation des circuits Arbiter$_P$ et CDT 126
7.2.2 Synthèse du circuit CDT 126
7.3 Cas d'étude 1 : Le contrôleur GenBuf 127
7.3.1 Présentation 127
7.3.2 Spécification du GenBuf 128
7.3.3 Résultat de l'annotation automatique 130
7.3.4 Résultats expérimentaux 131
7.4 Cas d'étude 2 : Le circuit conmax-ip 132
7.4.1 Contexte 132
7.4.2 Le contrôleur conmax-ip 136
7.4.3 Vérification du contrôleur 137
7.4.4 Modélisation de l'environnement 139
7.4.5 Evaluation de performances/Caractérisation du système complet 141
7.4.6 Résultats en synthèse pour l'instrumentation du conmax-ip . . 143
7.4.7 Bilan 146
7.4.8 Synthèse automatique du conmax-ip à partir de sa spécification 146

7.1 La plateforme Horus

Tous les outils développés dans le cadre des travaux rapportés ici font partie d'un projet plus global appellé projet Horus [OMAB08a, OMAB08b]. Ce projet s'articule autour de l'outil Horus et se décline en cinq sous-projets :

- **MAAT** (Hardware Synchronous Monitors) : synthèse automatique de moniteurs synchrones synthétisables pour la vérification en ligne de propriétés [BLOF06].
- **AMON** (Hardware Asynchronous Monitors) : synthèse automatique de moniteurs asynchrones pour la vérification de propriétés en ligne [MAFRB07]. La technologie asynchrone apporte de nombreux avantages : robustesse, faible consommation, hautes fréquences etc.
- **ISIS** (SystemC Monitors) : construction de moniteurs au niveau SystemC TLM pour vérification de propriétés sur des modèles haut niveau. À l'heure où SystemC devient un élément fondamental dans les premières étapes de nombreux flots de conception, fournir un support de vérification efficace et robuste basé sur l'ABV devenait un enjeux primordial. L'instrumentation de modèles à l'aide de moniteurs SystemC apporte une réponse à ce problème [PF08].
- **ATON** (Hardware Synchronous Generators) : une des étapes clé dans la vérification concerne la génération de stimuli efficaces. Les générateurs synchrones permettent de réaliser ceci en définissant les séquences de stimuli à l'aide de propriétés temporelles [OMAB06].
- **SYNTHORUS** (Hardware Synchronous Extended-Generators) : synthétise des circuits corrects par construction à partir de spécifications temporelles exprimées en PSL.

La combinaison de tous ces projets forme une plateforme de vérification efficace et adaptée à divers besoins. Il est possible de supporter la vérification à l'aide de moniteurs dès les premières étapes du flot de conception sur des modèles de circuits à haut niveau, puis de raffiner ces éléments de vérification au niveau HDL. À ce point, les stimuli peuvent aussi être produits à partir d'hypothèses sur l'environnement.

Enfin, alors que les systèmes de type GALS (Globally Asynchronous, Locally Synchronous) deviennent de plus en plus présents grâce à l'interconnexion de composants hétérogènes sur un même support, posséder des moniteurs en technologie asynchrone est un atout fondamental.

Finalement, si le circuit peut être décrit à l'aide d'un ensemble de propriétés temporelles, l'outil SyntHorus peut synthétiser sa description matérielle correcte par construction. L'utilisateur s'affranchit ainsi des coûteuses étapes d'implémentation et de vérification fonctionnelle.

L'outil Horus est composé de plusieurs blocs. Le module `psl-bin` représente le cœur de la plateforme. Il produit les moniteurs, les générateurs et les générateurs-étendus. Le module ISIS est responsable de la création de moniteur SystemC, et de l'instrumentation de descriptions SystemC. Ces deux modules sont couplés à une interface Java facilitant à l'utilisateur l'instrumentation de circuits. Enfin, le module SyntHorus est responsable de la synthèse de spécifications.

7.1.1 Le programme psl-bin

Ce module constitue le coeur de la plateforme Horus. Le programme `psl-bin` prend en entrée un fichier contenant une vunit PSL et fournit en sortie un moniteur, un générateur, ou un générateur-étendu. Ce programme représente vingt mille lignes de code C réparties dans une quarantaine de fichiers.

L'outil `psl-bin` est actuellement complètement re-développé par la société Dolphin Integration pour être intégré dans leur outil de simulation Sled. Ce module s'appelle Smash-Assert. Ce transfert vers l'industrie s'effectue dans le cadre du projet de recherche SFINCS (ANR-07-ARFU-009).

Comme le montre la figure 7.1, les temps de production sont bien meilleurs pour Horus que pour MyGen. Cette différence est principalement due au solveur booléen BoolSolve utilisé par MyGen pour lister toutes les valuations possibles des expressions booléennes correspondants aux conditions de transition de l'automate générateur.

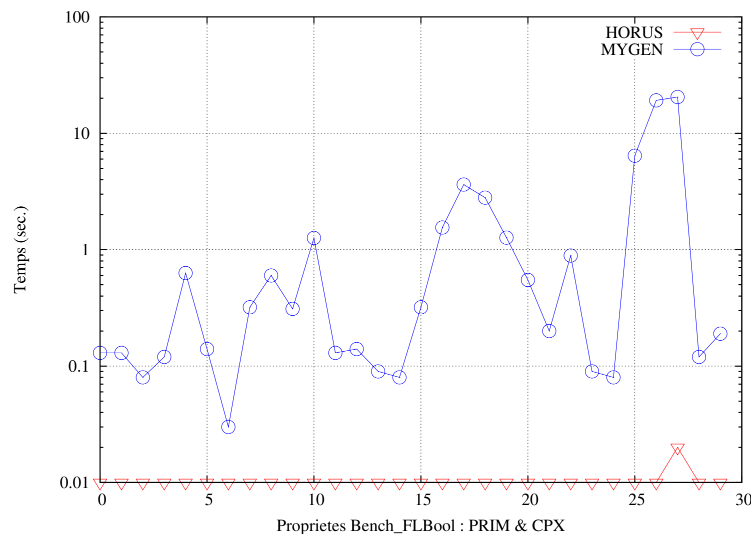

FIGURE 7.1 – Temps de production des générateurs pour Horus et MyGen

7.1.2 Intrumentation de circuits à l'aide d'Horus

L'environnement Horus aide l'utilisateur à instrumenter un circuit dans l'optique de faciliter d'éventuelles étapes de debug. Comme le montre la figure 7.2, 4 étapes sont nécessaires à l'instrumentation du circuit :

- **Etape 1 - Sélection du circuit** : le circuit et toute sa hiérarchie sont récupérés.
- **Etape 2 - Synthèse des propriétés** : récupération ou définition de propriétés, sélection du langage HDL cible et synthèse des moniteurs et générateurs. La figure 7.3 donne un aperçu de l'interface graphique d'Horus pour l'étape de synthèse de propriétés.
- **Etape 3 - Interconnexion des signaux** : grâce à l'interface graphique, l'utilisateur connecte facilement moniteurs et générateurs au circuit sous test. Toutes les variables et signaux contenus dans le circuit sont accessibles de manière hiérarchique. L'utilisateur sélectionne les signaux à connecter à chaque composant de vérification.

– **Etape 4 - Génération** : le circuit instrumenté est produit. Dans le cas où des signaux internes sont accédés par les moniteurs ou les générateurs, le circuit est légèrement modifié pour remonter ces signaux sur les sorties externes du circuit sous test (des ports peuvent être ajoutés).

Les sorties des composants de vérification sont passées à une instance d'un composant générique appellé Analyzer. Celui-ci récupère les informations des composants de vérification et envoie ces données à l'utilisateur par liaison série. Ces données fournissent un rapport de test final contenant toutes les informations pertinentes pour une éventuelle étape de debug.

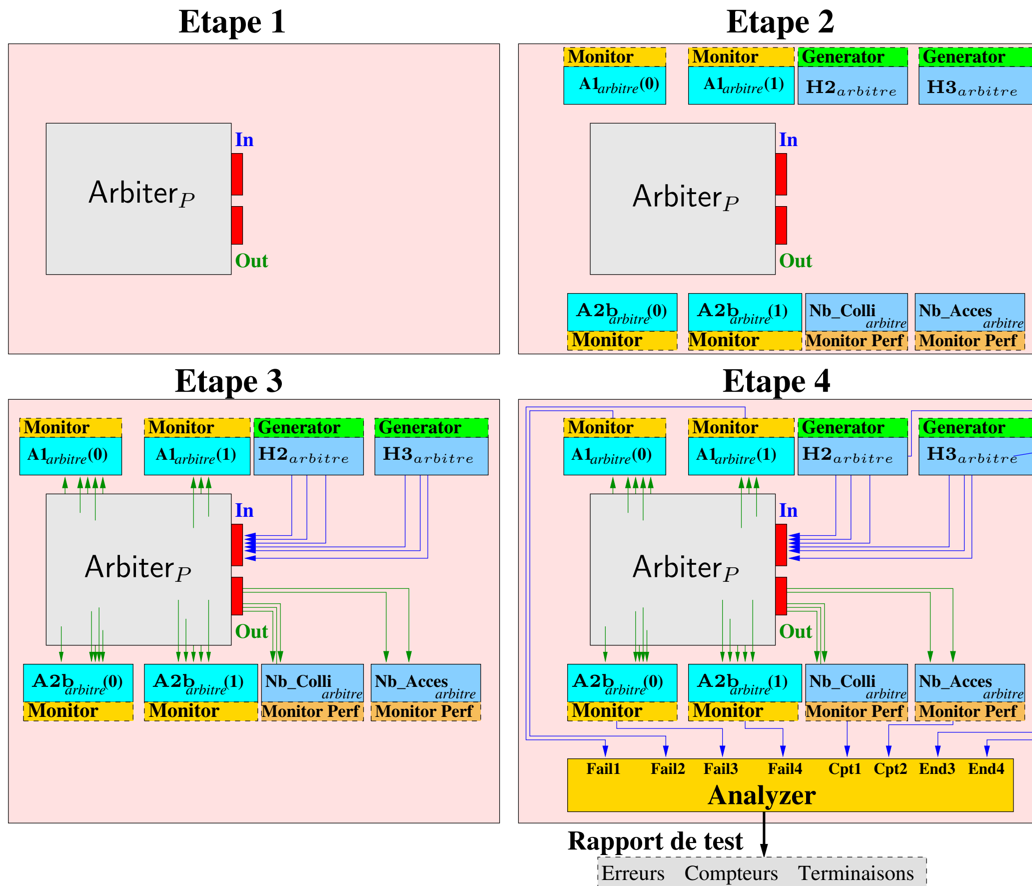

FIGURE 7.2 – Instrumentation de circuit par Horus

Le circuit instrumenté possède une interface générique compatible avec les architectures Avalon ou Wishbone. Si la plateforme FPGA utilisée possède une de ces deux architectures, l'utilisateur peut directement utiliser le circuit instrumenté sur le FPGA.

7.1.3 SyntHorus

L'outil SyntHorus fait partie intégrante de la plateforme Horus. Il fournit un environnement supportant la synthèse automatique de circuits à partir de spécifications PSL. L'outil représente 5000 lignes de code C et a été testé sur un PC portable équipé d'un processeur Dual Core et de 2Go de mémoire RAM. Le flot SyntHorus est décrit sur la figure 7.4.

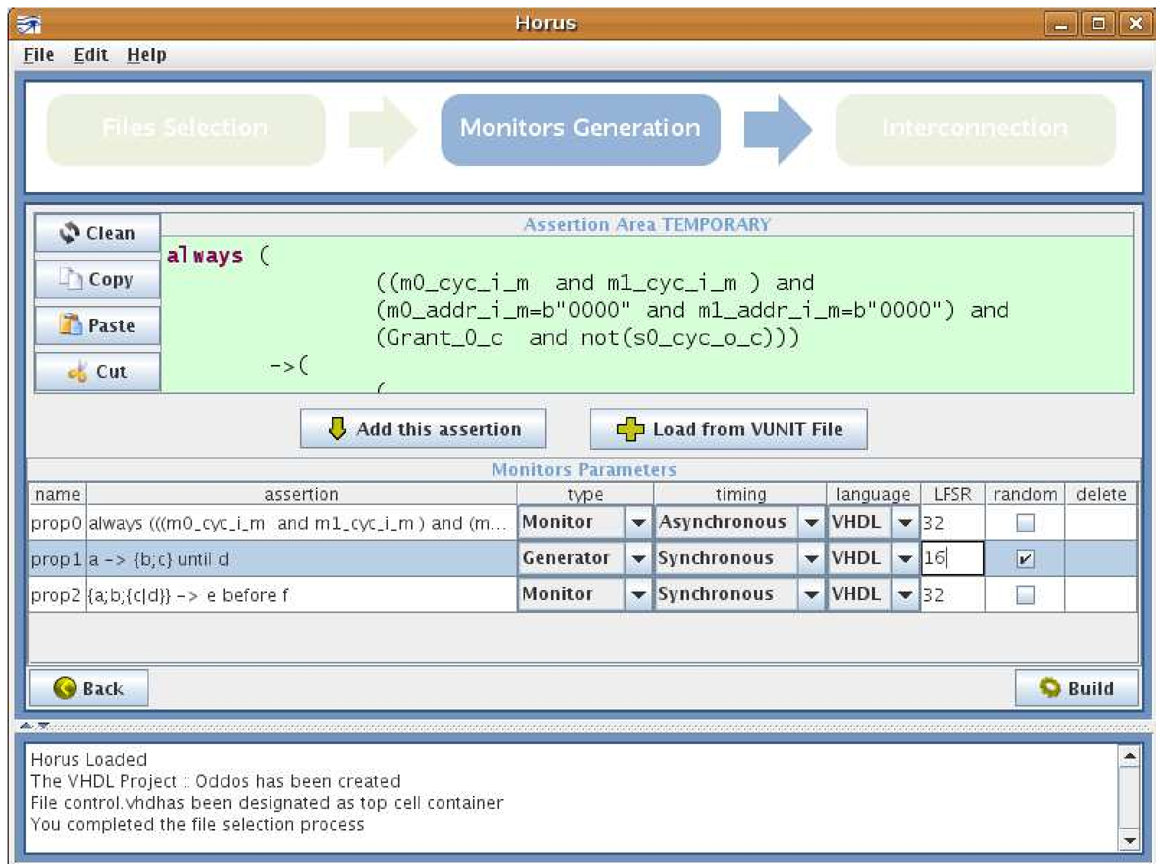

FIGURE 7.3 – Onglet Horus pour la synthèse de moniteurs et de générateurs

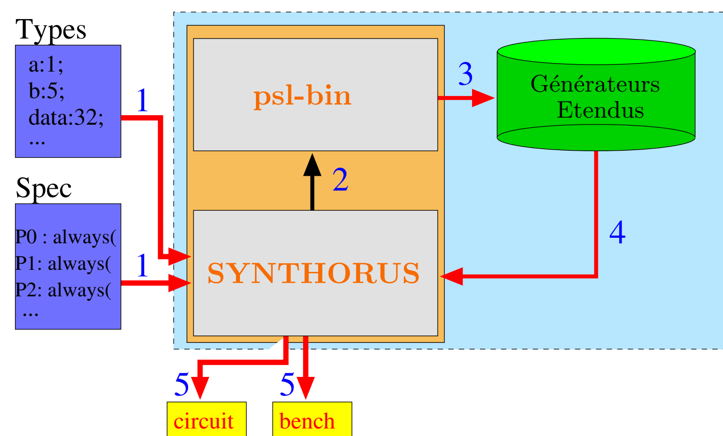

FIGURE 7.4 – Flot de syntèse pour SyntHorus

SyntHorus prend en entrée la spécification et un fichier décrivant les signaux utilisés : entrée, sortie, signal interne et type (bit, vecteur de bits etc.). La spécification peut-être partiellement annotée. Le module SyntHorus est interfacé avec le programme psl-bin qui produit les générateurs-étendus. Ensuite, SyntHorus connecte tous les générateurs-étendus entre eux à l'aide de solveurs, si nécessaire. Finalement, le résultat est encapsulé dans une entité représentant le circuit final automatiquement synthétisé.

Il est possible de faire passer dans SyntHorus des spécifications de taille considérables. Par exemple, une spécification de 700 propriétés (exemple conmax_ip décrit ci-après avec 8 maîtres et 16 esclaves) a permis de produire un circuit représentant 2,7 Mo de VHDL en 16 sec.

7.2 Exemple illustratif : Arbiter$_P$ et CDT

7.2.1 Instrumentation des circuits Arbiter$_P$ et CDT

Le tableau 7.1 contient les résultats obtenus en synthèse pour l'instrumentation de l'exemple illustratif. Moniteurs et moniteurs de performances sont peu complexes et leurs fréquences sont élevées. Les générateurs sont plus complexes, mais leurs fréquences restent assez élevées pour vérifier le circuit à sa vitesse nominale. Le surplus de complexité s'explique notamment à cause des blocs aléatoires embarqués (des LFSRs de 32 bits dans ces expérimentations).

Propriété	LCs	FFs	Freq. (Mhz)
Moniteurs			
$A1_{arbitre}$(i)	4	3	420.17
$A2b_{arbitre}$(i)	5	4	420.17
$A3_{arbitre}$(i)	5	5	420.17
$A1_{cdt}$	4	4	420.17
Générateurs			
$H1_{arbitre}$(i)	2	2	420.17
$H2_{arbitre}$	23	9	402.39
$H3_{arbitre}$	79	35	298.17
$H1_{cdt}$	57	40	372.66
$H2_{cdt}$	59	40	333.57
Moniteurs de performances			
Nb_Access$_{arbitre}$	4	4	420.17
NB_Collisions$_{arbitre}$(i)	4	4	420.17

Table 7.1 – Résultats de l'instrumentation des circuits Arbiter$_P$ et CDT

7.2.2 Synthèse du circuit CDT

L'annotation automatique de la spécification Spec_cdt a pu annoter 100% des signaux de la spécification.

La spécification Spec_cdt a été synthétisée à l'aide de SyntHorus et de MyGen. Les tableaux 7.2 et 7.3 regroupent les résultats obtenus en synthèse sur FPGA. Le circuit original (codé à la main) possède 18 cellules logiques et 6 registres, pour une fréquence de 420 Mhz.

Le circuit synthétisé par SyntHorus est plus efficace que celui produit par MyGen. puisqu'il possède moins de logique et de registres que celui construit à l'aide de MyGen. Les fréquences obtenues sont maximales dans les deux cas. Avec SyntHorus c'est le cas car le circuit est très simple, alors que pour MyGen cela est plutôt dû à un équilibre entre le nombre de cellules logiques et de registres pour les générateurs-étendus.

La complexité du circuit obtenu par SyntHorus est très proche de celle obtenue par un codage optimisé à la main.

SyntHorus	LCs	FFs	Fréq. (Mhz)
$F0_{cdt}$	1	0	-
$F1_{cdt}$	3	2	-
$F2_{cdt}$	6	6	-
Solvers	11	0	-
CDT	**21**	**8**	**420,17**

Table 7.2 – Circuit CDT produit avec SyntHorus

MyGen	LCs	FFs	Fréq. (Mhz)
$F0_{cdt}$	20	20	-
$F1_{cdt}$	19	17	-
$F2_{cdt}$	14	14	-
Solvers	11	0	-
CDT	**64**	**51**	**420,17**

Table 7.3 – Circuit CDT produit avec MyGen

7.3 Cas d'étude 1 : Le contrôleur GenBuf

7.3.1 Présentation

Notre premier cas d'étude de circuit complexe fût le Generalized Buffer (GenBuf), développé par IBM. Il permet l'interconnexion de plusieurs composants classés selon deux types : émetteurs et récepteurs. Les premiers envoient des données alors que les seconds reçoivent celles-ci. La figure 7.5 présente la structure d'une instance du GenBuf composée de 3 émetteurs. Celui-ci permet de connecter un nombre quelconque d'émetteurs (notés Emetteur0, Emetteur1 etc. sur la figure 7.5) à deux récepteurs (notés Récepteur0 et Récepteur1).

Le contrôleur GenBuf est composé d'un multiplexeur, d'une FIFO et du contrôleur lui-même. Trois interfaces se distinguent pour :

- la communication avec les émetteurs : GenBuf possède une entrée *StoB_REQ(i)* (signal de requête émis par l'émetteur lorsqu'il désire envoyer une donnée) et une sortie *BtoS_ACK(i)* (utilisée par GenBuf pour acquitter l'envoi de données par l'émetteur i) pour chaque émetteur.
- le contrôle du multiplexeur et de la FIFO : la sortie *SLC(0..P)*[1] permet de sélectionner le bus correspondant à l'émetteur qui a émis une requête. La sortie *ENQ*

[1] Pour N émetteurs, $P = log_2(N)$.

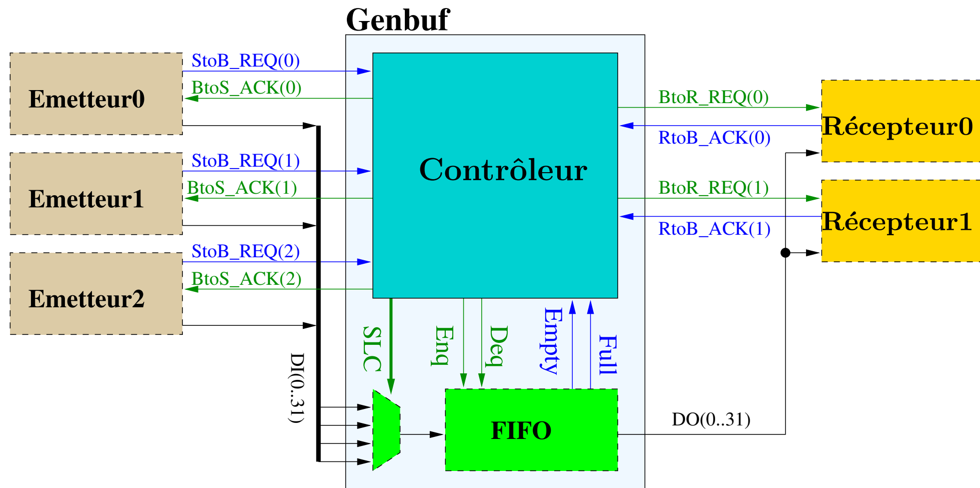

FIGURE 7.5 – Architecture du contrôleur GenBuf pour 3 émetteurs

permet d'introduire dans la FIFO la donnée émise, sélectionnée par le MUX. La sortie *DEQ* permet d'extraire un élément de la FIFO lors de la transmission de données vers un récepteurs.

- la communication avec les récepteurs : la sortie *BtoR_REQ(i)* est utilisée pour signaler une demande de transmission de données vers le récepteur i. Les données sont alors positionnées sur le bus. Une fois que le récepteur a reçu les données, il acquitte la requête en passant l'entrée *RtoB_ACK(i)* à '1'.

Les protocoles de communication entre GenBuf, les émetteurs et les récepteurs sont tous les deux des protocoles 4 phases :

- Phase 1 : l'émetteur positionne sa requête à '1'. Un cycle plus tard les données sont présentes sur le bus (cycle ♯2 figure 7.6).
- Phase 2 : au moins un cycle suivant la mise à '1' de la requête, GenBuf acquitte l'envoi (cycle ♯5 figure 7.6).
- Phase 3 : au cycle suivant l'acquittement de GenBuf, la requête est repassée à '0' (cycle ♯6 figure 7.6).
- Phase 4 : GenBuf désactive alors l'acquittement au moins un cycle plus tard (cycle ♯7 figure 7.6). Aucune transaction ne peut être effectuée tant que l'acquittement et maintenu à '1'.

7.3.2 Spécification du GenBuf

La spécification de la partie GenBuf du contrôleur a été écrite en PSL par IBM[2] dans le cadre d'un tutoriel pour RuleBase.

Les communications se font à l'aide de protocoles "poignée de mains". Le circuit GenBuf a été utilisé par Bloem *et Al.* dans [BGJ+07a] pour illustrer l'efficacité de leurs méthodes de synthèse sur des circuits complexes. La spécification qu'ils utilisent est fournie dans l'annexe C.

[2]Voir : http ://www.haifa.ibm.com/projects/verification/RB_Homepage/tutorial3/

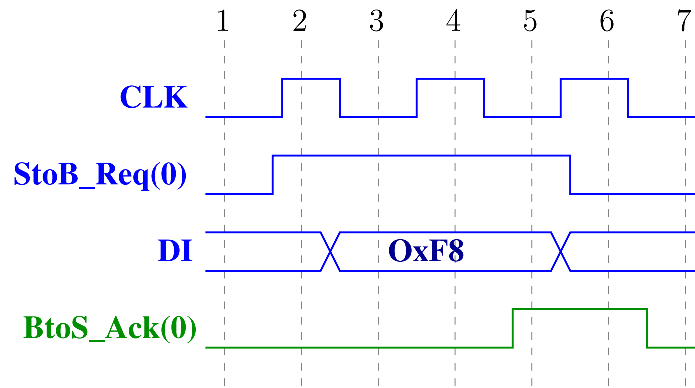

FIGURE 7.6 – Exemple de communication à l'aide du protocole 4 phases

La spécification utilisée pour SyntHorus est celle utilisée dans [BGJ+07a]. Une réécriture a été effectuée afin de synthétiser plusieurs propriétés. Nous obtenons la spécification suivante, équivalente à sa version originale :

- **MyG1** : Cette propriété est équivalente aux propriétés G1, G2 et G3 de la spécification initiale si le paramètre RANDOM des générateurs est fixé à '0'. Dans ces conditions : l'acquittement n'est pas effectué sans une requête, l'acquittement ne se produit pas au même cycle que la requête et finalement, si une requête a été émise, l'acquittement arrivera obligatoirement un jour.
  property MyG1 is $\forall i$ always($StoB_REQ(i)_o \rightarrow$ next! eventually! $BtoS_ACK(i)_g$)

- **G4** : L'acquittement est maintenu actif tant que la requête de l'émetteur est maintenu à 1.
  property G4 is $\forall i$, always($BtoS_ACK(i)_o$ and $StoB_REQ(i)_o \rightarrow$ next! $BtoS_ACK(i)_g$)

- **G6** : La requête est maintenue tant que le récepteur ne l'acquitte pas. Dès que le récepteur envoie l'acquittement, alors la requête est repositionnée à '0' au cycle suivant.
  - property G6_1 is $\forall i$, always(($BtoR_REQ(j)_o$ and not $RtoB_ACK(j)_o$) $\rightarrow$ next! $BtoR_REQ(j)_g$)
  - property G6_2 is $\forall j$, always($RtoB_ACK(j)_o \rightarrow$ next! not $BtoR_REQ(j)_g$)

- **G7** : Cette propriété correspond à la propriété G7 dans la spécification initiale. GenBuf ne doit pas envoyer deux requêtes simultanément. De plus les requêtes sont activées chacune à leur tour (schéma de distribution de type "tourniquet"). La propriété G7_2 n'est pas exactement la même car l'opérande droit du next_event a été modifié. En effet, au lieu d'interdire au même signal de s'activer deux fois de suite (ce qui en terme de génération est insuffisant), nous avons préféré exprimer clairement l'alternance des requêtes.
  - property G7_1 is always not($BtoR_REQ(0)_g$ and $BtoR_REQ(1)_g$
  - property G7_2 is $\forall j, k \in [0..1], k \neq j$, always(rose($BtoR_REQ(j)_o$) $\rightarrow$ next! next_event !(rose($BtoR_REQ(k)_o$) or rose($BtoR_REQ(1)_o$))($BtoR_REQ(k)_g$)

- **G9** : Cette propriété correspond à G9 dans la spécification initiale. Si un émetteur émet une requête, alors le signal ENQ est mis à '1' pour être prêt à insérer la donnée émise par l'envoyeur. De plus, la sélection du MUX est effectuée sur l'émetteur qui a émis la requête.

  - property G9_1 is always($\exists i$: rose$(BtoS_ACK(i)_o) \rightarrow ENQ_g$
  - property G9_2 is $\forall i$, always(rose$(BtoS_ACK(i)_o) \rightarrow SLC_g{=}i)$

- **G10** : Cette propriété correspond à G10 dans la spécification initiale. La donnée est extraite de la FIFO lorsque le transfert avec le récepteur est terminé.
  property G10 is always((fell$(RtoB_ACK(0)_o)$ or fell$(RtoB_ACK(1)_o)) \rightarrow DEQ_g)$

- **G11** : Cette propriété correspond à G11 dans la spécification initiale. Si la FIFO est vide aucun élément ne peut être extrait. De même, si la FIFO est pleine, aucun élément ne peut être inséré.

  - property G11_1 is always(($FULL_o$ and not $DEQ_o) \rightarrow$ not $ENQ_g)$
  - property G11_2 is always($EMPTY_o \rightarrow$ not $DEQ_g)$

- **G12** : Cette propriété correspond à G12 dans la spécification initiale. Tant que la FIFO n'est pas vide, des extractions seront effectuées.
  property G12 is always(not $EMPTY_o \rightarrow$ eventually! $DEQ_g)$

Les propriétés G2 et G3 n'apparaissent pas dans la nouvelle spécification car elles sont encapsulées dans G1. La propriété G5 n'apparaît pas dans la nouvelle spécification car elle semble être plutôt un assume qu'un guarantee. La propriété G8 n'apparaît pas dans la nouvelle spécification car elle est déjà contenue dans G6.

7.3.3 Résultat de l'annotation automatique

L'annotation automatique a été effectuée sur la spécification du GenBuf. Comme le montre la spécification suivante, obtenue par annotation automatique, jusqu'à 86% des signaux de la spécification ont été annotés à l'aide des règles décrites chapitre 5, section 5.2. Seuls les signaux de la propriété G7_1 n'ont pu être annotés.

Illustrons le fonctionnement de l'annotation automatique sur la propriété MyG1. La règle $\text{Rule}_{Assertion}$ (*cf.* chapitre 5, section 5.2) est appliquée en suivant la structure de la propriété.

Etape	Propriété courante	Règle appliquée	Action
1	always $StoB_REQ(i) \rightarrow$ next!eventually!$BtoS_ACK(i)$	always $\text{Rule}_{Assertion}$	
2	$StoB_REQ(i) \rightarrow$ next!eventually!$BtoS_ACK(i)$	$\text{B}_{g\vee o} \rightarrow \text{Rule}_{Assertion}$	
3	$StoB_REQ(i)$	signal d'entrée	annote "$_o$"
4	next! eventually! $BtoS_ACK(i)$	next*($\text{Rule}_{Assertion}$)	
5	eventually! $BtoS_ACK(i)$	eventually!($\text{Rule}_{Assertion}$)	
6	$BtoS_ACK(i)$	F_g	annote "$_g$"

Table 7.4 – Annotation automatique de la propriété MyG1

Comme le montre le tableau 7.4, l'annotation de MyG1 nécessite 6 étapes. A la troisième étape, le signal $StoB_REQ(i)$ est annoté en mode moniteur car c'est une entrée du circuit. L'analyse continue dans la partie droite de l'implication jusqu'à arriver à l'annotation du signal $BtoS_ACK(i)$. D'après la règle $\text{Rule}_{Assertion}$, pour que MyG1 soit une assertion, ce signal doit être annoté en mode génération. En appliquant ce raisonnement sur toutes les propriétés, l'annotation suivante est obtenue.

- property MyG1 is $\forall i$ always($StoB_REQ(i)_o \rightarrow$ next! eventually! $BtoS_ACK(i)_g$)
- property G4 is $\forall i$, always((**BtoS_ACK(i)** and $StoB_REQ(i)_o$) $\rightarrow$ next ! $BtoS_ACK(i)_g$)
- property G6_1 is $\forall i$, always((**BtoR_REQ(j)** and not $RtoB_ACK(j)_o$) $\rightarrow$ next! $BtoR_REQ(j)_g$)
- property G6_2 is $\forall j$, always($RtoB_ACK(j)_o \rightarrow$ next! not $BtoR_REQ(j)_g$)
- property G7_1 is always not(**BtoR_REQ(0)** and **BtoR_REQ(1)**)
- property G7_2 is $\forall j, k \in [0..1], k \neq j$, always(rose($BtoR_REQ(j)_o$) $\rightarrow$ next! next_event !(rose($BtoR_REQ(k)_o$) or rose($BtoR_REQ(1)_o$))($BtoR_REQ(k)_g$)
- property G9_1 is always($\exists i$: rose($BtoS_ACK(i)_o$) $\rightarrow ENQ_g$)
- property G9_2 is $\forall i$, always(rose($BtoS_ACK(i)_o$) $\rightarrow SLC_g{=}i$)
- property G10 is always((fell($RtoB_ACK(0)_o$) or fell($RtoB_ACK(1)_o$)) $\rightarrow DEQ_g$)
- property G11_1 is always(($FULL_o$ and not **DEQ**) $\rightarrow$ not ENQ_g)
- property G11_2 is always($EMPTY_o \rightarrow$ not DEQ_g)
- property G12 is always(not $EMPTY_o \rightarrow$ eventually! DEQ_g)

7.3.4 Résultats expérimentaux

La courbe figure 7.7 montre les surfaces utilisées pour GenBuf en faisant varier le nombre d'émetteurs entre 1 et 10 selon les deux approches : [BGJ+07a] et notre outil SyntHorus.

La figure 7.7 montre que la taille du circuit produit par notre approche augmente linéairement en fonction du nombre d'émetteurs (ou plus exactement en fonction du nombre de propriétés de la spécification).

L'approche [BGJ+07a] est la meilleure approche existante basée sur des automates. Elle code en dur l'énumération des comportements corrects dans la description HDL du

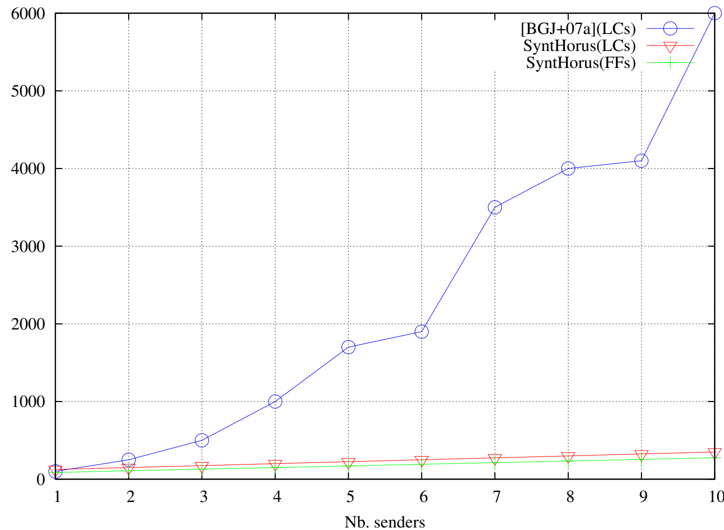

FIGURE 7.7 – Résultat de synthèse pour le GenBuf : Méthode [BGJ+07a]/Méthode SyntHorus

circuit, et possède une complexité en $0(N^3)$. La complexité cubique est clairement visible sur la figure.

Alors que le temps de production du circuit est insignifiant pour SyntHorus, il peut rapidement exploser pour [BGJ+07a] (13 heures sont requises pour la synthèse du GenBuf avec 60 émetteurs). Cette différence dans le temps de production du circuit s'explique car notre approche n'effectue aucune analyse statique de l'espace d'états de l'automate.

Aucun résultat en fréquence n'est donné pour [BGJ+07a]. La différence entre la quantité de logique (plusieurs milliers de portes) et de registres (quelques dizaines), nous fait penser que la fréquence ne doit pas être très élevée. Ce n'est pas le cas pour SyntHorus qui possède de bon résultats en fréquence (toujours supérieures à 250 Mhz).

7.4 Cas d'étude 2 : Le circuit conmax-ip

7.4.1 Contexte

7.4.1.1 Le projet OpenCores

Le projet OpenCores permet à tout utilisateur de concevoir un système sur puce complexe en réutilisant des IPs existantes. Celles-ci sont disponibles via le projet OpenCores et peuvent être récupérées sur le site de ce projet[3]. Le nombre et la variété d'IPs fournies sont conséquents et permettent une aide considérable dans le développement rapide de systèmes sur puce complexes.

Le problème lors de la réutilisation d'IPs est la connexion de celles-ci entre elles et avec les composants de l'utilisateur. Il faut comprendre l'interface des composants réutilisés et le protocole pour communiquer avec ceux-ci.

Pour lever ce problème, chaque IP développée dans le projet OpenCores possède une interface générique. Cette interface implique un protocole de communication particulier

[3] www.opencores.org

entre IPs. La définition des interfaces et du protocole de communication est contenue dans la norme Wishbone qui fait partie intégrante du projet OpenCores.

7.4.1.2 La norme Wishbone

Le document de spécification de cette norme[Her02] présente trois points fondamentaux :
- La définition de deux types de composants : maître et esclave.
- Une description des signaux composant les interfaces de communication pour maîtres et esclaves.
- Des explications détaillées des scénarios définis par le protocole de communication : lecture, écriture, rafales etc.

Chaque composant compatible Wishbone doit être soit de type maître, soit esclave. Seul un maître pourra initialiser une action de communication telle qu'une lecture ou une écriture. L'esclave répondra seulement à l'action demandée par le maître. Chacun a donc un ensemble de signaux d'interface particulier. Cet ensemble sera appelé interface maître et interface esclave pour les composants respectivement maîtres et esclaves.

Tout composant (maître et esclave) possède des ports de synchronisation *Clk* et *reset_i*, deux ports pour les données entrantes $data_i$ et sortantes $data_o$, et deux ports appelés tgc_i et tgd_o contenant des informations supplémentaires sur les données entrantes et sortantes.

Un composant maître possède trois ports indiquant si l'esclave a pu satisfaire une requête : ack_i signifie que le transfert s'est terminé correctement, err_i que le transfert a échoué, et rty_i que le transfert n'a pu débuter car l'esclave n'était pas prêt à recevoir une requête.

Tout maître possède aussi cinq signaux indiquant l'état d'un transfert :

- cyc_o : une communication valide a lieu sur le bus. Ce signal est à '1' durant toute la phase de communication.
- $lock_o$: le transfert est ininterruptible. Dés que le transfert commence et tant que ce signal, ou cyc_o vaut '1', alors le contrôleur de bus ne donne le bus à personne d'autre.
- sel_o : Indique où les données valides sont placées sur $data_o$ dans le cas d'une écriture et où les données valides doivent être écrites sur $data_i$ par l'esclave dans le cas d'une lecture.
- stb_o : transfert valide : lecture, écriture.
- we_o : Si ce signal vaut '1', alors le maître signale une écriture, sinon c'est une lecture.

Il possède enfin un port addr_o contenant l'adresse de l'esclave et l'adresse du registre où l'esclave doit placer les données.

Dans toute la suite, nous avons simplifié légèrement le protocole Wishbone en supprimant l'utilisation du signal *lock* et des Tags adresse *tga* et données *tgc*. Le signal *lock* n'est utile que pour certains transferts critiques, et si le signal *cyc* est actif, le bus est réservé aussi. Quant aux Tags, cela ajoute des données qui sont utiles au composant qui reçoit des données uniquement, mais ne complexifie pas le protocole en lui-même.

Comme le montre la figure 7.8, les interfaces maître et esclave sont totalement symétriques. A chaque port de sortie P_o d'un maître correspond un port d'entrée P_i.

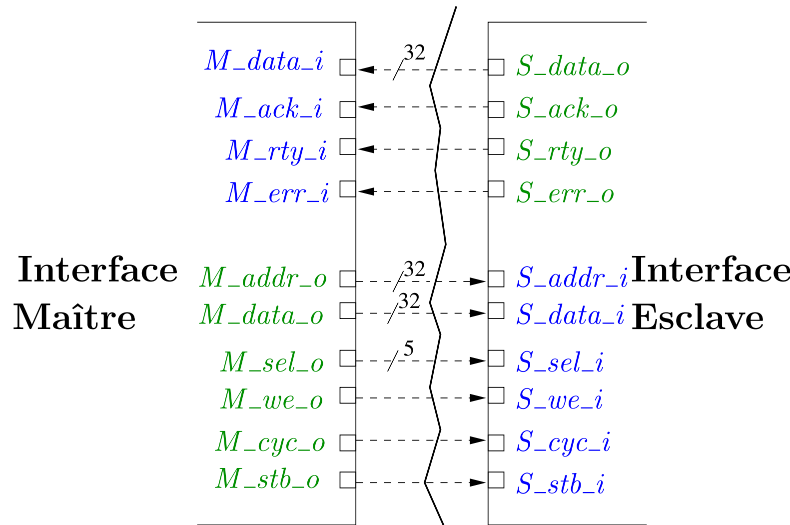

FIGURE 7.8 – Interfaces Wishbone maître/esclave

La figure 7.9 illustre le cas d'une écriture en rafale sur un esclave[4]. L'écriture est sélectionnée en plaçant le signal we_o à '1'. L'écriture en rafale commence au cycle ♯3 et termine au cycle ♯10. Le signal cyc_o réserve le bus entre ces deux instants. Deux écritures successives ont lieu entre les cycles ♯3 et ♯6. La première est acquittée immédiatement au cycle ♯4, alors que la seconde est acquittée avec un cycle de retard au cycle ♯6. Une autre écriture est effectuée au cycle ♯9. L'acquittement est immédiat.

Il est possible de remarquer la différence entre les signaux cyc_o et stb_o. Le signal cyc_o est utilisé pour réserver le bus durant tout le burst, alors que stb_o est actif seulement lors des écritures.

7.4.1.3 Architecture cross-bar

Un crossbar est un type de bus permettant une interconnexion très efficace de composants. Cette architecture permet une parallélisation des communications sur le bus. Le crossbar est une grille d'interconnexions qui se croisent sur des composants appelés "switch". Ces derniers peuvent établir ou supprimer dynamiquement des connexions entre composants connectés à cette grille. La figure 7.10 montre un cas où deux connexions parallèles sont établies. Les switchs (2,2) et (3,4) sont actifs.

Cette parallélisation permet une augmentation considérable de la bande passante du bus. En revanche, plusieurs points doivent être considérés avant d'utiliser ce type de bus :

- Il n'y a pas de parallélisation totale. Deux communications ne peuvent passer par la même ligne d'interconnexion. Ainsi les maître M2 et M4 ne peuvent pas accéder à S3 au même instant.
- Pour gérer les accès parallèles, il est nécessaire d'utiliser un arbitre. Il faut donc concevoir un arbitre adapté pour chaque instance de bus de type crossbar en fonction du nombre de composants interconnectés.

[4]NB : Un burst Wishbone peut être uniquement en lecture ou uniquement en écriture. Il n'y a pas de mélange entre les deux modes.

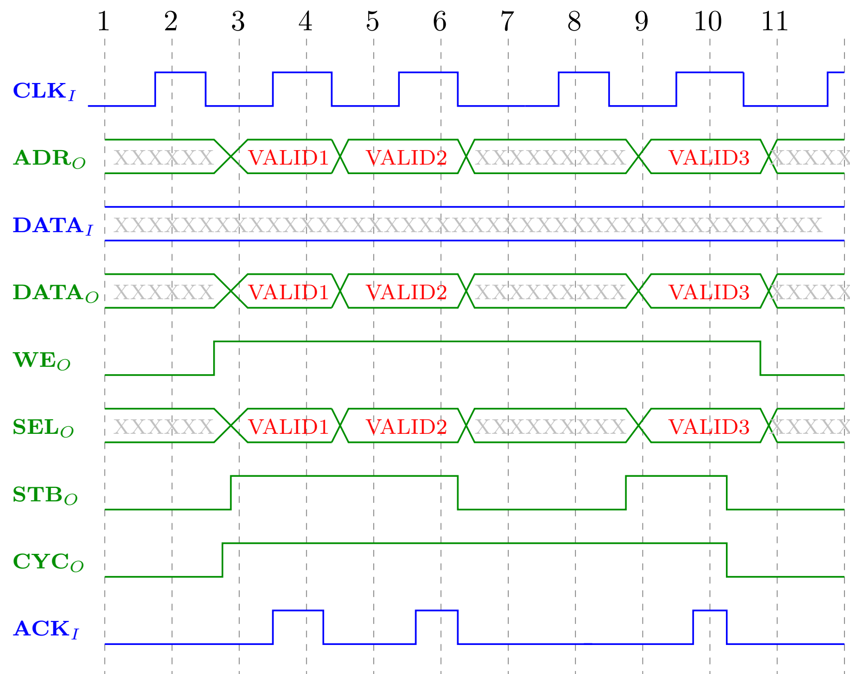

FIGURE 7.9 – Protocole Wishbone : Ecriture en rafale (burst)

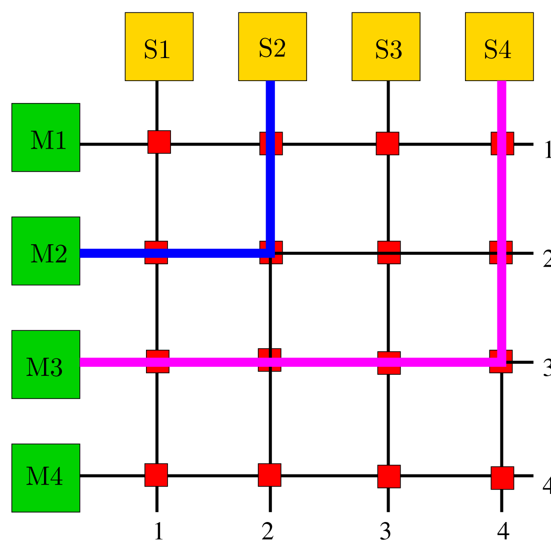

FIGURE 7.10 – Architecture de type cross-bar

- Le bus cross-bar utilise beaucoup plus de ressources (registres, cellules logiques et fils).

Dans la suite de ce document, un arbitre pour ce type de bus est étudié. Il sera de plus vérifié grâce à la plateforme Horus.

7.4.2 Le contrôleur conmax-ip

Cette section présente un contrôleur pour un bus de type cross-bar connectant jusqu'à 8 maîtres et 16 esclaves. Tous les composants sont compatibles avec la norme Wishbone. Comme le montre la figure 7.12, 8 interfaces maîtres et 16 interfaces esclaves sont connectées au contrôleur.

La figure 7.11 illustre un exemple où le conmax_ip connecte 3 maîtres à 4 esclaves. Les liens dessinés entre maîtres et esclaves représentent un état des connexions possibles à un instant donné. Le maître M1 est connecté à l'esclave S2. Deux autres communications s'effectuent en parallèle entre M2 et S1 et entre M3 et S3.

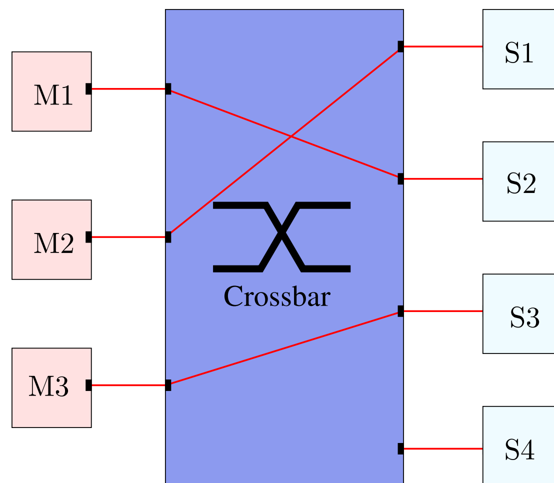

FIGURE 7.11 – conmax_ip pour 3 maîtres et 4 esclaves

Ce contrôleur peut ainsi connecter plusieurs maîtres à plusieurs esclaves en parallèle[5].

Afin de régler les cas où plusieurs maîtres demandent le même esclave au même instant, ce contrôleur possède deux règles de décision :

- Chaque maître possède une priorité. Celles-ci sont enregistrées dans un registre spécial nommé CONF. La priorité d'un maître est donnée par `CONF[2i..2i-1]`. À chaque cycle, le maître ayant la plus grande priorité peut communiquer avec l'esclave. Les autres devront attendre la terminaison de ce transfert. Le contrôleur conmax_ip peut utiliser deux, ou quatre niveaux de priorités.
- Si tous les maîtres concernés ont la même priorité, alors un algorithme de type tourniquet est utilisé. Ainsi, chaque maître possédera l'esclave tour à tour.

L'utilisation combinée de ces deux règles peut conduire à des problèmes de famine. Si des maîtres de priorités élevées demandent constamment accès aux esclaves, alors ceux de priorités inférieures ne seront jamais servis. Il faut donc définir soigneusement les priorités afin de ne pas avoir de blocages dans le système final.

[5] Toujours sous la condition que plusieurs maîtres ne demandent pas le même esclave au même instant.

La structure du contrôleur est décrite dans la figure 7.12. Elle se compose de 8 switchs permettant d'interconnecter chaque maître avec n'importe quel esclave, et d'un bloc CSR (Control Status Register). Celui-ci contient les registres de configuration et d'état du contrôleur. Les registres de configuration permettent de connaître, entre autres, les priorités de chaque maître. Les registres d'état permettent de connaître l'établissement des connexions. Pour connaître l'esclave demandé par le maître, les quatre bits de poids forts du port S_addr_i sont analysés. Ces quatres bits donnent le numéro de l'esclave auquel la connexion doit être effectuée. Par exemple, si la valeur de ce port vaut "0x30000000", alors c'est l'esclave numéro 3 qui est demandé.

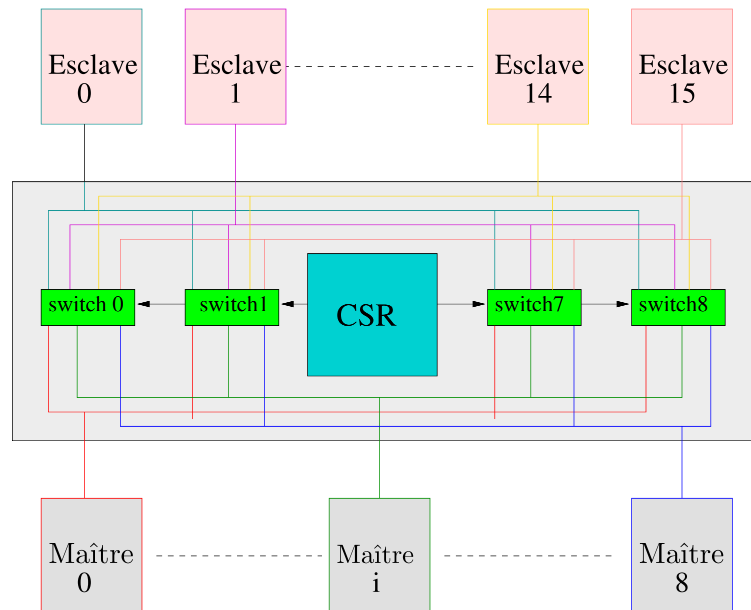

FIGURE 7.12 – Contrôleur conmax_ip : architecture

7.4.3 Vérification du contrôleur

Un ensemble de propriétés a été défini afin de vérifier plusieurs aspects du fonctionnement du contrôleur et ce, pour 8 maîtres et 16 esclaves. Pour chaque propriété, une description détaillée, ainsi qu'une trace obtenue en simulation seront données. Les moniteurs ont étés connectés au circuit conmax_ip ; et le tout simulé sous ModelSim.

7.4.3.1 Vérification de l'initialisation du conmax-ip

Tant que le signal *reset_i* du système est actif, la norme impose que tous les signaux *cyc* et *stb* doivent être maintenus à '0'. La propriété Reset_MJ vérifie ceci :

property Reset_MJ is
  assert always $reset_i \rightarrow$ (not $M^j_cyc_o$ and not $M^j_stb_o$) until not $reset_i$;

Le moniteur Reset_Mj vérifie que toutes les sorties de type *cyc* et *stb* du contrôleur sont bien maintenu à '0' lors des cycles où le signal *reset_i* est actif.

Une simulation du contrôleur conmax_ip instrumenté à l'aide des moniteurs Reset_M0, ..., Reset_M7 a été effectuée. La figure 7.13 illustre les chronogrammes obtenus en se focalisant sur les résultats obtenus au niveau des maîtres Master0 et Master1. Le premier cycle de simulation sert à initialiser les moniteurs. Les signaux *Pending* sont donc inactifs. Puis à partir du cycle suivant, le contrôleur conmax_ip étant dans un état d'initialisation, tous les moniteurs Reset_Mj sont activés (signaux *Pending* actifs). Cette vérification stoppe lorsque le signal *reset_i* devient inactif (à 57ns).

L'erreur suivante a été introduite dans le contrôleur : lorsque le signal *reset_i* est actif, le signal $M^0_cyc_o$ a été forcé à '1' à 33 ns. La figure 7.13 montre clairement que le moniteur Reset_M0 détecte une erreur. Le signal *Valid* tombe à '0' à 33 ns jusqu'à la fin de la vérification (à 57 ns).

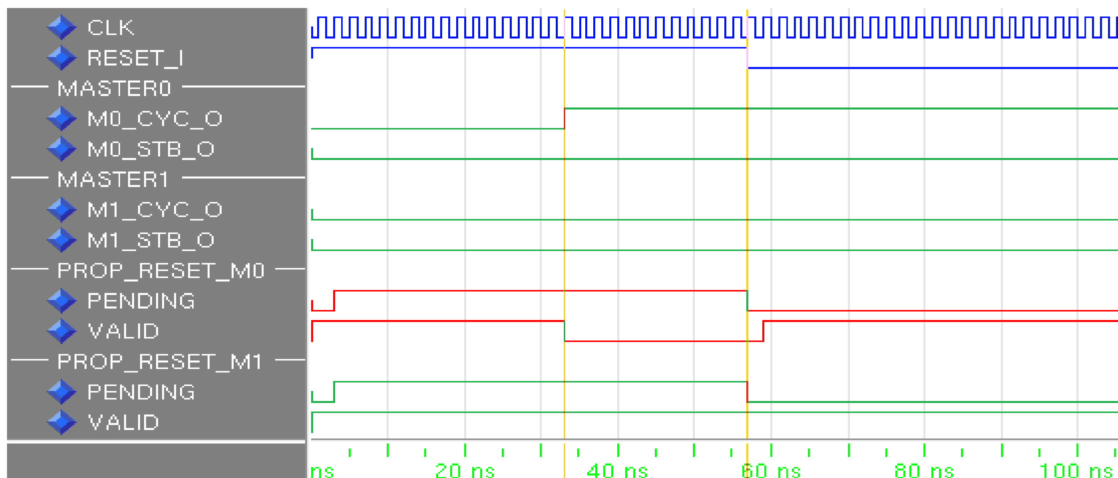

FIGURE 7.13 – Simulation pour la vérification du circuit conmax_ip au reset

7.4.3.2 Vérification des connexions

Lorsque le maître obtient un esclave, tous les ports du maître et de l'esclave doivent être correctement connectés. La propriété LinkMJ_SK vérifie que la connexion établie entre un maître numéro J et un esclave numéro K est correcte :

property LinkMJ_SK is assert always(
$M^j_cyc_o$ and $S^k_cyc_i$ and $M^j_addr_o$=$S^k_addr_i$)$\rightarrow$
($M^j_data_o$=$S^k_data_i$ and $M^j_data_i$=$S^k_data_o$
and $M^j_sel_o$=$S^k_sel_i$ and $M^j_stb_o$=$S^k_stb_i$
and $M^j_we_o$=$S^k_we_i$ and $M^j_ack_i$=$S^k_ack_o$
and $M^j_err_i$=$S^k_err_o$ and $M^j_rty_i$=$S^k_rty_o$) ;

La figure 7.14 illustre l'établissement correct de la connexion du maître M0 à l'esclave S1 de 33 ns à 53 ns. La propriété correspondante n'est pas violée.

7.4.3.3 Vérification des priorités

Le contrôleur doit respecter les priorités lorsque plusieurs maîtres accèdent un même esclave à un instant donné. La propriété PrioMJ_MK vérifie que si deux maîtres de nu-

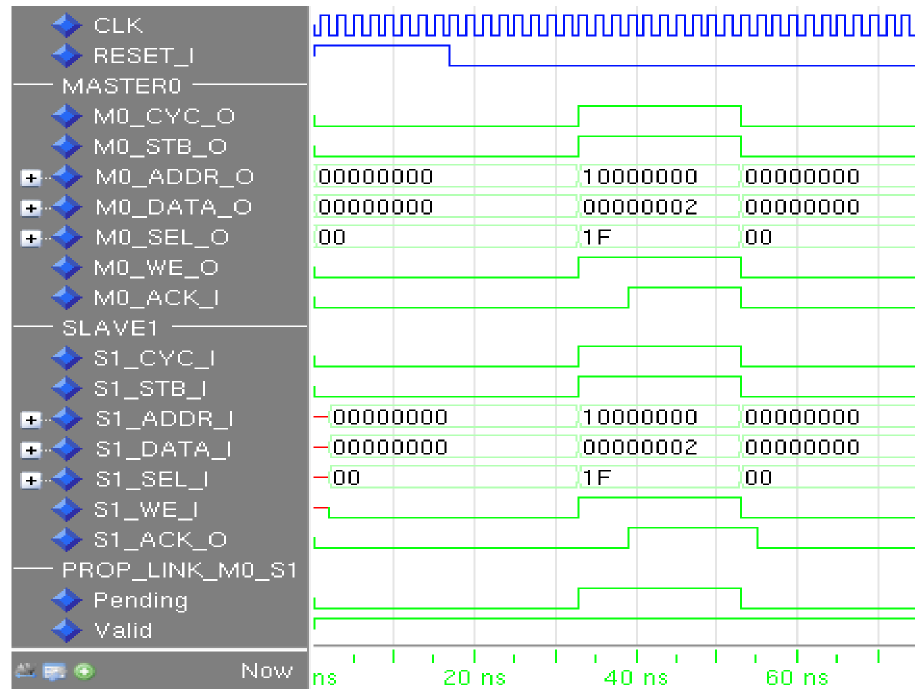

FIGURE 7.14 – Simulation pour la vérification d'une connexion correcte entre Master0 et Slave1

méros J et K de priorités respectives p_j et p_k ($p_k > p_j$) veulent accéder au même esclave à un instant donné, alors c'est le maître K qui y accède en premier.

property PrioMJ_MK is assert always($M^j_cyc_o$ and $M^k_cyc_o$ and
CONF[2k..2k-1] > CONF[2j..2j-1] and $M^j_addr_o[0..3]=M^k_addr_o[0..3]$)
$\rightarrow$ ($M^k_ack_i$ before $M^j_ack_i$) ;

La figure 7.15 est un exemple de simulation illustrant le phénomène des priorités. Les maîtres M0 et M1 envoient respectivement les données 0xC et 0x8 à l'esclave S0. Les deux maîtres initient le transfert à 55 ns. Le maître M0 étant plus prioritaire dans cet exemple, l'esclave lui est attribué en premier et la valeur 0xC est écrite dans l'esclave. Ce transfert se termine à 85 ns, ce qui laisse la possibilité au maître M1 d'effectuer son transfert.

7.4.4 Modélisation de l'environnement

7.4.4.1 Synthèse des maîtres à l'aide de générateurs

La modélisation des maîtres a été effectuée par plusieurs propriétés.

Ecriture vers un esclave : Le générateur correspondant à cette propriété exécutera l'action suivante : durant K cycles, le maître numéro *I* demandera d'effectuer une écriture dans l'esclave de numéro *J*. La valeur écrite dans l'esclave est choisie de manière aléatoire ici grâce au nombre `rand_nb`. Le numéro d'esclave est donné par le vecteur de bits `SlaveJ`.

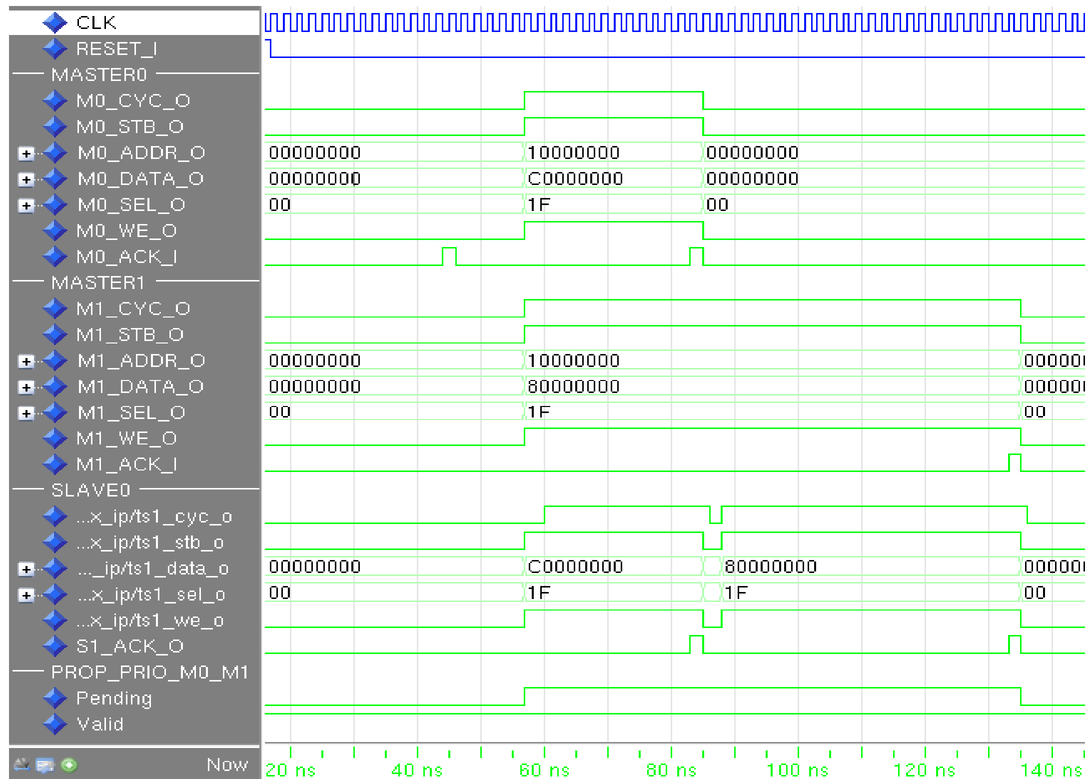

FIGURE 7.15 – Exemple de simulation pour la propriété PrioMj_Mk

property WriteMI_SJ is
assume eventually!(next_a[1..K](($M^I_cyc_o$ and $M^I_we_o$) and ($M^I_sel_o$ and $M^I_stb_o$) and ($M^I_data_o$=rand_nb) and ($M^I_addr_o$=SlaveJ))) ;

Lecture d'un esclave : De la même manière que le générateur précédent, le maître I demandera d'effectuer une lecture dans l'esclave de numéro J.

property ReadMI_SJ is
assume eventually!(next_a[1..K](($M^I_cyc_o$ and not($M^I_we_o$)) and ($M^I_sel_o$ and $M^I_stb_o$) and ($M^I_addr_o$=SlaveJ))) ;

Gestion de plusieurs maîtres : Il est possible d'automatiser encore plus la génération de stimuli en ajoutant une couche supplémentaire de générateurs. La propriété LaunchGenProp suivante illustre ce principe :

property LaunchGenProp is
assume eventually!(Start_M0 $\rightarrow$ next_e[16..32](Start_M1 and Start_M2)) ;

Cette propriété permet de définir un scénario où le maître numéro 0 est activé, puis les maîtres 1 et 2 sont activés au moins une fois entre 16 et 32 cycles plus tard. Si M1 et M2 ont la même priorité, cela permet de vérifier le bon comportement de l'algorithme

tourniquet implémenté par le contrôleur. Dans le cas contraire, cela permet de vérifier que les priorités sont bien respectées. Il est possible de définir de nombreux autres scénarios plus ou moins complexes. Le générateur `LaunchGenProp` permet de vérifier tous les points critiques du contrôleur : connexion établie correctement, priorité respectée, tourniquet bien appliqué.

7.4.4.2 Synthèse des esclaves à l'aide de générateurs-étendus

Combinés avec les générateurs décrits ci-dessus, l'utilisation de générateurs-étendus permettra d'obtenir un modèle complet de l'environnement.

Les générateurs simples ne suffisent pas pour modéliser les esclaves puisque ceux-ci réagissent en fonction du comportement des maîtres. Les sorties produites dépendent donc des entrées, ce qui justifie l'emploi des générateurs-étendus.

La modélisation des esclaves est faite grâce à plusieurs générateurs-étendus.

Réponse d'un esclave à une demande de lecture : Cette propriété modélise le comportement de l'esclave de numéro I à une requête de lecture exprimée par le maître de numéro J.

property Answer_Read_SI_MJ is
  assert always((($S^I_addr_i$=SlaveI) and(not($S^I_we_i$) and $S^I_cyc_i$ and $S^I_stb_i$))
  $\rightarrow$ next![4](($S^I_ack_o$) and $S^I_data_o$=rand_nb)) ;

La valeur lue par le maître est fixée de manière aléatoire ici via l'utilisation du nombre `rand_nb`. La réponse a été fixée quatre cycles après la requête. La donnée est passée sur le port S_data_o de l'esclave et un acquittement est envoyé au maître sur le port S_ack_o. La contrainte pourrait être moins forte en utilisant un opérateur next_e, voire même eventually!.

Pour modéliser l'échec d'un transfert (lecture ou écriture), il suffit de remplacer dans la propriété l'envoi du signal $S^I_ack_o$ par le signal $S^I_rty_o$ ou encore $S^I_err_o$.

Réponse d'un esclave à une demande d'écriture : De la même manière que pour la réponse à une lecture, il est possible de modéliser la réponse à une écriture.

property Answer_Write_SI_MJ is
  assert always((($S^I_addr_i$=SlaveI) and (($S^I_we_i$) and $S^I_cyc_i$ and $S^I_stb_i$))
  $\rightarrow$ next![4]($S^I_ack_o$)) ;

Dans ce cas, aucune donnée n'est passée sur le port S_data_o de l'esclave, et la propriété est alors simplifiée en s'affranchissant du $S^I_data_o$ dans la propriété.

7.4.5 Evaluation de performances/Caractérisation du système complet

Les travaux développés dans les sections précédentes permettent de définir un test bench (production de stimuli et analyse des réponses du contrôleur) afin de vérifier le bon comportement du contrôleur `conmax_ip` en utilisant générateurs et moniteurs.

Cette section décrit comment utiliser la plateforme Horus pour effectuer une première caractérisation (performances, quantifications de certaines actions) du système complet : contrôleur + maîtres et esclaves. Ceci peut être particulièrement utile pour mener efficacement deux analyses en parallèle lors de la simulation :

– Vérification du bon comportement du circuit sous test.
– Recueil d'informations quantitatives sur les performances du système complet.

La suite de cette section présente plus en détails le dernier point aux travers de diverses propriétés.

Nombre de transferts qui échouent : Il est possible de définir des propriétés afin de compter le nombre de transferts qui échouent. Pour cela, il suffit de détecter chaque transfert se terminant par un signal d'erreur ou de report ($S_err_o$ou S_rty_o). Les deux propriétés suivantes sont utilisées :

property CountErrorProp is cover never($M^0_err_i$ or ... or $M^7_err_i$) ;
property CountRetryProp is cover never($M^0_rty_i$ or ... or $M^7_rty_i$) ;

Nombre de collisions sur le crossbar : Une informations très importante quant au dimensionnement du système est de connaître le nombre de collisions[6]. Ceci permet de modifier rapidement le système en conséquence. La propriété suivante permet de compter les collisions :

property CountCollisionsPropMI_SJ is
cover always(($M^I_cyc_o$ and $M^I_addr_o$=SlaveJ and $M^I_data_o$=rand_nb)
$\rightarrow$ next!($S^I_cyc_i$ and $S^I_data_i$=rand_nb)) ;

Compter le nombre total de transferts durant une simulation. Ceci permettra de mieux apprécier les valeurs obtenues pour les propriétés décrites ci-dessus et de calculer différentes valeurs caractérisant le système : fréquence des collisions, taille moyenne des rafales etc. Afin d'être plus précis, trois propriétés peuvent être écrites :

Il est possible de calculer le nombre total de transferts à l'aide de la propriété suivante :
property CountTotalProp is
cover never(($M^0_ack_i$ or ... or $M^7_ack_i$) or ($M^0_err_i$ or ... or $M^7_err_i$)) ;

Les deux propriétés suivantes sont quasiment du même type que `CountTotalProp`. Elles servent à calculer le nombre total de lectures et d'écritures :

property CountTotalReadProp is cover never
((($M^0_ack_i$ or $M^0_err_i$) and not($M^0_we_o$)) and ....
(($M^7_ack_i$ or $M^7_err_i$) and not($M^7_we_o$))) ;

property CountTotalWriteProp is cover never
((($M^0_ack_i$ or $M^0_err_i$) and $M^0_we_o$) and ....
(($M^7_ack_i$ or $M^7_err_i$) and $M^7_we_o$)) ;

[6]Une collision se produit si un maître demande un esclave déjà occupé.

Mon/Gen	LCs	FFs	Freq
PropResetI	8	4	420
Nb props : 4	*32*	*16*	*420*
LinkMI_SJ	74	4	420
Nb props : 128	*9472*	*512*	*420*
PrioMI_MJ_SK	46	5	329
Nb props : 432	*19872*	*2160*	*329*
Total : 564 props	**29379**	**2688**	**329**
conmax_ip	**15084**	**1090**	-
IMPACT	**194%**	**246%**	-

Table 7.5 – Résultats de synthèse pour les moniteurs

Mon/Gen	LCs	FFs	Freq
WriteMI_SJ	81	56	420
Nb props : 128	*10368*	*7168*	*420*
ReadMI_SJ	57	36	420
Nb props : 128	*7296*	*4608*	*420*
LaunchGenProp	224	142	-
Nb props : 1	*224*	*142*	-
Total : 256 props	**17888**	**11918**	-
conmax_ip	**15084**	**1090**	-
IMPACT	**118%**	**1093%**	-

Table 7.6 – Résultats de synthèse pour les générateurs

7.4.6 Résultats en synthèse pour l'instrumentation du conmax-ip

Cette section présente les résultats obtenus en termes de surface de silicium et de fréquence maximale de fonctionnement lors de la synthèse de toutes les propriétés définies dans les chapitres précédents. Ceci permet de donner une idée précise du coût engendré par la méthode Horus si la vérification doit être menée en émulation.

7.4.6.1 Synthèse pour la vérification du contrôleur

Le tableau 7.5 permet de voir que tous les moniteurs obtenus ont des fréquences de fonctionnement élevées qui permettent ainsi de vérifier le circuit en ligne à sa vitesse nominale. Le nombre de registres utilisés par chacune des propriétés est très faible puisqu'il ne dépasse jamais 5 unités. Par contre, la quantité de cellules logiques est plus variable et oscille entre 8 et 74.

Le point important est que pour vérifier totalement le circuit, la première propriété doit être dupliquée 4 fois, la seconde 128 fois, et la troisième 432 fois[7] ! Ceci se traduit par une instrumentation finale de taille plus importante qui occupe une surface totale de l'ordre de 23 fois celle utilisée par le circuit lui-même.

7.4.6.2 Synthèse de l'environnement

Les générateurs sont plus complexes que les moniteurs et la duplication des propriétés accentue ce phénomène. Les blocs aléatoires embarqués (ici des LFSRs de 32 bits) sont en grande partie responsable de la différence notable de complexité entre générateurs et moniteurs. Les fréquences quant à elles restent toujours élevées, ce qui permet de tester le circuit à sa vitesse maximale.

Alors que pour les moniteurs, la surface utilisée représente un surcoût pour le système instrumenté, il n'en va pas de même pour les générateurs. Ceux-ci forment un modèle de l'environnement, et ils remplacent donc celui-ci. Le modèle capturant un ensemble

[7] 432=27*16 : Pour chaque esclave, il y a 27 comparaisons à effectuer

de comportements généralement moins complet que l'environnement lui-même, celui-ci, malgré sa complexité apparente, est généralement moins complexe que l'environnement complet. La surface occupée par l'ensemble des générateurs ne doit donc pas être analysée dans l'absolue, mais mise en relation avec la complexité de l'environnement réel. Les

Mon/Gen	**LCs**	**FFs**	**Freq**
AnswerReadNextSI_MJ	189	102	198
Sous-Total(128 props)	*24003*	*13056*	*198*
AnswerReadWaitSI_MJ	80	43	364
Sous-Total(128 props)	*10240*	*5504*	*364*
AnswerWriteNextSI_MJ	20	4	420
Sous-Total(128 props)	*2560*	*512*	*420*
AnswerErrorNextSI_MJ	20	4	420
Sous-Total(128 props)	*2560*	*512*	*420*
AnswerRetryNextSI_MJ	20	4	420
Sous-Total(128 props)	*2560*	*512*	*420*
AnswerWriteWaitSI_MJ	80	43	420
Sous-Total (128 props)	*10240*	*5504*	*420*
TOTAL (768 props)	**52169**	**25599**	**198**
conmax_ip	**15084**	**1090**	-
IMPACT	**345%**	**233%**	-

Table 7.7 – Résultats de synthèse pour les générateurs-étendus

générateurs-étendus ont tous une fréquence de fonctionnement supérieure à 198 Mhz. Ceci permet de tester le circuit à sa vitesse de fonctionnement nominale. Le nombre de registres est très variable : entre 4 et 102 unités. Cette variation entraîne une explosion du nombre de registres utilisés pour l'instrumentation totale du circuit à cause des duplications de chaque propriété en 128 exemplaires. De même, le nombre de cellules logiques varie entre 20 et 189, et explose pour l'instrumentation totale.

7.4.6.3 Synthèse pour l'évaluation de performances

Mon/Gen	LCs	Regs	Freq
CountErrorProp	5	3	420
Sous-Total	*5*	*3*	*420*
CountRetryProp	5	3	420
Sous-Total	*5*	*3*	*420*
CountCollisionsPropML_SJ			
Sous-Total	C1	C2	-
CountTotalProp	8	3	420
Sous-Total	*8*	*3*	*420*
CountTotalReadProp	11	3	420
Sous-Total	*11*	*3*	*420*
CountTotalWriteProp	11	3	420
Sous-Total	*11*	*3*	*420*
TOTAL	**36+C1**	**15+C2**	**420**
conmax_ip	**15084**	**1090**	-
IMPACT	**0,2%**	**1,3%**	-

Table 7.8 – Résultats de synthèse pour l'évaluation de performances

Comme le montre le tableau 7.8, les propriétés utilisées sont très simples et de ce fait possèdent une fréquence élevée (420 Mhz) et une surface très faible. Le nombre de registres ne tient pas compte des compteurs qui ne sont pas embarqués directement dans le moniteur, mais sont couplés à l'instrumentation par l'outil Horus.

Ces propriétés capturent de manière concise des comportements sur l'ensemble des composants maîtres/esclaves. Ainsi, aucune duplication ne survient pour l'analyse de performances à l'aide de moniteurs. Ceci se traduit par un surcoût très faible pour cet aspect de l'instrumentation.

7.4.6.4 Analyses

L'application de la méthode `Horus` à un circuit réel permet de voir clairement que l'instrumentation du circuit peut conduire à une explosion en surface de silicium. Ce problème dépend très fortement du circuit à vérifier et des propriétés à exprimer. Les deux principaux facteurs conduisant à une telle explosion sont :

- La duplication forte des propriétés, et ce, même si les moniteurs produits sont simples.
- L'utilisation d'un très grand nombre de propriétés pour vérifier le maximum d'aspects d'un circuit complexe.

En ce qui concerne les générateurs, leur complexité étant supérieure à celle des moniteurs, le surcoût globale de l'instrumentation croît très rapidement en fonction du nombre de duplications de propriétés. Cependant, il n'est pas obligatoire de connecter tous les générateurs au circuit sous test. Différentes campagnes de test peuvent être lancées avec divers sous-ensembles de générateurs, chacun ciblant des aspects particuliers à vérifier.

De plus, le coût en surface des générateurs ne doit pas être évalué dans l'absolu, mais mis en relation avec la surface qui serait utilisée par l'environnement de test réel.

L'évaluation de performances à base d'assertions est très efficace puisque ce type de propriétés permet de décrire des propriétés complexes, de manière concise, sur un ensemble conséquent de composants.

7.4.7 Bilan

Cette étude a permis dans un premier temps d'étudier un circuit réaliste, largement employé pour l'interconnexion d'IPs dans le cadre de la conception de systèmes sur puce. Une batterie de propriétés ont été écrites afin de couvrir 3 aspects de la vérification du circuit :

- Vérification du comportement du circuit.
- Modélisation d'un environnement contraint.
- Analyses du circuit en fonctionnement (performances).

L'application du flot Horus sur un cas réel montre qu'il est facile de couvrir ces trois aspects à partir seulement d'une bonne connaissance du langage PSL. Ceci permet donc de mener trois tâches en parallèle sans recourir à des compétences particulières dans plusieurs domaines différents (vérification formelle, génération de vecteurs de test, évaluation de performances etc.).

Cependant, cette étude à permis de faire ressortir un phénomène qui jusqu'à présent n'avait pas été pris en compte, puisque non analysé sur des circuits simples : l'explosion de l'instrumentation. La vérification complète du conmax_ip nécessite une duplication des propriétés pour vérifier toutes les combinaisons possibles de sources-destinations pour les communications. La complexité de l'instrumentation fait alors face à l'explosion combinatoire des possibilités de combinaisons en fonction du nombre de maîtres et d'esclaves. Ce phénomène, que l'on retrouve dans la plupart des composants à structure régulière, peut faire exploser la taille de l'instrumentation dans le cas d'une vérification complète du circuit.

7.4.8 Synthèse automatique du conmax-ip à partir de sa spécification

Le circuit conmax_ip a été synthétisé automatiquement par SyntHorus à partir de la spécification donnée dans l'annexe D. Toutes les expérimentations ont été menées en fixant le nombre de maîtres à 4. Les maîtres M0. La spécification a pu être complètement annotée.

La figure 7.16 regroupe les résultats obtenus lors de la synthèse de la spécification par SyntHorus. Le nombre de propriétés augmente linéairement lorsque le nombre d'esclaves varie. Le temps de synthèse augmente lui aussi linéairement, et seulement une dizaine de secondes sont requises pour construire le circuit conmax_ip pour 4 maîtres et 16 esclaves.

Un autre paramètre intéressant concerne l'évolution linéaire du nombre de solveurs. Ceux-ci ont un impact important dans le cas du circuit conmax_ip puisque une centaine de solveurs sont utilisés dans le cas où le circuit gère 16 esclaves. L'évaluation du nombre de solveurs est primordiale car ceux-ci sont combinatoires et auront donc une influence non négligeable sur la fréquence du circuit.

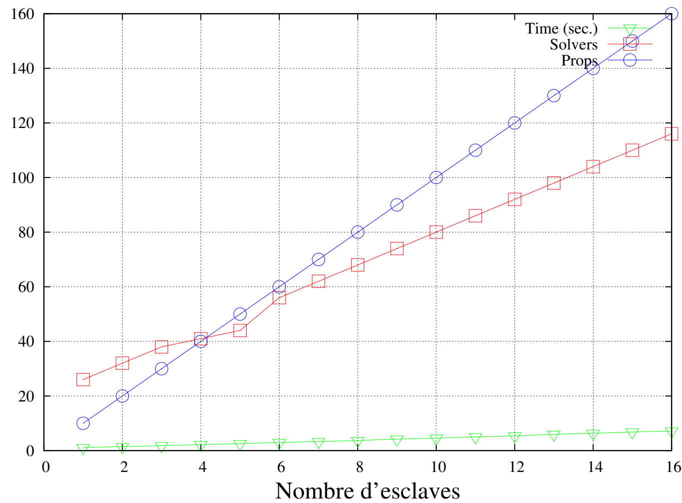

FIGURE 7.16 – Caractéristiques obtenues pour le circuit conmax_ip

Les courbes de la figure 7.17 donnent les résultats obtenus sur FPGA pour le conmax_ip produit par SyntHorus. Le nombre de ports d'entrée/sortie étant trop grand pour le FPGA dont nous disposions, les résultats en fréquences n'ont pu être fournis par Quartus que pour les circuits contenant entre 1 et 4 esclaves. Ces fréquences s'échelonnent entre 305Mhz et 200Mhz.

Le nombre de cellules logiques est important, et augmente linéairement en fonction du nombre de propriétés. Plus précisément, la quantité de cellules logiques est fortement couplée au nombre de solveurs. Dans notre expérimentation, les solveurs ont été utilisés dans leur version complète, possédant le port de sortie *Err* capable de détecter toute incohérence.

Le nombre de registres augmente proportionnellement au nombre d'esclaves, mais reste bien plus faible que le nombre de cellules logiques.

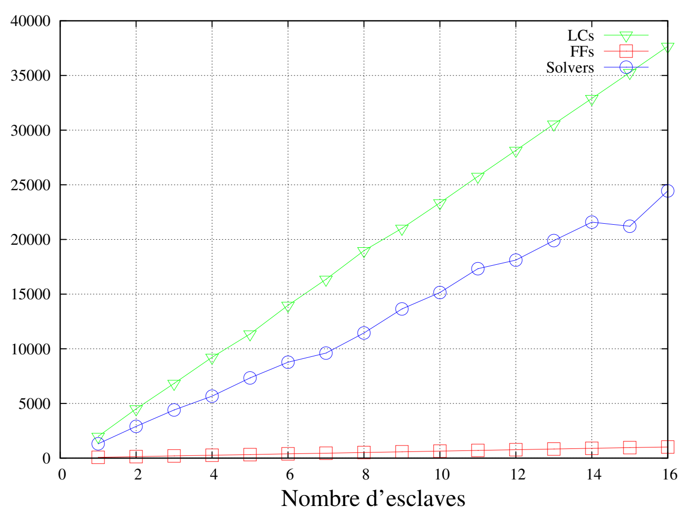

FIGURE 7.17 – Résultats en synthèse pour le conmax_ip

Finalement, la figure 7.18 illustre les différences entre la surface utilisée par le circuit

conmax_ip original et celui produit par SyntHorus. Ces résultats confirment les précédents : le nombre de cellules logiques du circuit obtenu par SyntHorus est globalement 6 fois supérieur au circuit original, alors que la différence du nombre de registres est moins marquée (facteur 5). La figure 7.18 présente une rupture de linéarité pour la complexité du conmax_ip original que nous n'avons pu expliquer jusqu'à présent.

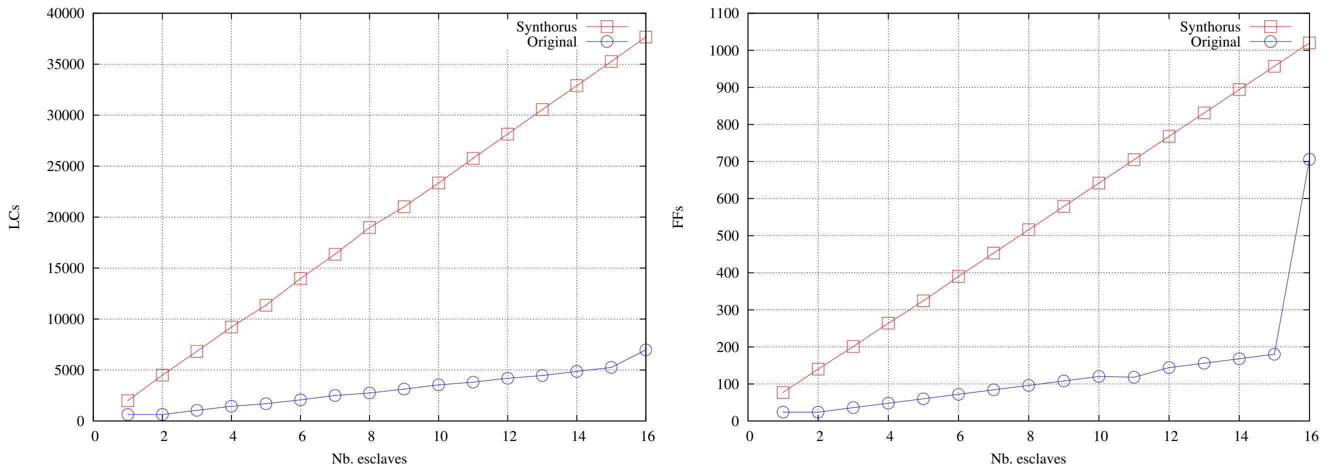

FIGURE 7.18 – Comparaison entre le conmax_ip original et celui produit par SyntHorus

La complexité du circuit obtenu par SyntHorus est bien plus complexe que celle du circuit original. Cependant, cette méthode a le mérite de fournir un circuit correct par construction dés les premières étapes du flot de conception. Ce circuit peut donc servir de modèle de référence pour les étapes de vérification tout au long du flot de conception. Non seulement la spécification fournit un document clair et fiable pour développer le circuit final, mais elle permet d'obtenir rapidement et de manière totalement automatique un élément clé dans la vérification du circuit : le modèle de référence.

De plus, cette différence de complexité est le résultat de la duplication des propriétés pour décrire le contrôleur conmax_ip. Dans plusieurs cas d'études que nous avons analysés, la spécification peut s'effectuer de manière concise, sans duplication de propriétés. Pour de tels systèmes, la différence du nombre de registres entre le circuit original et celui produit par SyntHorus est négligeable (c'est le cas du composant CDT). Du point de vue des cellules logiques, le surcoût dépend du nombre de composants de résolution utilisé, et donc du nombre de signaux dupliqués dans la spécification.

Ce dernier point ouvre la voie à une importante étude consistant à analyser s'il est possible de trouver pour toute spécification PSL, une formulation équivalente, possèdant un nombre de duplications minimal.

Chapitre 8

Conclusion et Perspectives

Sommaire

8.1 Bilan **150**
8.1.1 Instrumentation de circuits 150
8.1.2 Synthèse automatique 151
8.2 Perspectives **152**

8.1 Bilan

8.1.1 Instrumentation de circuits

L'aide à la vérification de circuits grâce à l'ABV se démocratise aujourd'hui et est supportée par les simulateurs industriels les plus importants du marché. Cette réussite est due à la capacité qu'ont les propriétés temporelles à exprimer des scénarios complexes de manière très simple. De plus, la possibilité de coupler les propriétés à la description HDL permet d'améliorer grandement la vérification et le debug tout au long du flot de conception.

Tout environnement de test se décompose en deux parties principales : génération de scénarios de test, et analyse des comportements du circuit pour ces scénarios. Alors que les moniteurs traitent efficacement la seconde partie, la génération de stimuli reste un processus complexe et coûteux.

Modélisation de l'environnement de test Nous avons proposé d'utiliser les propriétés temporelles afin d'automatiser la génération de vecteurs de test. Un générateur est un composant synthétisable produisant des vecteurs de signaux conformes à une propriété temporelle. Il permet de modéliser l'environnement d'un circuit sous test de manière simple et efficace. Notre approche synthétise un composant générateur en quelques secondes et fournit un composant matériel efficace.

L'outil MyGen(adaptation de l'outil MBAC développé à McGill pour la génération de vecteurs de test) a débouché sur une nouvelle méthode de création de générateurs à l'aide d'automates. Même si cette approche produit des générateurs de complexité plus élevée que ceux d'Horus, la principale différence se situe au niveau du bloc aléatoire. Si l'on compare uniquement le coeur des générateurs des deux méthodes l'écart est moins significatif. Par contre, les fréquences de fonctionnement maximales sont globalement meilleures pour MyGen que pour Horus.

Le modèle d'environnement s'effectue en combinant les générateurs. Si plusieurs générateurs pilotent un même signal, alors une résolution doit être effectuée afin de calculer la valeur finale pour ce signal. Des composants spécifiques de résolution ont été développés. Ils permettent de combiner les générateurs pilotant des signaux identiques afin d'obtenir un modèle d'environnement simple et efficace.

L'outil Horus Il permet de supporter la phase de test en fournissant un outil complet et simple d'utilisation gérant complètement l'instrumentation d'un circuit à vérifier. Horus permet de transformer assertions et hypothèses en moniteurs et générateurs en quelques secondes et d'interconnecter ceux-ci avec le circuit sous test à l'aide de simple clicks. L'outil combine ensuite le circuit et les composants de vérification. Il connecte ces derniers à un composant Analyzer. Celui-ci rassemble les informations pertinentes lors de la phase de test et produit un rapport de test concis et complet fournissant toutes les informations nécessaires pour une étape éventuelle de debug.

Horus prend en entrée des propriétés écrites dans les standards PSL ou SVA, ce qui permet une utilisation simple par la plupart des ingénieurs de test qui possèdent à l'heure actuelle de plus en plus de connaissances sur ces deux langages de spécification. Plusieurs caractéristiques distinguent Horus des autres outils de vérification :

- moniteurs et générateurs ont été prouvés corrects vis-à-vis de la sémantique des propriétés PSL correspondantes. Ceci garantit une fonctionnalité correcte des composants de vérification utilisés.
- alors que la plupart des outils automatisent l'analyse du comportement du circuit à l'aide de moniteurs, Horus fournit aussi le moyen d'automatiser la génération de stimuli.
- la complexité d'un moniteur et d'un générateur est linéaire vis-à-vis de la complexité de la propriété correspondante. Les composants matériels obtenus sont ainsi petits et rapides.
- les moniteurs et les générateurs peuvent être connectés simplement au circuit sous test. Les moniteurs peuvent être réutilisés pour vérifier une propriété en plusieurs endroits du circuit.

Mise en pratique de l'Assertion-Based Verification Alors que jusqu'à présent, l'état de l'art rapporte des études de cas mettant en jeux des propriétés temporelles simples sur des circuits, qui dans la majorité des cas sont de faible complexité, nous avons mis en avant les capacités et les limites de l'ABV au travers da la vérification du circuit conmax_ip. Ceci a montré que même si les composants de vérification sont très efficaces, la vérification complète de circuits complexes requiert en général un grand nombre d'assertions et d'hypothèses, rendant l'instrumentation particulièrement coûteuse.

Le choix d'embarquer des blocs aléatoires au sein des générateurs a été fait pour fournir des composants totalement indépendants de la plateforme de test, fournissant donc une méthode applicable quel que soit l'environnement de test. Cependant, si une source aléatoire est disponible sur la plateforme de test, les blocs aléatoires peuvent être retirés. Ceci simplifie largement la complexité des générateurs et donc du modèle d'environnement.

Enfin, la surface utilisée par les générateurs ne doit pas être considérée dans l'absolu, mais doit être mise en rapport avec celle qui serait utilisée par l'environnement réel. La modélisation de celui-ci simplifie généralement les comportements réels de l'environnement soit en réduisant le nombre de comportements, soit en simplifiant directement ceux-ci. Dans la plupart des cas, le modèle obtenu est alors moins complexe que l'environnement réel et mieux approprié pour une première phase de test.

8.1.2 Synthèse automatique

Au delà de la nécessité d'avoir une étape de vérification la plus efficace possible pour détecter et corriger tout bug éventuel introduit lors de l'écriture de la description matérielle du circuit, nous avons proposé une méthode synthétisant automatiquement un ensemble de propriétés temporelles en un composant matériel. Ce dernier étant correct par construction, les étapes d'implémentation et de vérification fonctionelle peuvent être supprimées du flot de conception. Ce sont ainsi deux phases complexes et fortement coûteuses qui disparaissent.

Etant donné le gain considérable qu'une telle méthode pourrait apporter, de nombreux chercheurs se sont penchés sur ce problème. La solution possède une complexité qui explose avec la taille de la spécification et rend le procédé impraticable pour des circuits réels. Pour résoudre ce problème, l'idée consiste soit à se focaliser sur un ensemble de circuits particulier, soit à restreindre fortement le type de propriétés traitées.

Notre approche Nous proposons ici une méthode capable de synthétiser efficacement une spécification en un composant synthétisable correct par construction. Pour cela, un nouveau composant générateur-étendu a été défini. Il permet de produire une trace spécifique lorsqu'une séquence de signaux particulière a été observée. La complexité finale du circuit est encapsulée dans toute une hiérarchie de composants prouvés corrects. La synthèse du circuit final s'effectue en une poignée de secondes, même pour des spécifications de plusieurs centaines de propriétés temporelles complexes. Ceci constitue à notre connaissance la première méthode de complexité linéaire permettant la synthèse de spécifications temporelles en circuits.

SyntHorus Cet outil automatise l'approche de synthèse. Tout d'abord, une annotation de la spécification est effectuée afin de déterminer quels signaux sont observés et générés. Cette étape n'est pour l'instant que partielle et peut requérir une intervention de l'utilisateur afin de compléter l'annotation. Ensuite, les générateurs-étendus sont construits et interconnectés via des composants de résolution : les solvers. La combinaison de tous ces composants forme le circuit final. La preuve de cette approche a été effectuée à l'aide de l'outil de preuve de théorèmes PVS. Les circuits obtenus sont donc garantis correct par construction.

Cas d'étude Nous avons pu montrer l'efficacité de notre approche au travers du contrôleur GenBuf. La comparaison entre notre méthode et celle décrite dans [BGJ+07a] illustre la complexité linéaire de notre approche, là où les meilleurs approches décrites dans l'état de l'art rapportent des complexités polynomiale en $O(N^3)$ [BGJ+07a, Reg05, SF07, FJR09].

Enfin la synthèse du conmax_ip a été effectuée. Des vérifications ont été faites en simulation et par model-checking pour s'assurer que le circuit produit par SyntHorus correspondait à sa spécification, et donc au circuit conmax_ip original. Cet exemple a permis de montrer qu'il suffit de quelques secondes pour synthétiser des spécifications très complexes de plusieurs centaines de propriétés.

8.2 Perspectives

Les travaux décrits dans cette thèse établissent les bases solides d'une méthode de modélisation d'environnements de test et de synthèse automatique de spécifications temporelles. Les résultats obtenus sont très encourageants et l'efficacité des approches définies tout au long de ces travaux les rend compétitives par rapport à l'état de l'art. Cependant, les techniques décrites ici ne sont pas une fin en soi, et font apparaître de nombreuses perspectives d'évolutions prometteuses pour enrichir ce qui a été fait.

Alors que nous nous somme concentrés sur des blocs aléatoires simples utilisant des lois de distribution uniformes, il serait intéressant de pouvoir modifier ces lois dans les blocs aléatoires. Ceci permettrait de modifier la taille de certaine traces, la fréquence des évènements etc. Pour MyGen, plusieurs modes sont disponibles pour l'aléatoire. Cependant, ceci n'agit pas sur les lois de distribution utilisées par le bloc aléatoire, mais sur l'automate du générateur lui même. Des travaux intéressants, pouvant être un point de départ pour la définition de nouveaux blocs aléatoires plus sophistiqués, sont rapportés dans [CLLV07].

Des langages tels que PSL et SVA ne sont pas adaptés pour la spécification de parties opératives. Nous avons défini une simple FIFO en PSL, et la complexité de la spécification a très bien illustré ce problème. Nous aimerions pouvoir intégrer au sein d'une même vunit, la spécification de la partie contrôle et le code VHDL de la partie opérative. Ceci permettrait d'avoir une description complète du circuit à synthétiser. L'objectif qui nous semble prometteur serait d'améliorer SyntHorus pour qu'il puisse prendre cette vunit hybride (mixant propriétés et code HDL), et synthétiser le circuit final combinant parties opérative et contrôle.

L'annotation de spécifications est un problème à part entière qui n'a été que partiellement résolu. Un travail plus en profondeur doit être effectué pour déterminer si une annotation complète est toujours possible quelle que soit la spécification.

Enfin, dans tous ces travaux, nous nous sommes concentrés sur l'efficacité des circuits produits en termes de surface et de fréquence. Aujourd'hui, posséder des circuits petits et rapides ne suffit plus. Les spécifications fixent la plupart du temps des contraintes de robustesse et surtout, de consommation.

Ceci prend tout son sens dans le cas où des moniteurs sont embarqués directement sur le silicium. L'instrumentation étant alors partie intégrante du circuit, elle doit être robuste pour ne pas fournir d'informations erronées sur l'état du circuit. De plus, l'énergie consommée par le module de vérification doit être la plus faible possible. De la même manière, SyntHorus et MyGen devraient pouvoir s'adapter à ce nouveau type de contraintes.

Les outils de conception ont commencé à intégrer les contraintes de consommation et de robustesse depuis plusieurs années. Actuellement, ces domaines sont en pleine expansion et le manque d'outils de vérification adaptés commence à se faire ressentir. Ceci n'échappe pas au domaine de l'Assertion-Based Verification qui commence à s'adapter [MAFRB07, JBIF09]. Ces considérations ouvrent tout un large champ de recherches pour adapter les méthodes de synthèse de propriétés de manière à produire des circuits efficaces, robustes, et à faible consommation.

Annexe A

Exemple Illustratif : Arbiter$_P$ et composant CDT

```
vunit {Arbiter_i}(Arbiter_p){
    default clock is rising_edge(clk);
    -- assertions used for monitors
    for i in 0 to P loop
        property A1_arbitre(i)  is
            assert always (Ask_i -> eventually! Grant_i);
    end loop;
    property OBE12_arbitre(i) is
        assert A (never (Use_1 and Use_2));
    property A2a_arbitre(i) is
        assert always ((Ask_i and next[2]Use_i) -> next[1]Grant_i);
    property A2b_arbitre(i) is
        assert always ({Ask_i;not Grant_i}|->{true;not Use_i});
    property A1_cdt is assert always (Req -> (Busy until! Ack));
    property S_arbitre is {Ask_i;true[*];Grant_i;Use_i[*5]}
    property A3_arbitre(i) is
        assert always ({Ask_i;Grant_i}|->{true;Use_i});
    -- assumptions used for generators
    for i in 0 to P loop
        property H1_arbitre(i) is
            assume always (not(Ask_i and Use_i));
    end loop;
    property H2_arbitre is
        assume always {Ask_i |-> {true[*];Grant_i;Use_i[*5]}};
    property H3_arbitre is
        assume{{Ask_i;true[*];Grant_i;Use_i[*5]}[*3]};
    -- assertions used for performance monitors
    property Nb_Access_arbitre is  cover always rose(tk_0);
    for i 0 to P loop
        property NB_Collisions_arbitre(i) is
            cover always(ask_i and Used);
    end loop;
}
```

FIGURE A.1 – vunit pour le circuit Arbiter$_P$

```
1  vunit {CDT_spec}(CDT){
2      default clock is rising_edge(\clk);
3      -- assertions used for monitors
4      property A1_cdt is assert always(Req ->(Busy until! Ack));
5      -- assumptions used for generators
6      property H1_cdt is assume always(Req ->(Busy until Ack))
7      property H2_cdt is
8          assume always(Req ->Data="00001111" until Ack);
9  }
```

Figure A.2 – vunit pour le circuit CDT

Annexe B

Propriétés utilisées pour les expérimentations

B.1 Batterie de propriétés FLs et Booléennes : Bench_FLBoo

- property $\mathbf{PRIM}_{0}$ is (not*sig1*) ;
- property $\mathbf{PRIM}_{1}$ is (*sig1* →*sig2*) ;
- property $\mathbf{PRIM}_{2}$ is (*sig1*) ;
- property $\mathbf{PRIM}_{3}$ is (*sig1* and *sig2*) ;
- property $\mathbf{PRIM}_{4}$ is (*sig1* or*sig2*) ;
- property $\mathbf{PRIM}_{5}$ is always(*sig1*) ;
- property $\mathbf{PRIM}_{6}$ is eventually!(*sig1*) ;
- property $\mathbf{PRIM}_{7}$ is (*sig1* until*sig2*) ;
- property $\mathbf{PRIM}_{8}$ is (*sig1* until_*sig2*) ;
- property $\mathbf{PRIM}_{9}$ is (*sig1* before*sig2*) ;
- property $\mathbf{PRIM}_{10}$ is (*sig1* before_*sig2*) ;
- property $\mathbf{PRIM}_{11}$ is (next!(*sig1*)) ;
- property $\mathbf{PRIM}_{12}$ is (next!$[6]$(*sig1*)) ;
- property $\mathbf{PRIM}_{13}$ is (next_a[2 :6](*sig1*)) ;
- property $\mathbf{PRIM}_{14}$ is (next_e[2 :6](*sig1*)) ;
- property $\mathbf{PRIM}_{15}$ is (next_event(*sig1*)(*sig2*)) ;
- property $\mathbf{PRIM}_{16}$ is (next_event(*sig1*)[6](*sig2*)) ;
- property $\mathbf{PRIM}_{17}$ is (next_event_a(*sig1*)[2 :6](*sig2*)) ;
- property $\mathbf{PRIM}_{18}$ is (next_event_e(*sig1*)[2 :6](*sig2*)) ;
- property $\mathbf{CPX}_{19}$ is always((*en_load* & *en_ud*) →next_event(*sig1*)[4](!*sig2*)) ;
- property $\mathbf{CPX}_{20}$ is always(!*resetg* →(*en_load* & *en_ud*) until(*sig1* & *sig2*)) ;
- property $\mathbf{CPX}_{21}$ is always(*sig1* →next!(*sig2* & *sig3*)) ;
- property $\mathbf{CPX}_{22}$ is always((*sig1* →*sig2*) -> (*sig3* before*sig4*)) ;
- property $\mathbf{CPX}_{23}$ is always((*en_load* or *en_ud*) before(*sig1* & (*sig2*))) ;
- property $\mathbf{CPX}_{24}$ is always((*en_load* & *en_ud*) →*sig1*) ;
- property $\mathbf{CPX}_{25}$ is always(!*resetg* →next_e[1 :10](*en_load* or *en_ud*)) ;
- property $\mathbf{CPX}_{26}$ is always(!*resetg* →next_event_a(*sig1*)[3 :10](*en_load* or *en_ud*)) ;
- property $\mathbf{CPX}_{27}$ is always(!*resetg* →next_event_e(*sig1*)[3 :10](*en_load* or *en_ud*)) ;
- property $\mathbf{CPX}_{28}$ is always(*sig1* or *sig2* or *sig3* or *sig4* or *sig5* or *sig6* or *sig7* or *sig8*) ;
- property $\mathbf{CPX}_{29}$ is always(*sig1* →(*sig2* & !*sig2*)) ;
- property $\mathbf{CPX}_{30}$ is always(*sig1* →next![1] (!*sig1*)) ;
- property $\mathbf{LIM}_{31}$ is always(*sig1* →next!$[128]$(*sig2*)) ;

- property $\mathbf{LIM}_{32}$ is always(*sig1* →next!$[256]$(*sig2*)) ;
- property $\mathbf{LIM}_{33}$ is always(*sig1* →next!$[512]$(*sig2*)) ;
- property $\mathbf{LIM}_{34}$ is always(*sig1* →next_e[1 :128](*sig2*)) ;
- property $\mathbf{LIM}_{35}$ is always(*sig1* →next_e[1 :256](*sig2*)) ;
- property $\mathbf{LIM}_{36}$ is always(*sig1* →next_e[1 :512](*sig2*)) ;
- property $\mathbf{LIM}_{37}$ is always(*sig1* →next_event(*sig2*)[128](*sig3*)) ;
- property $\mathbf{LIM}_{38}$ is always(*sig1* →next_event(*sig2*)[256](*sig3*)) ;
- property $\mathbf{LIM}_{39}$ is always(*sig1* →next_event(*sig2*)[512](*sig3*)) ;

B.2 Batterie de propriétés SEREs : Bench_SERE

- property $\mathbf{SERE_PRIM}_{0}$ is {*sig1* ; *sig2*} ;
- property $\mathbf{SERE_PRIM}_{1}$ is {*sig1*[*]} ;
- property $\mathbf{SERE_PRIM}_{2}$ is {*sig1*[*8]} ;
- property $\mathbf{SERE_PRIM}_{3}$ is {*sig1*[→8]} ;
- property $\mathbf{SERE_PRIM}_{4}$ is {*sig1*[:8]} ;
- property $\mathbf{SERE_PRIM}_{5}$ is {*sig1* && *sig2*} ;
- property $\mathbf{SERE_PRIM}_{6}$ is {*sig1* or*sig2*} ;
- property $\mathbf{SERE_PRIM}_{7}$ is {*sig1* : *sig2*} ;
- property $\mathbf{SERE_CPX}_{8}$ is always({{{*sig2*; *sig3*}[*4]} ; *sig4*}) ;
- property $\mathbf{SERE_CPX}_{9}$ is always({{{{*sig1*[*4]} ; *sig2*}[*8]} ; {*sig3*[*]; {*sig4* && *sig5*}[*2]}}) ;
- property $\mathbf{SERE_CPX}_{10}$ is always({*sig1*; *sig2*[*]} |-> {*sig3*[*2] ; *sig4*}) ;
- property $\mathbf{SERE_CPX}_{11}$ is always({*sig1*[*]; *sig2*} |-> (*sig3* before*sig4*)) ;
- property $\mathbf{SERE_CPX}_{12}$ is always({{*sig1* ; *sig2*} [*10]; *sig3*[*]}[*3] or →{*sig4* ;*sig5*[*3]}) ;
- property $\mathbf{SERE_CPX}_{13}$ is always({{*sig1*}[→4]; {{*sig3*} && {*sig4*}[*]}} |-> {*sig4*; *sig5*[:3]}) ;
- property $\mathbf{SERE_CPX}_{14}$ is always{{{*sig1*[*]; *sig2*[*]; *sig3*[*]}&& {*sig4*[*5 :7]}}: {*sig3*[*]}} ;
- property $\mathbf{SERE_CPX}_{15}$ is eventually!{{{*sig1*[*]; b[*]; *sig3*[*]}&& {*sig4*[*5 :7]}} : {*sig3*[=]}} ;
- property $\mathbf{SERE_CPX}_{16}$ is always(*sig1* →{{*sig2* ;*sig3*}[*]; *sig4*[*3]} until*sig5*) ;
- property $\mathbf{SERE_CPX}_{17}$ is always({*sig1*; *sig2*[*]} && {{!*sig2*}[*5] ; *sig4*}) ;
- property $\mathbf{SERE_LIM}_{18}$ is always({*sig1*} |-> {*sig2*[*128]}) ;
- property $\mathbf{SERE_LIM}_{19}$ is always({*sig1*} |-> {*sig2*[*256]}) ;
- property $\mathbf{SERE_LIM}_{20}$ is always({*sig1*} |-> {*sig2*[*512]}) ;

Annexe C

Spécification du contrôleur GenBuf

C.1 Spécification du GenBuf donnée dans [BGJ+07a]

- **G1** :
  - $\forall i$, always($StoB_REQ(i)$ → eventually! $BtoS_ACK(i)$)
  - $\forall i$, always(not $StoB_REQ(i)$ → eventually! not $BtoS_ACK(i)$)
- **G2** : $\forall i$, always(rose($StoB_REQ(i)$) → not $BtoS_ACK(i)$)
- **G3** : $\forall i$, always(rose($StoB_REQ(i)$) → prev($StoB_REQ(i)$))
- **G4** : $\forall i$, always($BtoS_ACK(i)$and $StoB_REQ(i)$ → next! $BtoS_ACK(i)$)
- **G5** : $\forall i, i' \neq i$, alwaysnot($BtoS_ACK(i)$and $BtoS_ACK(i')$)
- **G6** :
  - $\forall i$, always(($BtoR_REQ(i)$and not $RtoB_ACK(i)$) → next! $BtoR_REQ(i)$)
  - $\forall i$, always($RtoB_ACK(i)$ → next! not $BtoR_REQ(i)$)
- **G7** :
  - always¬($BtoR_REQ(0)$and $BtoR_REQ(1)$
  - $\forall i$, always(rose($RtoB_ACK(i)$) → next!
    next_event!(rose($BtoR_REQ(i)$)or rose($BtoR_REQ(1)$))(not $BtoR_REQ(i)$)
- **G8** : $\forall i$, always($RtoB_ACK(i)$ → next! not $BtoR_REQ(i)$)
- **G9** :
  - always(ENQ ↔ $\exists i$: rose($BtoS_ACK(i)$))
  - $\forall i$, always(rose($RtoB_ACK(i)$) → $SLC = i$)
- **G10** : always(DEQ ↔ (fell($RtoB_ACK(0)$)and fell($RtoB_ACK(1)$)))
- **G11** :
  - always(($FULL$and not DEQ) → not ENQ)
  - always($EMPTY$ → not DEQ)
- **G12** : always(¬$EMPTY$ → eventually! DEQ)

C.2 Spécification du GenBuf pour SyntHorus

- property MyG1 is $\forall i$ always($StoB_REQ(i)_o$ → next! eventually! $BtoS_ACK(i)_g$)
- property G4 is $\forall i$, always(($BtoS_ACK(i)_o$ and $StoB_REQ(i)_o$) → next ! $BtoS_ACK(i)_g$)

- property $G6_1$ is $\forall i$, always($(BtoR_REQ(j)_o$ and not $RtoB_ACK(j)_o)$ $\rightarrow$ next! $BtoR_REQ(j)_g$)
- property $G6_2$ is $\forall j$, always($RtoB_ACK(j)_o \rightarrow$ next! not $BtoR_REQ(j)_g$)
- property $G7_1$ is always not($BtoR_REQ(0)_o$ and $BtoR_REQ(1)_o$)
- property $G7_2$ is $\forall j, k \in [0..1], k \neq j$, always(rose($BtoR_REQ(j)_o$) $\rightarrow$ next! next_event !(rose($BtoR_REQ(k)_o$) or rose($BtoR_REQ(1)_o$))($BtoR_REQ(k)_g$)
- property $G9_1$ is always($\exists i$: rose($BtoS_ACK(i)_o$) $\rightarrow ENQ_g$)
- property $G9_2$ is $\forall i$, always(rose($BtoS_ACK(i)_o$) $\rightarrow SLC_g{=}i$)
- property $G10$ is always((fell($RtoB_ACK(0)_o$) or fell($RtoB_ACK(1)_o$)) $\rightarrow DEQ_g$)
- property $G11_1$ is always(($FULL_o$ and not DEQ_o) $\rightarrow$ not ENQ_g)
- property $G11_2$ is always($EMPTY_o \rightarrow$ not DEQ_g)
- property $G12$ is always(not $EMPTY_o \rightarrow$ eventually! DEQ_g)

Annexe D

Spécification du contrôleur CONMAX-IP pour 4 maîtres et 3 esclaves

– property **Master0ToSlave0** is
  always(
    (($m0_cyc_i_o$ and $m0_addr_i_o$="0000") and
    (not $m1_cyc_i_o$ or $m1_addr_i_o$/="0000") and
    not $s0_cyc_o_o$)
   → next[1]((
     ($s0_addr_o_g$=$m0_addr_i_o$ and $s0_data_o_g$=$m0_data_i_o$) and
     ($s0_sel_o_g$=$m0_sel_i_o$ and $s0_we_o_g$=$m0_we_i_o$) and
     ($s0_stb_o_g$=$m0_stb_i_o$ and $s0_cyc_o_g$=$m0_cyc_i_o$) and
     ($m0_data_o_g$=$s0_data_i_o$ and $m0_ack_o_g$=$s0_ack_i_o$) and
     ($m0_rty_o_g$=$s0_rty_i_o$ and $m0_err_o_g$=$s0_err_i_o$))
    until
     ($m0_ack_o_o$ or $m0_rty_o_o$ or $m0_err_o_o$))
  );
– property **Master0ToSlave1** is
  always(
    (($m0_cyc_i_o$ and $m0_addr_i_o$="0001") and
    (not $m1_cyc_i_o$ or $m1_addr_i_o$/="0001") and
    not $s1_cyc_o_o$)
   → next[1]((
     ($s1_addr_o_g$=$m0_addr_i_o$ and $s1_data_o_g$=$m0_data_i_o$) and
     ($s1_sel_o_g$=$m0_sel_i_o$ and $s1_we_o_g$=$m0_we_i_o$) and
     ($s1_stb_o_g$=$m0_stb_i_o$ and $s1_cyc_o_g$=$m0_cyc_i_o$) and
     ($m0_data_o_g$=$s1_data_i_o$ and $m0_ack_o_g$=$s1_ack_i_o$) and
     ($m0_rty_o_g$=$s1_rty_i_o$ and $m0_err_o_g$=$s1_err_i_o$))
    until
     ($m0_ack_o_o$ or $m0_rty_o_o$ or $m0_err_o_o$))
  );
– property **Master0ToSlave2** is
  always(
    (($m0_cyc_i_o$ and $m0_addr_i_o$="0010") and
    (not $m1_cyc_i_o$ or $m1_addr_i_o$/="0010") and

not $s2_cyc_o_o$)
$\longrightarrow$ next[1]((
($s2_addr_o_g$=$m0_addr_i_o$ and $s2_data_o_g$=$m0_data_i_o$) and
($s2_sel_o_g$=$m0_sel_i_o$ and $s2_we_o_g$=$m0_we_i_o$) and
($s2_stb_o_g$=$m0_stb_i_o$ and $s2_cyc_o_g$=$m0_cyc_i_o$) and
($m0_data_o_g$=$s1_data_i_o$ and $m0_ack_o_g$=$s2_ack_i_o$) and
($m0_rty_o_g$=$s2_rty_i_o$ and $m0_err_o_g$=$s2_err_i_o$))
until
($m0_ack_o_o$ or $m0_rty_o_o$ or $m0_err_o_o$))
);

- property **Master1ToSlave0** is
always(
(($m1_cyc_i_o$ and $m1_addr_i_o$="0000") and
(not $m1_cyc_i_o$ or $m0_addr_i_o$/="0000") and
not $s0_cyc_o_o$)
$\longrightarrow$ next[1]((
($s0_addr_o_g$=$m1_addr_i_o$ and $s0_data_o_g$=$m0_data_i_o$) and
($s0_sel_o_g$=$m1_sel_i_o$ and $s0_we_o_g$=$m1_we_i_o$) and
($s0_stb_o_g$=$m1_stb_i_o$ and $s0_cyc_o_g$=$m1_cyc_i_o$) and
($m1_data_o_g$=$s0_data_i_o$ and $m1_ack_o_g$=$s0_ack_i_o$) and
($m1_rty_o_g$=$s0_rty_i_o$ and $m1_err_o_g$=$s0_err_i_o$))
until
($m1_ack_o_o$ or $m1_rty_o_o$ or $m1_err_o_o$))
);

- property **Master1ToSlave1** is
always(
(($m1_cyc_i_o$ and $m1_addr_i_o$="0001") and
(not $m1_cyc_i_o$ or $m0_addr_i_o$/="0001") and
not $s1_cyc_o_o$)
$\longrightarrow$ next[1]((
($s1_addr_o_g$=$m1_addr_i_o$ and $s1_data_o_g$=$m1_data_i_o$) and
($s1_sel_o_g$=$m1_sel_i_o$ and $s1_we_o_g$=$m1_we_i_o$) and
($s1_stb_o_g$=$m1_stb_i_o$ and $s1_cyc_o_g$=$m1_cyc_i_o$) and
($m1_data_o_g$=$s1_data_i_o$ and $m1_ack_o_g$=$s1_ack_i_o$) and
($m0_rty_o_g$=$s1_rty_i_o$ and $m1_err_o_g$=$s1_err_i_o$))
until
($m1_ack_o_o$ or $m1_rty_o_o$ or $m1_err_o_o$))
);

- property **Master1ToSlave2** is
always(
(($m1_cyc_i_o$ and $m1_addr_i_o$="0010") and
(not $m1_cyc_i_o$ or $m0_addr_i_o$/="0010") and
not $s2_cyc_o_o$)
$\longrightarrow$ next[1]((
($s2_addr_o_g$=$m1_addr_i_o$ and $s2_data_o_g$=$m1_data_i_o$) and
($s2_sel_o_g$=$m1_sel_i_o$ and $s2_we_o_g$=$m1_we_i_o$) and
($s2_stb_o_g$=$m1_stb_i_o$ and $s2_cyc_o_g$=$m1_cyc_i_o$) and

($m1_data_o_g$=$s2_data_i_o$ and $m1_ack_o_g$=$s2_ack_i_o$) and
($m0_rty_o_g$=$s2_rty_i_o$ and $m1_err_o_g$=$s2_err_i_o$))
until
($m1_ack_o_o$ or $m1_rty_o_o$ or $m1_err_o_o$))
);

- property **Master2ToSlave0** is
always(
(($m2_cyc_i_o$ and $m2_addr_i_o$="0000") and
(not($m0_cyc_i_o$) or $m0_addr_i_o$/="0000") and
(not($m1_cyc_i_o$) or $m1_addr_i_o$/="0000") and
(not($m3_cyc_i_o$) or $m3_addr_i_o$/="0000") and not($s0_cyc_o_o$))
→next[1]((
($s0_addr_o_g$=$m2_addr_i_o$ and $s0_data_o_g$=$m2_data_i_o$) and
($s0_cyc_o_g$=$m2_cyc_i_o$ and $s0_sel_o_g$=$m2_sel_i_o$) and
($s0_we_o_g$=$m2_we_i_o$ and $s0_stb_o_g$=$m2_stb_i_o$) and
($m2_data_o_g$=$s0_data_i_o$ and $m2_ack_o_g$=$s0_ack_i_o$) and
($m2_rty_o_g$=$s0_rty_i_o$ and $m2_err_o_g$=$s0_err_i_o$)
until
($m2_ack_o_o$ or ($m2_rty_o_o$ or $m2_err_o_o$)))
);

- property **Master2ToSlave1** is
always(
(($m2_cyc_i_o$ and $m2_addr_i_o$="0001") and
(not($m0_cyc_i_o$) or $m0_addr_i_o$/="0001") and
(not($m1_cyc_i_o$) or $m1_addr_i_o$/="0001") and
(not($m3_cyc_i_o$) or $m3_addr_i_o$/="0001") and not($s1_cyc_o_o$))
→next[1]((
($s1_addr_o_g$=$m2_addr_i_o$ and $s1_data_o_g$=$m2_data_i_o$) and
($s1_cyc_o_g$=$m2_cyc_i_o$ and $s1_sel_o_g$=$m2_sel_i_o$) and
($s1_we_o_g$=$m2_we_i_o$ and $s1_stb_o_g$=$m2_stb_i_o$) and
($m2_data_o_g$=$s1_data_i_o$ and $m2_ack_o_g$=$s1_ack_i_o$) and
($m2_rty_o_g$=$s1_rty_i_o$ and $m2_err_o_g$=$s1_err_i_o$)
until
($m2_ack_o_o$ or ($m2_rty_o_o$ or $m2_err_o_o$)))
);

- property **Master2ToSlave2** is
always(
(($m2_cyc_i_o$ and $m2_addr_i_o$="0010") and
(not($m0_cyc_i_o$) or $m0_addr_i_o$/="0010") and
(not($m1_cyc_i_o$) or $m1_addr_i_o$/="0010") and
(not($m3_cyc_i_o$) or $m3_addr_i_o$/="0010") and not($s2_cyc_o_o$))
→next[1]((
($s2_addr_o_g$=$m2_addr_i_o$ and $s2_data_o_g$=$m2_data_i_o$) and
($s2_cyc_o_g$=$m2_cyc_i_o$ and $s2_sel_o_g$=$m2_sel_i_o$) and
($s2_we_o_g$=$m2_we_i_o$ and $s2_stb_o_g$=$m2_stb_i_o$) and
($m2_data_o_g$=$s2_data_i_o$ and $m2_ack_o_g$=$s2_ack_i_o$) and
($m2_rty_o_g$=$s2_rty_i_o$ and $m2_err_o_g$=$s2_err_i_o$)

until
($m2_ack_o_o$ or ($m2_rty_o_o$ or $m2_err_o_o$)))
);

- property **Master3ToSlave0** is
always(
(($m3_cyc_i_o$ and $m3_addr_i_o$="0000") and
(not $m0_cyc_i_o$ or $m0_addr_i_o$/="0000") and
(not $m1_cyc_i_o$ or $m1_addr_i_o$/="0000") and
(not $m2_cyc_i_o$ or $m2_addr_i_o$/="0000") and not($s0_cyc_o_o$))
→next[1](((
($s0_addr_o_g$=$m3_addr_i_o$ and $s0_data_o_g$=$m2_data_i_o$ and
($s0_cyc_o_g$=$m3_cyc_i_o$)) and ($s0_sel_o_g$=$m3_sel_i_o$ and
$s0_we_o_g$=$m3_we_i_o$) and ($s0_stb_o_g$=$m3_stb_i_o$ and
$m3_data_o_g$=$s0_data_i_o$) and ($m3_ack_o_g$=$s0_ack_i_o$ and
$m3_rty_o_g$=$s0_rty_i_o$) and $m3_err_o_g$=$s0_err_i_o$)
until
($m3_ack_o_o$ or $m3_rty_o_o$ or $m3_err_o_o$)))
);

- property **Master3ToSlave1** is
always(
(($m3_cyc_i_o$ and $m3_addr_i_o$="0001") and
(not $m0_cyc_i_o$ or $m0_addr_i_o$/="0001") and
(not $m1_cyc_i_o$ or $m1_addr_i_o$/="0001") and
(not $m2_cyc_i_o$ or $m2_addr_i_o$/="0001") and not($s1_cyc_o_o$))
→next[1](((
($s1_addr_o_g$=$m3_addr_i_o$ and $s1_data_o_g$=$m2_data_i_o$ and
($s1_cyc_o_g$=$m3_cyc_i_o$)) and ($s1_sel_o_g$=$m3_sel_i_o$ and
$s1_we_o_g$=$m3_we_i_o$) and ($s1_stb_o_g$=$m3_stb_i_o$ and
$m3_data_o_g$=$s1_data_i_o$) and ($m3_ack_o_g$=$s1_ack_i_o$ and
$m3_rty_o_g$=$s1_rty_i_o$) and $m3_err_o_g$=$s1_err_i_o$)
until
($m3_ack_o_o$ or $m3_rty_o_o$ or $m3_err_o_o$)))
);

- property **Master3ToSlave2** is always(
(($m3_cyc_i_o$ and $m3_addr_i_o$="0010") and
(not $m0_cyc_i_o$ or $m0_addr_i_o$/="0010") and
(not $m1_cyc_i_o$ or $m1_addr_i_o$/="0010") and
(not $m2_cyc_i_o$ or $m2_addr_i_o$/="0010") and not($s2_cyc_o_o$))
→next[1](((
($s2_addr_o_g$=$m3_addr_i_o$ and $s2_data_o_g$=$m2_data_i_o$ and
($s2_cyc_o_g$=$m3_cyc_i_o$)) and ($s2_sel_o_g$=$m3_sel_i_o$ and
$s2_we_o_g$=$m3_we_i_o$) and ($s2_stb_o_g$=$m3_stb_i_o$ and
$m3_data_o_g$=$s2_data_i_o$) and ($m3_ack_o_g$=$s2_ack_i_o$ and
$m3_rty_o_g$=$s2_rty_i_o$) and $m3_err_o_g$=$s2_err_i_o$)
until
($m3_ack_o_o$ or $m3_rty_o_o$ or $m3_err_o_o$)))
);

- property **InitM01RRS0** is
  (
  $(((m0_cyc_i_o$ and $m1_cyc_i_o)$ and not $s0_cyc_o_o)$ and
  (not $Grant_0_o$ and not $Grant_1_o)$ and
  $(m0_addr_i_o$="0000" and $m1_addr_i_o$="0000"))
  →next[1](((
  $(s0_addr_o_g=m0_addr_i_o$ and $s0_data_o_g=m0_data_i_o)$ and
  $(s0_sel_o_g=m0_sel_i_o$ and $s0_we_o_g=m0_we_i_o)$ and
  $(s0_stb_o_g=m0_stb_i_o$ and $s0_cyc_o_g=m0_cyc_i_o)$ and
  $(m0_data_o_g=s0_data_i_o$ and $m0_ack_o_g=s0_ack_i_o)$ and
  $(m0_rty_o_g=s0_rty_i_o$ and $m0_err_o_g=s0_err_i_o))$
  until
  $(m0_ack_o_o$ or $m0_rty_o_o$ or $m0_err_o_o))$ and
  $((((Grant_1_g$ and not $Grant_0_g)$
  until
  (rose($(m0_cyc_i_o$ and $(m1_cyc_i_o))$ and
  $(m0_addr_i_o$="0000" and $m1_addr_i_o$="0000")))))))
  );
- property **InitM01RRS1** is
  (
  $(((m0_cyc_i_o$ and $m1_cyc_i_o)$ and not $s1_cyc_o_o)$ and
  (not $Grant_0_o$ and not $Grant_1_o)$ and
  $(m0_addr_i_o$="0001" and $m1_addr_i_o$="0001"))
  →next[1](((
  $(s1_addr_o_g=m0_addr_i_o$ and $s1_data_o_g=m0_data_i_o)$ and
  $(s1_sel_o_g=m0_sel_i_o$ and $s1_we_o_g=m0_we_i_o)$ and
  $(s1_stb_o_g=m0_stb_i_o$ and $s1_cyc_o_g=m0_cyc_i_o)$ and
  $(m0_data_o_g=s1_data_i_o$ and $m0_ack_o_g=s1_ack_i_o)$ and
  $(m0_rty_o_g=s1_rty_i_o$ and $m0_err_o_g=s1_err_i_o))$
  until
  $(m0_ack_o_o$ or $m0_rty_o_o$ or $m0_err_o_o))$ and
  $((((Grant_1_g$ and not $Grant_0_g)$
  until
  (rose($(m0_cyc_i_o$ and $(m1_cyc_i_o))$ and
  $(m0_addr_i_o$="0001" and $m1_addr_i_o$="0001")))))))
  );
- property **InitM01RRS2** is
  (
  $(((m0_cyc_i_o$ and $m1_cyc_i_o)$ and not $s2_cyc_o_o)$ and
  (not $Grant_0_o$ and not $Grant_1_o)$ and
  $(m0_addr_i_o$="0000" and $m1_addr_i_o$="0010"))
  →next[1](((
  $(s2_addr_o_g=m0_addr_i_o$ and $s2_data_o_g=m0_data_i_o)$ and
  $(s2_sel_o_g=m0_sel_i_o$ and $s2_we_o_g=m0_we_i_o)$ and
  $(s2_stb_o_g=m0_stb_i_o$ and $s2_cyc_o_g=m0_cyc_i_o)$ and
  $(m0_data_o_g=s2_data_i_o$ and $m0_ack_o_g=s2_ack_i_o)$ and
  $(m0_rty_o_g=s2_rty_i_o$ and $m0_err_o_g=s2_err_i_o))$

until
$(m0_ack_o_o$ or $m0_rty_o_o$ or $m0_err_o_o))$ and
$((((Grant_1_g$ and not $Grant_0_g)$
until
(rose$((m0_cyc_i_o$ and $(m1_cyc_i_o))$ and
$(m0_addr_i_o$="0010" and $m1_addr_i_o$="0010")))))))
);

- property **InitM23RRS0** is
(((
$(m2_cyc_i_o$ and $m3_cyc_i_o)$ and not$(s0_cyc_o_o))$ and
(not $Grant_2_o$ and not $Grant_3_o)$ and
$(m2_addr_i_o$="0000" and $m3_addr_i_o$="0000"))
→next[1](((
$(s0_addr_o_g=m2_addr_i_o$ and $s0_data_o_g=m2_data_i_o)$ and
$(s0_sel_o_g=m2_sel_i_o$ and $s0_we_o_g=m2_we_i_o)$ and
$(s0_stb_o_g=m2_stb_i_o$ and $s0_cyc_o_g=m2_cyc_i_o)$ and
$(m2_data_o_g=s0_data_i_o$ and $m2_ack_o_g=s0_ack_i_o)$ and
$(m2_rty_o_g=s0_rty_i_o$ and $m2_err_o_g=s0_err_i_o))$
until$(m2_ack_o_o$ or $m2_rty_o_o$ or $m2_err_o_o))$ and
$((((Grant_1_g$ and not $Grant_0_g)$
until
(rose$((m2_cyc_i_o$ and $(m3_cyc_i_o))$ and
$(m2_addr_i_o$="0000" and $m3_addr_i_o$="0000")))))))
);

- property **InitM23RRS1** is
(((
$(m2_cyc_i_o$ and $m3_cyc_i_o)$ and not$(s1_cyc_o_o))$ and
(not $Grant_2_o$ and not $Grant_3_o)$ and
$(m2_addr_i_o$="0001" and $m3_addr_i_o$="0001"))
→next[1](((
$(s1_addr_o_g=m2_addr_i_o$ and $s1_data_o_g=m2_data_i_o)$ and
$(s1_sel_o_g=m2_sel_i_o$ and $s1_we_o_g=m2_we_i_o)$ and
$(s1_stb_o_g=m2_stb_i_o$ and $s1_cyc_o_g=m2_cyc_i_o)$ and
$(m2_data_o_g=s1_data_i_o$ and $m2_ack_o_g=s1_ack_i_o)$ and
$(m2_rty_o_g=s1_rty_i_o$ and $m2_err_o_g=s1_err_i_o))$
until$(m2_ack_o_o$ or $m2_rty_o_o$ or $m2_err_o_o))$ and
$((((Grant_1_g$ and not $Grant_0_g)$
until
(rose$((m2_cyc_i_o$ and $(m3_cyc_i_o))$ and
$(m2_addr_i_o$="0001" and $m3_addr_i_o$="0001")))))))
);

- property **InitM23RRS2** is
(((
$(m2_cyc_i_o$ and $m3_cyc_i_o)$ and not$(s2_cyc_o_o))$ and
(not $Grant_2_o$ and not $Grant_3_o)$ and
$(m2_addr_i_o$="0010" and $m3_addr_i_o$="0010"))
→next[1](((

($s2_addr_o_g$=$m2_addr_i_o$ and $s2_data_o_g$=$m2_data_i_o$) and
($s2_sel_o_g$=$m2_sel_i_o$ and $s2_we_o_g$=$m2_we_i_o$) and
($s2_stb_o_g$=$m2_stb_i_o$ and $s2_cyc_o_g$=$m2_cyc_i_o$) and
($m2_data_o_g$=$s2_data_i_o$ and $m2_ack_o_g$=$s2_ack_i_o$) and
($m2_rty_o_g$=$s2_rty_i_o$ and $m2_err_o_g$=$s2_err_i_o$))
until($m2_ack_o_o$ or $m2_rty_o_o$ or $m2_err_o_o$)) and
(((($Grant_1_g$ and not $Grant_0_g$)
until
(rose(($m2_cyc_i_o$ and ($m3_cyc_i_o$)) and
($m2_addr_i_o$="0010" and $m3_addr_i_o$="0010"))))))
);

- property **Master01_Slave0RR** is
always((
($m0_cyc_i_o$ and $m1_cyc_i_o$) and
($m0_addr_i_o$="0000" and $m1_addr_i_o$="0000") and
($Grant_0_c$ and not($s0_cyc_o_o$)))
→next[1]((((
($s0_addr_o_g$=$m0_addr_i_o$ and $s0_data_o_g$=$m0_data_i_o$) and
($s0_sel_o_g$=$m0_sel_i_o$ and $s0_we_o_g$=$m0_we_i_o$) and
($s0_stb_o_g$=$m0_stb_i_o$ and $m0_data_o_g$=$s0_data_i_o$) and
($m0_ack_o_g$=$s0_ack_i_o$ and $m0_rty_o_g$=$s0_rty_i_o$) and
($m0_err_o_g$=$s0_err_i_o$))
until((($m0_ack_o_o$ or $m0_rty_o_o$) or $m0_err_o_o$))) and
((((not($Grant_0_g$) and $Grant_1_g$))
until
(rose(($m0_cyc_i_o$ and $m1_cyc_i_o$) and
($m0_addr_i_o$="0000" and $m1_addr_i_o$="0000"))))))
);

- property **Master10_Slave0RR** is
always((
($m0_cyc_i_o$ and $m1_cyc_i_o$) and
($m0_addr_i_o$="0000" and $m1_addr_i_o$="0000") and
($Grant_1_o$ and not($s0_cyc_o_o$)))
→next[1]((((
($s0_addr_o_g$=$m1_addr_i_o$ and $s0_data_o_g$=$m1_data_i_o$) and
($s0_sel_o_g$=$m1_sel_i_o$ and $s0_we_o_g$=$m1_we_i_o$) and
($s0_stb_o_g$=$m1_stb_i_o$ and $m1_data_o_g$=$s0_data_i_o$) and
($m1_ack_o_g$=$s0_ack_i_o$ and $m1_rty_o_g$=$s0_rty_i_o$) and
($m1_err_o_g$=$s0_err_i_o$))
until
((($m1_ack_o_o$ or $m1_rty_o_o$) or $m1_err_o_o$))) and
((((not($Grant_1_g$) and $Grant_0_g$))
until
(rose(($m0_cyc_i_o$ and $m1_cyc_i_o$) and
($m0_addr_i_o$="0000" and $m1_addr_i_o$="0000"))))))
);

- property **Master01_Slave1RR** is

always(
((*m0_cyc_i*$_o$ and *m1_cyc_i*$_o$) and
(*m0_addr_i*$_o$="0001" and *m1_addr_i*$_o$="0001") and
(*Grant_0*$_o$ and not(*s1_cyc_o*$_o$)))
→next[1]((((
(*s1_addr_o*$_g$=*m0_addr_i*$_o$ and
s1_data_o$_g$=*m0_data_i*$_o$) and (*s1_sel_o*$_g$=*m0_sel_i*$_o$ and
s1_we_o$_g$=*m0_we_i*$_o$) and (*s1_stb_o*$_g$=*m0_stb_i*$_o$ and
m0_data_o$_g$=*s1_data_i*$_o$) and (*m0_ack_o*$_g$=*s1_ack_i*$_o$ and
m0_rty_o$_g$=*s1_rty_i*$_o$) and (*m0_err_o*$_g$=*s1_err_i*$_o$))
until(((*m0_ack_o*$_o$ or *m0_rty_o*$_o$) or *m0_err_o*$_o$))) and
((((not(*Grant_0*$_g$) and *Grant_1*$_g$))
until
(rose((*m0_cyc_i*$_o$ and *m1_cyc_i*$_o$) and
(*m0_addr_i*$_o$="0001" and *m1_addr_i*$_o$="0001")))))))
);

– property **Master10_Slave1RR** is
always(
((*m0_cyc_i*$_o$ and *m1_cyc_i*$_o$) and
(*m0_addr_i*$_o$="0001" and *m1_addr_i*$_o$="0001") and
(*Grant_1*$_o$ and not(*s1_cyc_o*$_o$)))
→next[1]((((
(*s1_addr_o*$_g$=*m1_addr_i*$_o$ and *s1_data_o*$_g$=*m1_data_i*$_o$) and
(*s1_sel_o*$_g$=*m1_sel_i*$_o$ and *s1_we_o*$_g$=*m1_we_i*$_o$) and
(*s1_stb_o*$_g$=*m1_stb_i*$_o$ and *m1_data_o*$_g$=*s1_data_i*$_o$) and
(*m1_ack_o*$_g$=*s1_ack_i*$_o$ and *m1_rty_o*$_g$=*s1_rty_i*$_o$) and
(*m1_err_o*$_g$=*s1_err_i*$_o$))
until(((*m1_ack_o*$_o$ or *m1_rty_o*$_o$) or *m1_err_o*$_o$))) and
((((not(*Grant_1*$_g$) and *Grant_0*$_g$))
until
(rose((*m0_cyc_i*$_o$ and *m1_cyc_i*$_o$) and
(*m0_addr_i*$_o$="0001" and *m1_addr_i*$_o$="0001")))))))
);

– property **Master01_Slave2RR** is
always(
((*m0_cyc_i*$_o$ and *m1_cyc_i*$_o$) and
(*m0_addr_i*$_o$="0010" and *m1_addr_i*$_o$="0010") and
(*Grant_0*$_o$ and not(*s2_cyc_o*$_o$)))
→next[1]((((
(*s2_addr_o*$_g$=*m0_addr_i*$_o$ and *s2_data_o*$_g$=*m0_data_i*$_o$) and
(*s2_sel_o*$_g$=*m0_sel_i*$_o$ and *s2_we_o*$_g$=*m0_we_i*$_o$) and
(*s2_stb_o*$_g$=*m0_stb_i*$_o$ and *m0_data_o*$_g$=*s2_data_i*$_o$) and
(*m0_ack_o*$_g$=*s2_ack_i*$_o$ and *m0_rty_o*$_g$=*s2_rty_i*$_o$) and
(*m0_err_o*$_g$=*s2_err_i*$_o$))
until(((*m0_ack_o*$_o$ or *m0_rty_o*$_o$) or *m0_err_o*$_o$))) and
((((not(*Grant_0*$_g$) and *Grant_1*$_g$))
until

(rose(($m0_cyc_i_o$ and $m1_cyc_i_o$) and
($m0_addr_i_o$="0010" and $m1_addr_i_o$="0010")))))))
);

– property **Master10_Slave2RR** is
always(
(($m0_cyc_i_o$ and $m1_cyc_i_o$) and
($m0_addr_i_o$="0010" and $m1_addr_i_o$="0010") and
($Grant_1_o$ and not($s2_cyc_o_o$)))
→next[1]((((
($s2_addr_o_g$=$m1_addr_i_o$ and $s2_data_o_g$=$m1_data_i_o$) and
($s2_sel_o_g$=$m1_sel_i_o$ and $s2_we_o_g$=$m1_we_i_o$) and
($s2_stb_o_g$=$m1_stb_i_o$ and $m1_data_o_g$=$s2_data_i_o$) and
($m1_ack_o_g$=$s1_ack_i_o$ and $m1_rty_o_g$=$s2_rty_i_o$) and
($m1_err_o_g$=$s2_err_i_o$))
until((($m1_ack_o_o$ or $m1_rty_o_o$) or $m1_err_o_o$))) and
((((not($Grant_1_g$) and $Grant_0_g$))
until
(rose(($m0_cyc_i_o$ and $m1_cyc_i_o$) and
($m0_addr_i_o$="0010" and $m1_addr_i_o$="0010")))))))
);

– property **Master23_Slave0RR** is
always(
(($m2_cyc_i_o$ and $m3_cyc_i_o$) and
($m2_addr_i_o$="0000" and $m3_addr_i_o$="0000") and
($Grant_2_o$ and not($s0_cyc_o_o$)))
→next[1]((((
($s0_addr_o_g$=$m2_addr_i_o$ and $s0_data_o_g$=$m2_data_i_o$) and
($s0_sel_o_g$=$m2_sel_i_o$ and $s0_we_o_g$=$m2_we_i_o$) and
($s0_stb_o_g$=$m2_stb_i_o$ and $m2_data_o_g$=$s0_data_i_o$) and
($m2_ack_o_g$=$s0_ack_i_o$ and $m2_rty_o_g$=$s0_rty_i_o$) and
($m2_err_o_g$=$s0_err_i_o$))
until((($m2_ack_o_o$ or $m2_rty_o_o$) or $m2_err_o_o$))) and
((((not($Grant_2_g$) and $Grant_3_g$))
until
(rose(($m2_cyc_i_o$ and $m3_cyc_i_o$) and
($m2_addr_i_o$="0000" and $m3_addr_i_o$="0000")))))))
);

– property **Master32_Slave0RR** is
always(
(($m2_cyc_i_o$ and $m3_cyc_i_o$) and
($m2_addr_i_o$="0000" and $m3_addr_i_o$="0000") and
($Grant_3_o$ and not($s0_cyc_o_o$)))
→next[1]((((
($s0_addr_o_g$=$m3_addr_i_o$ and $s0_data_o_g$=$m2_data_i_o$) and
($s0_sel_o_g$=$m3_sel_i_o$ and $s0_we_o_g$=$m3_we_i_o$) and
($s0_stb_o_g$=$m3_stb_i_o$ and $m3_data_o_g$=$s0_data_i_o$) and
($m3_ack_o_g$=$s0_ack_i_o$ and $m3_rty_o_g$=$s0_rty_i_o$) and

$(m3_err_o_g=s0_err_i_o))$
until($((m3_ack_o_o$ or $m3_rty_o_o)$ or $m3_err_o_o)))$ and
$(((($not$(Grant_3_g)$ and $Grant_2_g))$
until
(rose$((m2_cyc_i_o$ and $m3_cyc_i_o)$ and
$(m2_addr_i_o=$"0000" and $m3_addr_i_o=$"0000")))))))
);

- property **Master23_Slave1RR** is
always(
$((m2_cyc_i_o$ and $m3_cyc_i_o$) and
$(m2_addr_i_o=$"0001" and $m3_addr_i_o=$"0001") and
$(Grant_2_o$ and not$(s1_cyc_o_o)))$
$\rightarrow$next[1]((((
$(s1_addr_o_g=m2_addr_i_o$ and $s1_data_o_g=m2_data_i_o)$ and
$(s1_sel_o_g=m2_sel_i_o$ and $s1_we_o_g=m2_we_i_o)$ and
$(s1_stb_o_g=m2_stb_i_o$ and $m2_data_o_g=s1_data_i_o)$ and
$(m2_ack_o_g=s1_ack_i_o$ and $m2_rty_o_g=s1_rty_i_o)$ and
$(m2_err_o_g=s1_err_i_o))$
until($((m2_ack_o_o$ or $m2_rty_o_o)$ or $m2_err_o_o)))$ and
$(((($not$(Grant_2_g)$ and $Grant_3_g))$
until
(rose$((m2_cyc_i_o$ and $m3_cyc_i_o)$ and
$(m2_addr_i_o=$"0001" and $m3_addr_i_o=$"0001")))))))
);

- property **Master32_Slave1RR** is
always(
$((m2_cyc_i_o$ and $m3_cyc_i_o$) and
$(m2_addr_i_o=$"0001" and $m3_addr_i_o=$"0001") and
$(Grant_3_o$ and not$(s1_cyc_o_o)))$
$\rightarrow$next[1]((((
$(s1_addr_o_g=m3_addr_i_o$ and $s1_data_o_g=m2_data_i_o)$ and
$(s1_sel_o_g=m3_sel_i_o$ and $s1_we_o_g=m3_we_i_o)$ and
$(s1_stb_o_g=m3_stb_i_o$ and $m3_data_o_g=s1_data_i_o)$ and
$(m3_ack_o_g=s1_ack_i_o$ and $m3_rty_o_g=s1_rty_i_o)$ and
$(m3_err_o_g=s1_err_i_o))$
until($((m3_ack_o_o$ or $m3_rty_o_o)$ or $m3_err_o_o)))$ and
$(((($not$(Grant_3_g)$ and $Grant_2_g))$
until
(rose$((m2_cyc_i_o$ and $m3_cyc_i_o)$ and
$(m2_addr_i_o=$"0001" and $m3_addr_i_o=$"0001")))))))
);

- property **Master23_Slave2RR** is
always(
$((m2_cyc_i_o$ and $m3_cyc_i_o$) and
$(m2_addr_i_o=$"0010" and $m3_addr_i_o=$"0010") and
$(Grant_2_o$ and not$(s2_cyc_o_o)))$
$\rightarrow$next[1]((((

($s2_addr_o_g$=$m2_addr_i_o$ and $s2_data_o_g$=$m2_data_i_o$) and
($s2_sel_o_g$=$m2_sel_i_o$ and $s2_we_o_g$=$m2_we_i_o$) and
($s2_stb_o_g$=$m2_stb_i_o$ and $m2_data_o_g$=$s2_data_i_o$) and
($m2_ack_o_g$=$s2_ack_i_o$ and $m2_rty_o_g$=$s2_rty_i_o$) and
($m2_err_o_g$=$s2_err_i_o$))
until((($m2_ack_o_o$ or $m2_rty_o_o$) or $m2_err_o_o$))) and
((((not($Grant_2_g$) and $Grant_3_g$))
until
(rose(($m2_cyc_i_o$ and $m3_cyc_i_o$) and
($m2_addr_i_o$="0010" and $m3_addr_i_o$="0010")))))))
);

– property **Master32_Slave2RR** is
always(
(($m2_cyc_i_o$ and $m3_cyc_i_o$) and
($m2_addr_i_o$="0010" and $m3_addr_i_o$="0010") and
($Grant_3_o$ and not($s2_cyc_o_o$)))
→next[1]((((
($s2_addr_o_g$=$m3_addr_i_o$ and $s2_data_o_g$=$m2_data_i_o$) and
($s2_sel_o_g$=$m3_sel_i_o$ and $s2_we_o_g$=$m3_we_i_o$) and
($s2_stb_o_g$=$m3_stb_i_o$ and $m3_data_o_g$=$s2_data_i_o$) and
($m3_ack_o_g$=$s2_ack_i_o$ and $m3_rty_o_g$=$s2_rty_i_o$) and
($m3_err_o_g$=$s2_err_i_o$))
until((($m3_ack_o_o$ or $m3_rty_o_o$) or $m3_err_o_o$))) and
((((not($Grant_3_g$) and $Grant_2_g$))
until
(rose(($m2_cyc_i_o$ and $m3_cyc_i_o$) and
($m2_addr_i_o$="0010" and $m3_addr_i_o$="0010")))))))
);

Bibliographie

[ABBSV00] A. Aziz, F. Balarin, R-K. Brayton, and A-L. Sangiovanni-Vincentelli. Sequential synthesis using S1S. In *IEEE Trans. on CAD of Integrated Circuits and Systems*, volume 19 - 10, pages 1149–1162, 2000.

[ABM98] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Procedings of the 1st IEEE International Conference on Formal Engineering Methods: ICFEM'98*, page 46, 1998.

[BBCL06] S. Bayliss, C-S. Bouganis, George-A. Constantinides, and W. Luk. An FPGA implementation of the simplex algorithm. In *Proceedings of the IEEE International Conference on Field Programmable Technology: FPT'06*, pages 49–56, 2006.

[BBM91] An experimental comparison of different approaches to ROM BIST. In *Proceedings of the 91th Advanced Computer Technology, Reliable Systems and Applications: CompEuro'91*, pages 567–571, May 1991.

[BCC+03] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[BCE+04] R. Bloem, R. Cavada, C. Eisner, I. Pill, M. Roveri, and S. Semprini. Manual for property simulation and assurance tool (deliverable 1.2/4-5). Technical report, PROSYD Project, Jan. 2004.

[BCHN06] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale. *Verification Methodology Manual for SystemVerilog.* Number ISBN : 978-0-387-25556-9. Springer, 2006.

[BCZ06] M. Boulé, J-S. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *Proceedings of the 24th IEEE International Conference on Computer Design: ICCD'06*, Oct 2006.

[Ber06] V. Bertacco. *Scalable Hardware Verification with Symbolic Simulation.* Springer Science + Business Media, Jan. 2006.

[BGJ+07a] R. Bloem, S. Galler, B. Jobstman, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run : Hardware from PSL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 190(4):3–16, 2007.

[BGJ+07b] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, and A. Pnueli. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *Proceedings of the conference on Design, automation and test*

in Europe: DATE '07, pages 1188–1193, San Jose, CA, USA, 2007. EDA Consortium.

[BK95] S. Boubezari and B. Kaminska. A deterministic built-in self-test generator based on cellular automata structures. In IEEE, editor, *IEEE Transactions on Computer*, volume 44, June 1995.

[BLOF06] D. Borrione, M. Liu, P. Ostier, and L. Fesquet. *Applications of Specification and Design Languages for SoCs Selected papers from FDL 2005*, chapter PSL-based online monitoring of digital systems, pages 5–22. Springer Netherlands, Sep 2006.

[boo] The boolsolve tool: http://freshmeat.net/projects/bool-expr-solve/.

[Bou08] M. Boulé. *Assertion-Checker Synthesis for Hardware Verification, In-Circuit Debugging and On-Line Monitoring.* PhD thesis, Department of Electrical and Computer Engineering McGill University - Montréal, February 2008.

[BOY01] P. E. Black, V. Okun, and Y. Yesha. Mutation of model checker specifications for test generation and evaluation. pages 14–20, 2001.

[BZ05] M. Boulé and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *Proceedings of the 23rd IEEE International Conference on Computer Design: ICCD'05*, pages 221–228. IEEE Computer Society, 2005.

[BZ06] M. Boulé and Z. Zilic. Efficient automata-based assertion-checker synthesis of PSL properties. In *Proceedings of IEEE International High Level Design Validation and Test Workshop: HLDVT'06*, Nov 2006.

[Cal05] J.R. Calamé. Specification-based test generation with TGV. Technical Report R0508, Centrum voor Wsikunde en Informatica, may 2005.

[CCPSR97] S. Chiusano, F. Corno, P. Prinetto, and M. Sonza Reorda. Cellular automata for deterministic sequential test pattern generation. In *Proceedings of the 15th IEEE VLSI Test Symposium: VTS '97*, page 60, Washington, DC, USA, 1997. IEEE Computer Society.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CHBW04] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Logic of constraints: A quantitative performance and functional constraint formalism. In IEEE, editor, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 23, pages 1243 – 1255, August 2004.

[Cim08] A. Cimatti. Beyond boolean sat: Satisfiability modulo theories. In *Proceedings of the 9th International Workshop on Discrete Event Systems: WODES'08*, pages 68–73, May 2008.

[Cla07] K. Claessen. A coverage analysis for safety property lists. In *Proceedings of the 7th Conference on Formal Methods in Computer Aided Design FMCAD'07*, pages 139–145, Nov. 2007.

[CLLV07] Ray C. C. Cheung, Dong-U Lee, Wayne Luk, and John D. Villasenor. Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method. In *Proceedings of the IEEE Transac-*

tions on Very Large Scale Integration Systems, volume 15, pages 952–962, Piscataway, NJ, USA, 2007. IEEE Educational Activities Department.

[CM04] K. Claessen and J. Martensson. An operational semantics for weak PSL. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design: FMCAD'04, LNCS 3312*, pages 337–351. Springer Verlag, 2004.

[Cor05] Altera Corporation. *Nios Development Board - Cyclone II Edition Reference Manual.* Altera, 2005.

[CRS99] F. Corno, M. Sonza Reorda, and G. Squillero. High quality test pattern generation for RT-level VHDL descriptions. In *Proceedings of the 2nd International Workshop on Microprocessor Test and Verification Common Challenges and Solutions: MTV'99*, Sept. 1999.

[CRST06] A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta. From PSL to NBA: a modular symbolic encoding. In *Proceedings of the 6th Conference on Formal Methods in Computer Aided Design FMCAD'06*, pages 125–133, 2006.

[CSRS04] F. Corno, E. Sánchez, M. Sonza Reorda, and G. Squillero. Automatic test program generation: A case study. *IEEE Design & Test of Computers*, 21(2):102–109, 2004.

[Dan98] G. Dantzig. *Linear Programming and Extensions.* Princeton University Press - ISBN=0691059136, August 1998.

[die] http://www.stat.fsu.edu/pub/diehard/.

[DTT08] E. Dubrova, M. Teslenko, and H. Tenhumen. On analysis and synthesis of (n,k)-non-linear feedback shift registers. In *Proceedings of the Conference on Design Automation and Test in Europe: DATE'08*, 2008.

[EBS+07] H. Eveking, M. Braun, M. Schickel, M. Schweikert, and V. Nimbler. Multi-level assertion-based design. In *Proceedings of the 5th ACM & IEEE International Conference on Formal Methods and Models for Co-Design MEMOCODE'07*, pages 85–87, Jun. 2007.

[Edv99] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science Engineering in Linköping: ICCSIT'99*, pages 21–28, Oct. 1999.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems).* Springer-Verlag New York, Inc. ISBN=0387353135, Secaucus, NJ, USA, 2006.

[FFS98] F. Ferrandi, F. Fummi, and D. Sciuto. Implicit test generation for behavioral VHDL models. In *Proceedings of the IEEE International Test Conference: ITC '98*, pages 587–596, oct 1998.

[FJR09] E. Filiot, N. Jin, and J-F. Raskin. An antichain algorithm for LTL realizability. In *Proceedings of the 21st International Conference on Computer Aided Verification: CAV'09*, pages 263–277, Jul. 2009.

[FKL03] H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design.* Kluwer Academic Publishers, ISBN=1402074980, Jun. 2003.

[FLT06] H. Foster, K. Larsen, and M. Turpin. Introduction to the new Accellera open verification library. In *Proceedings of the Conference on Design Automation and Test in Europe: DATE'06*, 2006.

[FWMG05] H. Foster, Y. Wolfshal, E. Marschner, and IEEE 1850 Work Group. *IEEE standard for property specification language PSL.* pub-IEEE-STD, pub-IEEE-STD:adr, Oct 2005.

[Gas05] E. Gascard. From sequential extended regular expressions to deterministic automata. Technical Report - TIMA Laboratory, Jul. 2005.

[GG05] S-V. Gheorghita and R. Grigore. Constructing checkers from PSL properties. In *Proceedings of the 15th International Conference on Control Systems and Computer Science: CSCS'05*, pages 757–762, May 2005.

[Gor03] M-J-C. Gordon. Validating the PSL/Sugar semantics using automated reasoning. In *Special Issue of the Formal Aspects of Computing Journal on Semantic Foundations of Engineering Design Languages*, 2003.

[Gra08] Mentor Graphics. Assertion coding guidelines with SVA, PSL, OVL, QVL and 0-in assertions, www.mentor.com/products/fv/0-in_fv/. Technical report, Feb. 2008.

[Gre04] D. Greaves. Automated hardware synthesis from formal specification using SAT solvers. In *Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping: RSP'04*, 2004.

[Her02] R. Herveille. WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores. Technical report, http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf, Sept 2002.

[HNV05] S. Heymans, D-V. Nieuwenborgh, and D. Vermeir. Synthesis from temporal specifications using preferred answer set programming. 3701:280–294, 2005.

[HW05] J. Havlicek and Y. Wolfshal. PSL and SVA: Two standard assertion languages addressing complementary engineering needs. In *Proceedings of the Design and Verification Conference: DVCon'05*, 2005.

[Ins96] Texas Instruments. What's an LFSR?, http://focus.ti.com/general/docs/, Dec. 1996.

[JB06] B. Jobstman and R. Bloem. Optimizations for LTL synthesis. In *Proceedings of the 6th Conference on Formal Methods in Computer Aided Design: FMCAD'06*, Nov. 2006.

[JBIF09] S. Jadcherla, J. Bergeron, Y. Inoue, and D. Flynn. *Verification Methodlogy Manual for Low-Power.* Synopsys, 2009.

[JJ05] C. Jard and T. Jéron. TGV: Theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.

[JJJ+01] C. Jard, T. Jensen, T. Jéron, JM. Jézéquel, S.Pickin, L. Van Aertrick, L. Du Bousquet, and Y. Ledru. Etat de l'art sur la synthèse (génération automatique) de test. Technical report, IRISA, September 11 2001.

[JLS09] S. Jha, R. Limaye, and S-A. Seshiaater. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In *Proceedings of the 21st International Conference on Computer Aided Verification: CAV'09*, pages 668–673, Jul. 2009.

[JNZ08] N. Jin, T. Ni, and J. Zhou. iPSL: An environment for IP-based specifications. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems: ICECCS'08*, 2008.

[JPJ97] J.F.Pradat-Peyre and J.Printz. Utilisation de contraintes pertinentes pour la génération automatique de tests, http://cedric.cnam.fr/publis/rc417.pdf, 1997.

[JRCU07] M. Jenihhin, J. Raik, A. Chepurov, and R. Ubar. Assertion checking with PSL and high-level decision diagrams. In *Proceedings of the 8th Workshop on RTL and High Level Testing: WRTLT'07*, Beijing, Oct. 2007.

[JS07] N. Jin and C. Shen. Dynamic verifying the properties of the simple subset of PSL. In *Proceedings of the International Workshop on Harnessing Theories for Tool Support in Software: TTSS'07*, Sept 2007.

[KDB99] M. Keim, N. Drechsler, and B. Becker. Combining GAs and symbolic methods for high quality tests of sequential circuits. In *Proceedings of the 1999 Conference on Asia South Pacific Design Automation: ASP-DAC'99*, pages 315–318. IEEE, 1999.

[KGN+09] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing testing with formal verification in Intel Core i7 processor execution engine validation. In *Proceedings of the 21st International Conference on Computer Aided Verification: CAV'09*, pages 414–429, Jul. 2009.

[KLM06] M-R. Krug, M-S. Lubasewski, and M-S. Moraes. Improving ATPG gate-level fault coverage by using test vectors generated from behavioral HDL descriptions. In *Proceedings of the 2006 IFIP International Conference on Very Large Scale Integration: VLSISOC'06*, pages 314–319, Oct. 2006.

[KPKG02] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 21, pages 1377–1394, Dec 2002.

[KS00] J-H. Kukula and T-R. Shiple. Building circuits from relations. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification: CAV'00*, volume 1855 of *Lecture Notes in Computer Science*, pages 113–123. Springer, 2000.

[LM00] Lijian L. and Yinghua M. An efficient BIST design using LFSR-ROM architecture. In *Proceedings of the 9th Asian Test Symposium: ATS '00*, page 386, Washington, DC, USA, 2000. IEEE Computer Society.

[MAB05] K. Morin-Allory and D. Borrione. A proof of correctness for the construction of property monitors. In *Proceedings of the 10th IEEE Intl. High Level Design Validation and Test Workshop: HLDVT'05*, Dec. 2005.

[MAB06] K. Morin-Allory and D. Borrione. Proven correct monitors from psl specifications. In *Proceedings of the Conference on Design, Automation and Test in Europe: DATE'06*, pages 1246–1251, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[MABBZ08] K. Morin-Allory, M. Boulé, D. Borrione, and Z. Zilic. Proving and disproving assertion rewrite rules by automated theorem proving. In *Proceedings of*

the IEEE International High Level Design Validation and Test Workshop: HLDVT'08, Nov. 2008.

[MADS07] D. Mathaikutty, S. Ahuja, A. Dingankar, and S. Shukla. Model-driven test generation for system level validation. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop: HLDVT'07*, pages 83–90, Nov. 2007.

[MAFRB07] K. Morin-Allory, L. Fesquet, B. Roustan, and D. Borrione. Asynchronous online-monitoring of logical and temporal assertions. In *Proceedings of the 10th Forum on Specification and Design Languages: FDL'07*, pages 286–290. ECSI, 2007.

[MAGB07] K. Morin-Allory, E. Gascard, and D. Borrione. Synthesis of property monitors for online fault detection. In *Journal of Circuits, Systems, and Computers (JCSC), Vol. 16 (6)*, Dec. 2007.

[MB08] L. De Moura and N. Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[MBO04] Y. Makris, I. Bayraktaroglu, and A. Orailoglu. Enhancing reliability of RTL controller-datapath circuits via invariant-based concurrent test. In *IEEE Transactions on Reliability (T. REL)*, volume 53-2, pages 269–278, 2004.

[McC76] T. McCabe. A software complexity measure. In *IEEE Trans. on Software Engineering*, volume 2-6, Dec. 1976.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, ISBN: 0-7923-9380-5, 1993.

[MO99] Y. Makris and A. Orailoglu. Property based RTL test justification and propagation analysis. In *Proceedings of the 6th International Test Synthesis Workshop: ITSW'99*, 1999.

[NADB08] K. Nepal, N. Alves, J. Dworak, and R-I. Bahar. Using implications for online error detection. In *Proceedings of the 6th IEEE International Test Conference: ITC'08.*, pages 1–10, Oct 2008.

[Öbe99] J. Öberg. *ProGram : A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols*. PhD thesis, Royal Institute of Technologoy - Department of Electronics, Eletronic System Design, Sweden, 1999.

[OBMAP09] F. Ouchet, D. Borrione, K. Morin-Allory, and L. Pierre. High-level symbolic simulation for automatic model extraction. In *Proceedings of the 12th IEEE Symposium on Design and Diagnostics of Electronic Systems: DDECS'09*, Apr. 2009.

[ÖKH96] J. Öberg, A. Kumar, and A. Hemani. Grammar-based hardware synthesis of data communication protocols. In *Proceedings of the 9th international symposium on System synthesis: ISSS'96*, pages 14–19, 1996.

[OMAB06] Y. Oddos, K. Morin-Allory, and D. Borrione. On-line test vector generation from temporal constraints written in PSL. In *Proceedings of the 14th IFIP/IEEEInternational Conference on Very Large Scale Integration System on Chip: VLSI SoC'06*, October 2006.

[OMAB08a] Y. Oddos, K. Morin-Allory, and D. Borrione. Assertion-based design with horus. In *Proceedings of the 6th ACM-IEEE International Conference on Formal Methods and Models for Codesign: MEMOCODE'2008*, Jun. 2008.

[OMAB08b] Y. Oddos, K. Morin-Allory, and D. Borrione. Assertion-based verification and on-line testing in HORUS. In *Proceedings of the 3rd International Design and Test Workshop: IDT'08*, Monastir, Dec. 2008.

[Orl09] M. Orlov. Optimized random number generation in an interval. In *Information Processing Letters*, volume 109, pages 722 – 725, 2009.

[Pao01] C. Paoli. *Validation de descriptions VHDL fondée sur des techniques issues du domaine du test de logiciels*. PhD thesis, Université de Corse, Dec. 2001.

[PBSD08] B. Pal, A. Banerjee, A. Sinha, and P. Dasgupta. Accelerating assertion coverage with adaptative testbenches. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 27 - 5, pages 967–972, May 2008.

[PF08] L. Pierre and L. Ferro. A tractable and fast method for monitoring systemC TLM specifications. In *IEEE Transactions on Computers*, volume 57, pages 1346–1356, 2008.

[PK00] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In *Proceedings of the International Conference on Computer Design: ICCD'00*, pages 459–464, 2000.

[PKBMF05] D. Pidan, S. Keidar-Barner, M. Moulin, and D. Fisman. Optimized algorithms for dynamic verification (deliverable 3.2/5). Technical report, PROSYD Project, 2005.

[PLBN05] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil. Synthesis of synchronous assertions with guarded atomic actions. In *Proceedings of the 4th ACM-IEEE International Conference on Formal Methods and Models for Codesign: MEMOCODE'05*, pages 15–24, Jul. 2005.

[PW85] N-H. Packard and S. Wolfram. Two-dimensional cellular automata. In *Journal of Statistical Physics*, volume 38, pages 901–946, Mar. 1985.

[Reg05] J. Regenberg. Synthesis of reactive systems. Master's thesis, Universität des Saarlandes, Dec. 2005.

[RMJ04] V. Rusu, H. Marchand, and T. Jéron. Verification and symbolic test generation for safety properties. Technical Report PI-1640, IRISA, August 2004.

[SB94] A. Seawright and F. Brewer. Clairvoyant: A synthesis system for production-based specification. *IEEE Trans. on VLSI*, 2(2):172–185, Jun 1994.

[Sch99] P. Schnoebelen. *Vérification de logiciels : Techniques et outils de model checking*. Vuibert, ISBN: 978-2711786466, 1999.

[SD02] K. Shimizu and D.-L. Dill. Deriving a simulation input generator and a coverage metric from a formal specification. In *Proceedings of the 39th annual Design Automation Conference: DAC '02*, pages 801–806, New York, NY, USA, 2002. ACM.

[SD05] R. Seater and G. Dennis. Automated test data generation with SAT'05. *Proceedings of the ACM SIGUCS FALL Conference: FALL '05*, 2005.

[SF07] S. Schewe and B. Finkbeiner. Bounded synthesis. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis: ATVA'07*, pages 474–488. Springer-Verlag, 2007.

[Shi02] K. Shimizu. *Writing, Verifying, and Exploiting Formal Specifications for Hardware Designs.* PhD thesis, Stanford University, Dept. of Electrical Engineering, Oct. 2002.

[Siw06] M. Siwinski. Incisive assertion library : Jump-start assertion-based coverage-driven verification. Technical report, May 2006.

[SJE06] S. Safari, A-H. Jahangir, and H. Esmaeilzadeh. A parameterized graph-based framework for high-level test synthesis. In *VLSI journal*, volume 39, pages 363–381, Amsterdam, The Netherlands, 2006. Elsevier Science Publishers B. V.

[SM02] R. Siegmund and D. Müller. Automatic synthesis of communication controller hardware from protocol specifications. In *IEEE Design & Test of Computers*, volume 19, pages 84–95, 2002.

[SMB+05] J. Srouji, S. Mehta, D. Brophy, K. Pieper, S. Sutherland, and IEEE 1800 Work Group. IEEE standard for systemverilog - unified hardware design, specification, and verification language. Technical report, pub-IEEE-STD:adr, Nov 2005.

[SNBE07a] M. Schickel, V. Nimbler, M. Braun, and H. Eveking. *Advances in Design and Specification Languages for Embedded Systems*, chapter An Efficient Synthesis Method for Property-Based Design in Formal Verification: On Consistency and Completeness of Property-Sets, pages 179–196. Number 978-1-4020-6149-3 (Online). Springer Netherlands, Jul. 2007.

[SNBE07b] M. Schickel, V. Nimbler, M. Braun, and H. Eveking. *An Efficient Synthesis Method for Property Based Design in Formal Verification*, chapter 11, pages 179 – 196. Springer Netherlands, 2007.

[SORSC01] N. Shankar, S. Owre, J. Rushby, and D. Stringer-Calvert. PVS prover guide. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 2001.

[SOSE08] M. Schickel, M. Oberkonig, M. Schweikert, and H. Eveking. *Embedded Systems Specification and Design Languages*, chapter A Case-Study in Property-Based Synthesis: Generating a Cache Controller from Property-Set, pages 271–275. Springer Netherlands, 2008.

[STJCS02] B. Shackleford, M. Tanaka, R. J-Carter, and G. Snider. High-performance cellular automata random number generators for embedded probabilistic computing systems. In *Proceedings of the 2002 NASA/NOD Conference on Evolvable Hardware: EH'02*, 2002.

[TA97] R-S. Tupuri and J-A. Abraham. A novel functional test generation method for processors using commercial ATPG. In *Proceedings of the 1997 IEEE International Test Conference ITC '97*, page 743, Washington, DC, USA, 1997. IEEE Computer Society.

[TBS04] D. Toma, D. Borrione, and G. Al Sammane. Combining several paradigms for circuit validation and verification. In *Proceedings of the International*

workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart devices CASSIS'04, volume 3362, pages 229–249, 2004.

[TKA99] R-S. Tupuri, A. Krishnamachary, and J-A. Abraham. Test generation for gigahertz processors using an automatic functional constraint extractor. In *Proceedings of the 36th Design Automation Conference DAC'99*, pages 647–652, 1999.

[TS05] T. Tuerk and K. Schneider. From PSL to LTL: A formal validation in HOL. In *Proceedings of the 18th Conference TPHOLs*, pages 342–357, 2005.

[US03] I. Ugarte and P. Sanchez. Functional vector generation for assertion-based verification at behavioral level using interval analysis. In *Proceedings of the 8th IEEE International High-Level Design Validation and Test Workshop, HLDVT'03.*, pages 102–107, Nov. 2003.

[Var06] M-Y. Vardi. From church and prior to PSL. *Proceedings of the Workshop on 25 Years of Model Checking, Federated Logic Conference: FLOC'06*, Aug. 2006.

[WB07] I. Wagner and V. Bertacco. Engineering trust with semantic guardians. In *Proceedings of the IEEE Conference on Design Automation and Test in Europe: DATE'07*, April 2007.

[WBA08] I. Wagner, V. Bertacco, and T. Austin. Using field-repairable control logic to correct design errors in microprocessors. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 27, pages 380 – 393, Feb 2008.

[WHM09] C-M. Wintersteiger, Y. Hamadi, and L. De Moura. A concurrent portfolio approach to SMT solving. In *Proceedings of the 21st International Conference on Computer Aided Verification: CAV'09*, pages 715–720, Jul. 2009.

[WM07] R. Ward and T. Molteno. Table of linear feedback shift registers. *Advances in Computers*, Oct. 2007.

[Wol83a] S. Wolfram. Cellular automata. In *Los Alamos Science*, volume 9, pages 2–21, 1983.

[Wol83b] S. Wolfram. Statistical mechanics of cellular automata. In *Review of Modern Physics*, volume 55, pages 601–644, Jul. 1983.

[Wol86] S. Wolfram. Random sequence generation by cellular automata. In *Advances in Mathematics*, volume 7, pages 123–169, 1986.

[WTFAK07] H. Waeselynck, P. Thévenod-Fosse, and O. Abdellatif-Kaddour. *Empirical Software Engineering*, volume 12, chapter Simulated Annealing Applied to Test Generation: Landscape Characterization Stopping Criteria, pages 35–63. Springer Netherlands, Feb. 2007.

[WWC05] Charles H.-P. Wen, Li-C. Wang, and K-T. Cheng. Simulation-based functionnal test generation for embedded processors. In *Proceedings of the 10th Annual IEEE International Workshop on High Level Design Validation and Test: HLDVT'05*, pages 3–10, 2005.

Index

A

ABV . 10, 11, 13, 16, 21, 22, 25, 108, 122, 150, 151
ACL2 6
Arbiter$_P$ 6–8, 10, 16, 18, 52, 126, 155

B

BoolSolve 65, 123

C

CA 73
CDT . . 8, 9, 18, 42, 74, 95, 103, 112, 127, 148, 156
conmax_ip . . 126, 136–138, 141, 143–148, 151, 152
CTL 5, 16, 17, 20, 35

F

FL . 18, 22, 38, 40, 42, 46, 51, 58, 75, 96, 98, 99
FoCs 22, 23, 26, 27
FPGA . . . 4, 23, 33, 38, 44, 58, 67, 68, 81, 108, 124, 127, 147

G

générateur 11–13, 23, 39, 44, 46–54, 57–68, 72–75, 78, 79, 83, 87, 91, 94, 95, 98–100, 102–106, 108, 110, 118, 120, 122–126, 139–141, 143–146, 150–152
générateur-étendu 94, 95, 98–100, 102–106, 108, 110, 118–120, 123, 126, 127, 141, 144, 152
GenBuf 127–132, 152

H

HOL 6, 110
Horus . . 12, 22, 23, 29, 38, 45, 46, 48, 51, 68, 69, 74, 75, 122–125, 142, 143, 145, 146, 150, 151

L

LFSR 69, 70, 73, 74, 126
LTL 5, 16, 17, 26–28, 32, 35, 110

M

MBAC . 22, 23, 26, 29, 45, 46, 61, 62, 75, 103, 150
moniteur 11–13, 23, 26–29, 39–49, 61–63, 68, 75, 95, 98–100, 102–105, 110, 111, 113–115, 118–120, 122–125, 137, 138, 143, 145, 150, 151
MyGen 22, 38, 39, 46, 54, 61, 63, 65–69, 74, 75, 103, 104, 106, 108, 123, 127, 150, 152, 153

P

Prim_Solver 85–87
PSL . 12, 16–23, 26, 27, 32, 34, 38–41, 44, 47, 61, 95, 98, 100, 102, 106, 108, 110, 112, 118, 120, 122–124, 128, 146, 148, 150, 151, 153
PSL$_{ss}$ . . xvi, 20, 21, 34, 35, 59, 90, 95, 98
PVS . . 6, 12, 110–113, 116, 118, 120, 152

R

Rand_Block$_{orlov}$ 74, 75, 88, 89
RAT 79, 99, 105

S

SERE . . 17–19, 21, 26, 27, 38, 46, 51–54, 58, 69, 75
Solver$_{err}$ 83, 84
Solver$_{sig}$ 83, 84

Solver_{sys2} 83, 87, 89–91, 99
Solver_bit_{sys1} 82–85, 91, 99, 103
$\text{Solver_vect}_{sys1}$ 82, 85, 90, 91, 99
Spec_{ctrl} 96, 103
Spec_cdt 95, 103, 108, 126, 127
SVA .. 12, 21–23, 26, 27, 34, 39, 150, 153
SyntHorus 95, 99, 103, 106, 108, 122, 124–127, 129, 131, 132, 146–148, 152, 153

V

VHDL ... 6, 16, 17, 20, 31, 34, 63, 65, 67, 84, 86, 104–106, 111, 126, 153

W

Wishbone 124, 133–136

Publications de l'Auteur

Conférences internationales avec comité de lecture, Sélection sur le papier complet

- Y. Oddos, K. Morin-Allory, D. Borrione, *Synthorus : Highly Efficient Automtic Synthesis from PSL to HDL*, Proceedings of the 17th IFIP/IEEE International Conference On Very Large Scale Integration VLSI SoC'09, Florianópolis - Brazil, Oct. 2009.

- Y. Oddos, M. Boulé, K. Morin-Allory, D. Borrione, Z. Zilic. *MYGEN : Automata-based on-line Test Generator for Assertion Based Verification*, Proceedings of the 19th Great Lakes Symposium on VLSI : GLSVLSI'09, Boston - USA, May 2009.

- Y. Oddos, K. Morin-Allory, D. Borrione. *Assertion Based Verification and on-line Testing in Horus*, 3rd International Design and Test Workshop : IDT'08, Monastir - Tunisia, Dec. 2008.

- Y. Oddos, K. Morin-Allory, D. Borrione. *Assertion Based Verification with Horus*, Proceedings of the 6th ACM-IEEE International Conference on Formal Methods and Models for Codesign : MEMOCODE'08, Anaheim - California, Jun. 2008.

- Y. Oddos, K. Morin-Allory, D. Borrione. *Prototyping Generators for on-line test vector generation based on PSL properties*, Proceedings of the Design and Diagnostics of Electronic Circuits and Systems (DDECS), IEEE, Krakow - Poland, Apr. 2007.

- Y. Oddos, K. Morin-Allory, D. Borrione. *On-Line Test Vector Generation from temporal regular expressions*, Proceedings of the International Workshop on System on Chip (IWSOC), IEEE, Cairo - Egypt, Dec 2006.

- Y. Oddos, K. Morin-Allory, D. Borrione. *On-line Test Vector Generation from Temporal Constraints Written in PSL*, Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration System on Chip : VLSI SoC'06, Nice - France, Oct. 2006.

Chapitre de livre

- D. Borrione, K. Morin-Allory, Y. Oddos, *Chapter 10 : Property-based Dynamic Verification and Test*, Heterogeneous Embedded Systems - Design Theory and Practice, Springer, To Appear in Feb. 2010.

Conférences nationales avec comité de lecture, Sélection sur le papier complet

– Y. Oddos. *Génération de vecteurs de test en ligne à partir de propriétés PSL*, Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM),Mai 2007.

Résumé

La vérification à base de propriétés (PBV) est devenue un élément essentiel des flots de conception pour supporter la vérification de circuits complexes. La vérification dynamique à base de propriétés connecte au circuit des moniteurs et des générateurs de test synthétisés à partir de propriétés pour construire de manière simple un environnement de test. Durant cette thèse une partie des travaux à consisté à développer une approche de synthèse de propriétés pour la génération de vecteurs de test. Dans ce contexte, les propriétés décrivent l'environnement du circuit sous test. Elles sont synthétisées en générateurs produisant des séquences de test respectant la propriété correspondante. Il est alors possible de spécifier et d'obtenir un modèle pour tout l'environnement du circuit. Alors que notre approche est modulaire, une méthode à base d'automates a été développée en collaboration avec l'université de McGill. La contribution la plus intéressante de cette thèse tiens dans la méthode qui a été mise en place pour synthétiser une spécification temporelle en un circuit correct par construction. Alors que les approches de l'état de l'art ont une complexité polynomiale, la nôtre est linéaire en la spécification. L'outil SyntHorus a été développé pour supporter cette méthode et synthétise en quelques secondes un circuit correct par construction à partir d'une spécification de plusieurs centaines de propriétés. La correction des générateurs et de la méthode de synthèse a été effectuée à l'aide du prouveur de théorème PVS. Les méthodes et outils développés durant cette thèse ont été validés, renforcés et transférés dans l'industrie grâce à plusieurs coopérations (Thalès Group, Dolphin Integration et ST-Microelectronics) et au projet ANR SFINCS.

Abstract

Property-Based Verification (PBV) has become a main stream part of industrial design flows. For large systems that defeat formal verification methods, dynamic verification is called on designs directly connected to test generators and signal observers that are compiled from the properties. The work achieved during this thesis aims at automatically creating test-benches for digital-designs verification using the PBV. We have developed an approach to achieve property synthesis for test-vector generation. In this context, properties describe the environment of a design under test. They are synthesized into generators which produce test sequences complying with the corresponding properties. It is then possible to specify the whole environment with temporal properties. A parallel method which is automata-based has been developed in McGill University. The most interesting part of the thesis lies in the method that has been developed to automatically synthesize specifications into VHDL hardware descriptions. While the state of the art approaches have, for the best of them, polynomial complexities $O(N^3)$, ours is linear in the specification operators. A tool called SyntHorus has been designed to automate the method. It shows that synthesis of complex specifications of hundred properties can be synthesized within few seconds into efficient hardware components. The correctness of the generators and the specification synthesis approach has been done with the PVS theorem prover. The methods and tools developed during the thesis have been tested, reinforced and transferred to industry through some cooperations (Thalès Group, Dolphin Integration and ST-Microelectronics) and the ANR project SFINCS.