SIMULATION SYMBOLIQUE DES CIRCUITS DÉCRITS AU NIVEAU ALGORITHMIQUE

G. Al-Sammane

To cite this version:

G. Al-Sammane. SIMULATION SYMBOLIQUE DES CIRCUITS DÉCRITS AU NIVEAU ALGORITHMIQUE. Micro et nanotechnologies/Microélectronique. Université Joseph-Fourier - Grenoble I, 2005. Français. NNT : . tel-00009776

HAL Id: tel-00009776
https://theses.hal.science/tel-00009776

Submitted on 19 Jul 2005

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.

UNIVERSITE JOSEPH FOURIER GRENOBLE 1

N° attribué par la bibliothèque
|__/__/__/__/__/__/__/__/__/__|

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Spécialité : Microélectronique

préparée au laboratoire **TIMA** dans le cadre de
l'Ecole Doctorale d'« Electronique, Electrotechnique, Automatique, Télécommunications, Signal »

présentée et soutenue publiquement
par

Ghiath AL SAMMANE

Le 18 Juillet 2005

SIMULATION SYMBOLIQUE DES CIRCUITS DÉCRITS AU NIVEAU ALGORITHMIQUE

JURY

M. Ahmed Amine Jerraya	Président
M. Jean-Luc Lambert	Rapporteur
M. Patrice Quinton	Rapporteur
Mme. Dominique Borrione	Directeur de thèse

Remerciements

Je souhaite tout d'abord remercier les personnes qui ont permis la soutenance et la réalisation de cette thèse : ***Jean-Luc Lambert*** et ***Patrice Quinton***, qui ont bien voulu être mes rapporteurs malgré leur charge de travail. Je remercie également ***Ahmed Amine Jerraya***, qui a accepté de participer à mon jury. Merci à ***Bernard Courtois***, pour m'avoir accueilli au sein du laboratoire TIMA durant mes années de thèse.

Je tiens aussi à exprimer ma gratitude envers mon directeur de thèse, ***Dominique Borrione***, pour l'orientation de mon sujet de doctorat, son encadrement minutieux et attentif et ses réflexions précises et pertinentes. Elle a su orienter ma thématique de recherche avec mes goûts scientifiques.

Je remercie également les chercheurs de l'équipe VDS, ***Eric Gascard, Katell Morin-Allory*** et ***Pierre Ostier***, pour l'intérêt qu'ils ont porté à mon travail. Leurs réflexions sur les différentes problématiques que j'ai abordé dans cette thèse ont amélioré la qualité, la cohérence et la profondeur de mon travail.

Je remercie particulièrement ***Diana Toma-Moisuc*** pour son implication dans nos collaborations, ses commentaires judicieux et opportuns et son amitié sincère. Je remercie aussi les anciens et actuels membres de l'équipe VDS : ***Menouer Boubekeur, Emil Dumitrescu, Philippe Georgelin, Claude Le Faou et Julien Schmaltz*** pour les moments d'échanges très enrichissants, scientifiques et personnels. Je n'oublie pas également ***Bénédicte Fluxa*** pour son aide à la correction et aux suggestions de présentation de ce manuscrit.

Je tiens à témoigner ma reconnaissance à ma famille pour leur soutien, leur encouragement et l'exemple qu'elle a toujours su me donner. Enfin je remercie tout spécialement mon âme sœur, ***Nour***, pour sa compréhension, sa patience et pour tout l'amour et le bonheur qu'elle m'apporte chaque jour.

Le vrai danger, ce n'est pas quand les ordinateurs penseront comme les hommes, c'est quand les hommes penseront comme les ordinateurs.

Sydney J. Harris

Table des matières

1 INTRODUCTION 7

1.1 PRÉLIMINAIRE 8
1.2 ÉTAT DE L'ART 9
1.2.1 Les phases de conception des systèmes intégrés 9
1.2.2 Les niveaux de description matérielle 11
1.2.3 Les méthodes et les outils de vérification 13
1.3 LA SIMULATION SYMBOLIQUE 16
1.3.1 La simulation symbolique par ROBDD 17
1.3.2 La simulation symbolique par démonstration de théorème 19
1.3.3 Positionnement de notre approche de simulation symbolique 20
1.4 LA CONTRIBUTION DE LA THÈSE 21
1.5 L'ORGANISATION DE LA THÈSE 23

2 INTRODUCTION AU SYSTÈME MATHEMATICA 25

2.1 PRÉLIMINAIRE 26
2.2 LE LANGAGE DE PROGRAMMATION 26
2.2.1 La syntaxe et les objets de base 27
2.2.2 Les motifs et la reconnaissance de forme 29
2.2.3 Les définitions des fonctions et des règles de transformation 32
2.3 LE NOYAU D'ÉVALUATION 35
2.3.1 Valeurs des symboles 35
2.3.2 Principes de l'évaluation 36
2.4 DÉFINIR UNE STRATÉGIE D'ÉVALUATION 39
2.4.1 Les structures de contrôle 39
2.4.2 Les attributs des fonctions 40
2.5 LA STRATÉGIE D'ÉVALUATION STANDARD 41
2.6 LES FONCTIONS DE CALCULS ALGÉBRIQUES 42
2.7 CONCLUSION 43

3 MODÉLISATION DE VHDL PAR UN SYSTÈME D'EQUATIONS RÉCURRENTES (SER) 45

3.1 LE SOUS-ENSEMBLE VHDL 46
3.2 LE MODÈLE MATHÉMATIQUE 48
3.2.1 Motivation 48
3.2.2 Définitions et propriétés de la fonction Ife 49
3.2.3 Système d'équations récurrentes (SER) 50
3.3 REPRÉSENTATION D'UNE DESCRIPTION VHDL 51
3.3.1 Les affectations des objets 51
3.3.2 Les composants VHDL 52
3.3.3 Les types de données 53
3.3.4 Illustration sur un exemple 56
3.4 EXTRACTION DU MODÈLE 58
3.4.1 Principe de la méthode d'extraction 58
3.4.2 Normalisation des instructions séquentielles 59
3.4.3 Extraction des équations récurrentes à partir des instructions normalisées 62
3.4.4 Normalisation d'instructions concurrentes 68
3.5 IMPLÉMENTATION DANS MATHEMATICA 70
3.5.1 Le compilateur vers TheoSim 70
3.5.2 Les paquetages VHDL 73
3.5.3 Représentation d'une fonction 75
3.6 CONCLUSION 76

4 LA SIMULATION SYMBOLIQUE ... 77

4.1 Le mécanisme de simulation VHDL ... 78
4.1.1 Le simulateur VHDL ... 78
4.1.2 Le cycle delta ... 79
4.2 La simulation symbolique avec un SER ... 81
4.2.1 Exécution d'un cycle de simulation ... 82
4.2.2 L'algorithme de simulation ... 84
4.2.3 Le simulateur symbolique : TheoSim ... 85
4.3 La méthodologie de test symbolique ... 88
4.3.1 Partie contrôle et partie opérative du système ... 88
4.3.2 Les modes de vecteurs de test symboliques ... 89
4.3.3 Illustration sur un exemple ... 90
4.3.4 Utilité de la méthodologie de simulation symbolique ... 96
4.4 La réduction du modèle ... 98
4.4.1 La réduction par scénario ... 98
4.4.2 La réduction par abstraction ... 100
4.5 Conclusion ... 103

5 LA MÉTHODOLOGIE DE VÉRIFICATION ... 105

5.1 Présentation générale ... 106
5.2 Le paradigme de correspondance de formes ... 107
5.2.1 La vérification par simulation ... 107
5.2.2 Étude d'un filtre RIF ... 108
5.2.3 Vérification de la mémoire SPSMALL ... 114
5.3 Les paradigmes : démonstration de théorèmes et SAT ... 116
5.3.1 Les propriétés logiques et temporelles ... 116
5.3.2 Vérification par SAT ... 118
5.3.3 Vérification par un démonstrateur de théorème ... 120
5.4 Évaluation de la méthodologie ... 123

6 CONCLUSION ET PERSPECTIVES ... 125

6.1 Contributions ... 126
6.2 Perspectives ... 127

7 ANNEXES ... 129

7.1 Implémentation VHDL du filtre FIR ... 130
7.2 Fichiers M-code du filtre FIR ... 137
7.3 Le code VHDL du paquetage ... 147
7.4 Le LIF du paquetage ... 149
7.5 Modèle statique du paquetage ... 153
7.6 Modèle dynamique du paquetage ... 153

8 BIBLIOGRAPHIE ... 155

Chapitre 1

1 Introduction

1.1 Préliminaire

Au cours de cette dernière décennie, l'industrie des semi-conducteurs a connu une diminution continue dans la taille du transistor MOS et la capacité d'intégration n'a pas cessé d'augmenter : Intel annonce une taille de plus d'un milliard de transistors pour son prochain processeur *Dual Core Itanium*. Cette complexité est une lourde pression pour les concepteurs qui doivent livrer à temps leurs circuits en assurant à l'optimum leurs fonctionnalités. Une erreur dans la conception peut entraîner un retard dans la livraison et un coût supplémentaire en centaines de millions d'euros. Aussi, nous dépendons de plus en plus des circuits numériques dans notre vie et une erreur non révélée avant la fabrication peut poser un risque majeur spécialement dans les applications critiques (aérospatiales, centres nucléaires, appareils chirurgicaux…etc.).

Il n'est donc pas surprenant que jusqu'à 70 % du temps de conception soit consacré à la vérification. En réalité les méthodes de conception et de vérification ne progressent pas au même rythme que la complexité des circuits. Certes, des progrès sont réalisés dans les langages de description matérielle tels que SystemVerilog [IEEE01,Acc04b] et la nouvelle norme VHDL pour la synthèse [IEEE04], la synthèse de haut niveau ou encore les langages de spécification de propriétés tels que PSL [Acc04a]. Cependant, les méthodes de vérification dans l'industrie sont restées les mêmes.

La simulation numérique est jusqu'à aujourd'hui la première méthode de vérification dite aussi de validation. La simulation des vecteurs de test révèle la plupart des erreurs de conception, mais pour dévoiler les bugs difficiles les méthodes formelles sont utilisées. La description matérielle du circuit est exprimée en logique propositionnelle pour les circuits combinatoires et en machines d'états finis pour les circuits séquentiels. Des combinaisons de la représentation de fonctions booléennes en BDD avec des algorithmes tels que la vérification de modèle ou SAT, sont implémentées dans les outils commerciaux de vérification formelle. Ces techniques ne sont pas applicables sur les niveaux de descriptions où des types de données abstraits sont utilisés. Donc, ces méthodes ne suivent pas le progrès dans le niveau d'abstraction des langages de description matérielle.

D'un autre côté, la démonstration de théorème est applicable sur tous les niveaux de description, mais cette technique demande des experts en preuves. Ceci limite l'intégration de la

démonstration de théorème dans les parties sensibles du circuit conçu par de grandes entreprises.

1.2 État de l’art

1.2.1 Les phases de conception des systèmes intégrés

Pendant son développement, une conception numérique passe par des transformations multiples, de la spécification originale jusqu’au produit final. Chacune de ces transformations correspond à une description différente du système, qui contient progressivement plus de détails et qui a sa propre sémantique. Le schéma 1.1 présente le flot de conception de la spécification jusqu’au produit final. Il montre une vision simplifiée descendante. En réalité, une conception industrielle est beaucoup plus complexe, comportant beaucoup d'itérations dans les étapes de ce schéma pour que la conception converge vers une forme qui couvre la fonctionnalité, la synchronisation, la consommation et le coût spécifiés initialement.

Les spécifications d’une conception sont généralement données par un document décrivant un ensemble de fonctionnalités, souvent en anglais, que la solution finale devra fournir, et un ensemble de contraintes qu'il doit satisfaire. Dans ce contexte, la description fonctionnelle est la première description précise du circuit. En raison de la complexité de la tâche, une approche hiérarchique est utilisée, de sorte que chaque groupe de concepteurs se focalise sur un composant du circuit. La description fonctionnelle est écrite en un langage de programmation de haut niveau tel que SystemC, Matlab, Mathematica, LISP…etc. Elle est utilisée pour l’évaluation des algorithmes choisis et également comme un modèle de référence pour vérifier le comportement des descriptions plus détaillées dans les étapes suivantes.

A partir de la description fonctionnelle, l'équipe de conception matérielle procède à la phase de description matérielle. Pendant cette phase, l’architecture finale est déterminée : chaque module de la description fonctionnelle est décrit par un ou plusieurs composants à l’aide d’un langage de description matérielle HDL tels que VHDL ou VERILOG. Cette phase voit également l’apparition des éléments de synchronisation et l’estimation des paramètres tels que la vitesse et la consommation de la conception.

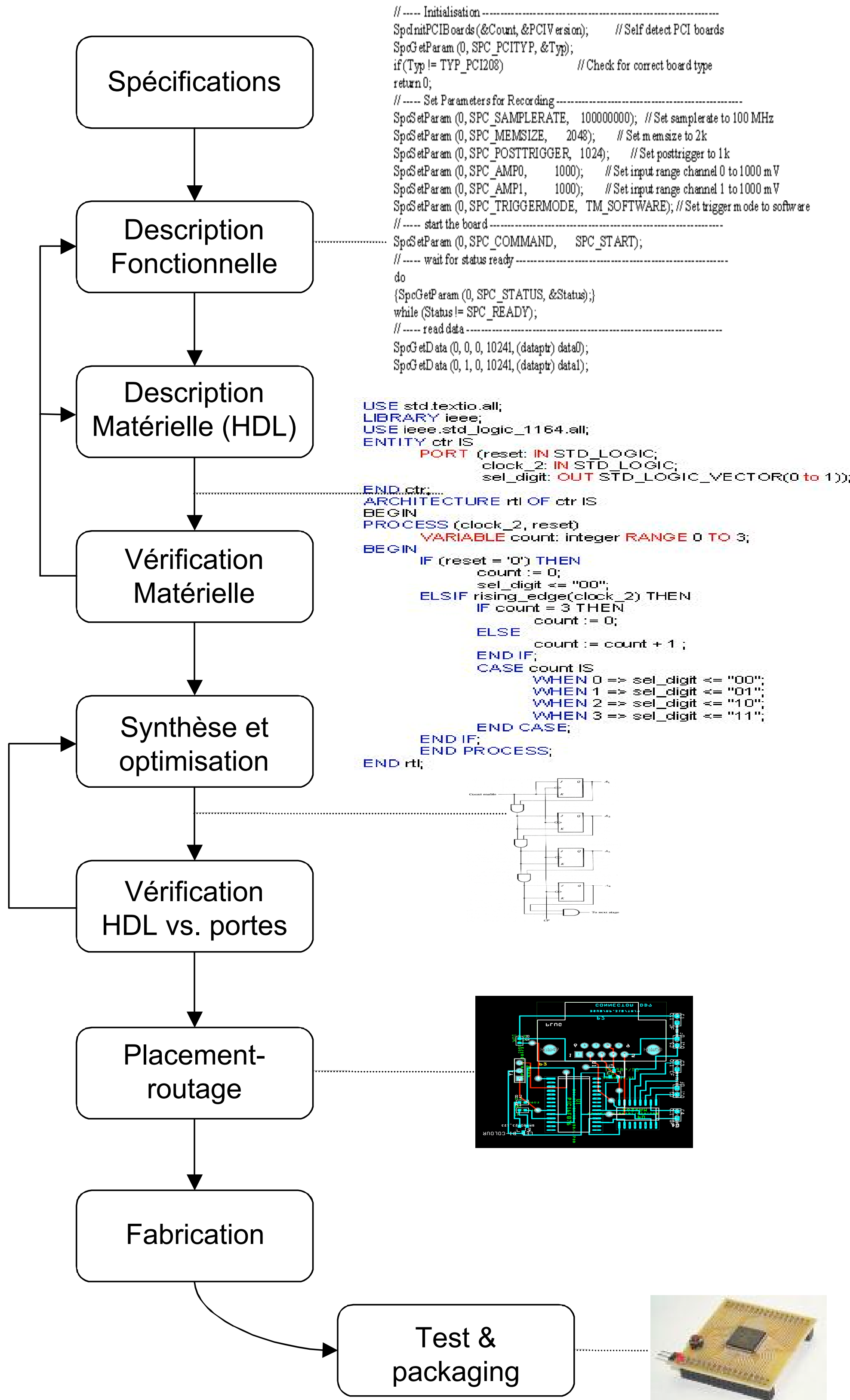


Figure 1.1 : Schéma simplifié du flot de conception numérique.

La vérification de la description matérielle consiste à assurer qu'elle fonctionne correctement par rapport la description fonctionnelle et les spécifications. Le but est d'éliminer toute erreur de conception possible avant la fabrication du circuit. Chaque fois qu'une erreur de conception est trouvée, le modèle doit être modifié et vérifié à nouveau.

Il est également possible que la phase de vérification de la description matérielle indique une certaine incohérence dans les spécifications, et celle-ci sont donc mises à jour. La simplification du schéma 1.1 donne l'illusion que la vérification est une phase isolée. Cependant, en pratique elle est menée en parallèle avec les autres activités et parfois jusqu'au placement-routage du composant.

La phase suivante est la synthèse de la description matérielle pour obtenir un modèle détaillé au niveau du réseau de portes logiques : ET, OU, NON…etc., et des éléments mémoires. Le modèle généré est optimisé selon certaines contraintes : consommation d'énergie, superficie de silicium ou rapidité du produit final. Parfois, les contraintes de coût exigent plus d'optimisation que ne le permettent les outils automatiques. Dans ce cas, les concepteurs interviennent et optimisent de façon manuelle la description, soit en changeant directement la description au niveau des portes, soit en modifiant la description matérielle. Cela peut provoquer l'apparition d'erreurs fonctionnelles, c'est pourquoi la vérification du résultat de la phase de synthèse et d'optimisation est importante.

La vérification d'équivalence fonctionnelle entre la description au niveau portes et la description matérielle est un processus complètement automatique qui demande un minimum d'interaction humaine. Après la vérification d'équivalence, il est possible de procéder au placement-routage. Le résultat de cette phase est une description géométrique du circuit qui définit les couches physiques nécessaires au procédé de fabrication. Finalement, le circuit est fabriqué et les puces sont testées.

1.2.2 Les niveaux de description matérielle

Les langages de description matérielle tels que VHDL permettent de décrire le circuit avec plusieurs niveaux d'abstraction [Jer02]:

Le niveau comportemental : décrit le système par un algorithme au niveau d'un cycle de simulation sans se soucier d'une éventuelle réalisation au niveau des composants tels que les

mémoires, les multiplexeurs, les circuits de commande. Par exemple, une multiplication entre deux signaux est écrite en une seule ligne en langage tel que VHDL. De même qu'à ce niveau, la description peut contenir des signaux de types de données abstraites tels que les entiers. Le but est d'augmenter la productivité pour la conception des systèmes complexes en utilisant la synthèse de haut niveau. De même qu'une description comportementale est plus compacte (au niveau du nombre d'instructions HDL), la vérification (notamment par simulation numérique) à ce niveau demande moins de temps d'exécution.

Le niveau du transfert de registre (RTL) : décrit le système en termes de registres, circuits combinatoires, bus de bas niveau et circuits de commande. Pour comparer avec le niveau comportemental, une multiplication entre deux signaux est écrite en décrivant les détails de l'implémentation structurelle à l'aide des composants existants (registres, multiplexeurs...etc.). Il est habituellement implémenté en termes de machine d'états finis tout en intégrant les détails au niveau du temps d'exécution. La conception au niveau RTL est la plus fréquentée dans l'industrie. La synthèse à partir de ce niveau est dite synthèse RTL.

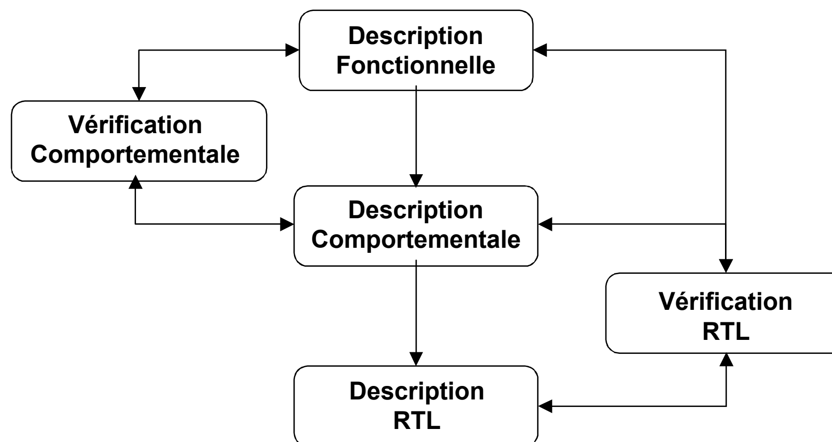


Figure 1.2 : succession d'étapes de vérification

Les descriptions de plus bas niveau : comportent les niveaux logiques, portes et physiques. Le niveau logique, par exemple, décrit le circuit en termes de fonctions booléennes et de simples éléments mémoires tels que des bascules. L'écriture d'une description à ces niveaux est presque abandonnée dans l'industrie. Seulement, dans le cas d'une conception qui intègre un composant issu d'une ancienne technologie, le concepteur peut alors modifier un élément décrit à l'aide de ces niveaux.

Aujourd'hui les outils de synthèse permettent de synthétiser une description RTL à partir du niveau comportemental. Pour optimiser le résultat, des interventions humaines au niveau RTL sont nécessaires. Ensuite le niveau RTL est synthétisé à son tour pour obtenir la description logique du circuit. En effet, comme le montre la figure 1.2, une succession d'étapes de vérification est nécessaire pendant la conception et la vérification d'un circuit numérique.

La description matérielle comportementale est vérifiée vis-à-vis de la description fonctionnelle, et la description RTL est vérifiée par rapport à la description comportementale et dans certains cas directement par rapport à la description fonctionnelle.

1.2.3 Les méthodes et les outils de vérification

Nous distinguons plusieurs formes de vérification [Jer02]:

La vérification d'équivalence : permet de montrer au même niveau de description que les sorties obtenues de deux descriptions du même circuit sont égales pour les mêmes entrées. Elle est utilisée pour valider une synthèse ou une étape d'optimisation.

La vérification de conformité : permet de montrer qu'une description est une réalisation d'un niveau de description plus abstrait d'un circuit. Elle est utilisée pour prouver que la description matérielle respecte la spécification.

La vérification des propriétés logicotemporelles : est utilisée pour vérifier que les circuits de contrôle (arbitres de bus, microcontrôleurs, contrôleurs de caches…) satisfont leurs spécifications initiales. Les propriétés sont décrites à l'aide de chronogrammes, d'une logique temporelle arborescente (CTL) ou d'une logique temporelle linéaire (LTL) [CES86] et plus récemment par le langage PSL [Acc04a]. L'absence d'une situation de blocage ou le fait qu'une sortie ne sera pas retardée indéfiniment sont des exemples de telles propriétés.

La vérification des propriétés mathématiques : est utilisée lorsque le circuit vérifié implémente un opérateur mathématique tel que les circuits de traitement du signal. Les propriétés sont des formules mathématiques que les sorties doivent satisfaire, ou des propriétés telles que la commutativité de l'opérateur .

Dans l'industrie, les outils de vérification sont variés. Les simulateurs numériques sont utilisés pour tous les niveaux d'abstraction et pour vérifier toutes les formes de propriétés. Le principe est de compiler la description matérielle dans un langage tel que C pour obtenir un modèle exécutable [Han88]. Le modèle est exécuté avec une batterie significative d'entrées et la sortie est examinée pour assurer le bon comportement. Aujourd'hui, les outils de CAO génèrent automatiquement les vecteurs de tests [HKM01, Ber03]. Ces vecteurs sont produits aléatoirement en se basant sur les spécifications. Les outils fournissent des langages pour décrire le scénario de tests [HMN01] et ils vérifient les sorties des composants par rapport à ces tests. La qualité de tous ces efforts de validation numérique est analytiquement évaluée en termes de couverture (coverage en anglais). Elle mesure la portion de la conception qui a été vérifiée [KaN96]. Le but est de maximaliser la couverture sachant qu'elle est toujours partielle.

La complexité croissante des circuits intégrés a rendu la validation numérique exhaustive impossible, et les outils de vérification formelle sont employés pour compléter la simulation et détecter les erreurs de conception qui échappent à la simulation. Ces outils utilisent des techniques mathématiques pour prouver les propriétés du circuit sans se limiter à un ensemble de vecteurs de test. Il n'existe pas de technique ou de logiciel capable de vérifier toutes les formes de propriétés et tous les niveaux de description. Un outil est choisi selon les propriétés et le niveau vérifiés. Nous distinguons les classes suivantes :

Les démonstrateurs automatiques de théorème : ce sont des bibliothèques d'axiomes et des règles de déductions logiques. Ils emploient des algorithmes de raisonnement logique, tels que l'induction, la généralisation et la réécriture pour prouver qu'un théorème est dérivé des axiomes. La vérification du circuit avec ces démonstrateurs consiste à formaliser les spécifications et la description du circuit dans la logique du démonstrateur, et à raisonner ensuite sur ces modèles pour vérifier les propriétés du circuit. La vérification est une vérification de conformité. De même nous pouvons vérifier toutes les formes de propriétés logiques et mathématiques. Malgré l'automatisation de ces démonstrateurs, leur utilisation sur le circuit n'a rien d'automatique. En réalité, la formalisation de la description dans la logique de démonstrateur est une opération longue et n'est pas automatisée. En outre, la formalisation des propriétés du circuit nécessite souvent d'ajouter une quantité importante de théorèmes et lemmes pour étendre la logique du démonstrateur. Ajoutons à toutes ces difficultés le besoin d'une longue expertise dans la logique de démonstrateur. En conséquence, seuls les gros industriels sont capables d'employer des spécialistes dans ce domaine, tels qu'Intel et IBM, et

seulement pour vérifier les parties les plus critiques du circuit. Il existe plusieurs démonstrateurs de théorème, citons comme exemples :

- PVS : un démonstrateur de théorème inductif, décrit dans [OSR92]. Il est doté d'une logique d'ordre supérieur. Rueß a prouvé la correction de la description matérielle de l'unité de division flottante SRT vis-à-vis des spécifications en utilisant PVS [RSS96].
- ACL2 : un démonstrateur de théorème inductif développé initialement par CLI et ensuite par J Moore et M. Kaufmann [KMM00a] à l'université de Texas, Austin. Il est l'héritier du célèbre démonstrateur de théorème de Boyer-Moore NQTHM [BoM97]. Dans le chapitre 13 de [KMM00b], *Russinoff et al* ont vérifié une unité de muliplication en virgule flottante à l'aide d'ACL2. Dans [SaH02], ACL2 est utilisé pour la vérification 'un microprocesseurs pipeline.

Les vérificateurs de modèles : permettent de vérifier les propriétés logicotemporelles. Ces outils reposent sur la technique de la vérification de modèle (Model-checking en anglais) qui a été proposée initialement par Clarke et Emerson [CES86], où le circuit est modélisé par une machine d'états finis [CBM90]. McMillan a utilisé les diagrammes de décision binaire ROBDD [Bry86] pour représenter symboliquement l'espace des états dans le modèle de la machine d'états finis [McM93]. Il a présenté aussi une procédure efficace de calcul de point fixe pour déterminer si le modèle de la machine d'états finis valide des propriétés logicotemporelles. Cette procédure a rendu possible la vérification de circuits de tailles importantes. SMV est le premier vérificateur de modèle qui a prouvé l'efficacité de cette technique sur un espace d'états de l'ordre de 10^{20} [BCM92]. Des outils qui utilisent plutôt des moteurs de test de satisfaction booléenne (SAT) à la place de BDD sont apparus [McM03]. Ils effectuent une vérification de modèle bornée BMC (Bounded Model Checking) [BCC99] en donnant un état initial et un nombre de pas fixes. Le but est de chercher un contre-exemple rapidement. Cette technique ne garantit pas que la propriété est valide dans tous les cas possibles, mais elle reste utile car elle est applicable sur des circuits de taille plus importante que ceux traités par la vérification de modèle classique. Les vérificateurs de modèles sont beaucoup plus automatiques que les démonstrateurs de théorèmes. Donc ces outils intègrent mieux le milieu industriel. Il existe plusieurs vérificateurs de modèles industriels basés sur les mêmes principes :

- FormalCheck : commercialisé par Cadence [WWW02].
- Magellan : commercialisé par Synopsys [WWW03].
- RuleBase : développé par IBM Haifa Research Lab [WWW01].

Les vérificateurs d'équivalence : sont utilisés pour vérifier l'équivalence qui peut être séquentielle ou booléenne. Ces logiciels sont très bien automatisés et l'intervention humaine est minime par rapport aux autres outils de vérification formelle. Par conséquent, ils sont parfaitement intégrés dans le flot de conception actuelle. Citons par exemple les outils industriels suivants :

- Formality : vendu par Synopsys [WWW03].
- Encounter : proposé par Cadence [WWW02].
- ESP-CV : conçu initialement par Innologic et vendu aujourd'hui par Synopsys [WWW03]. Il a la particularité d'utiliser la simulation symbolique par ROBDD pour vérifier l'équivalence.

1.3 La simulation symbolique

La simulation symbolique est une technique de vérification considérée par certaines communautés de vérification comme formelle, et par d'autres communautés comme semi-formelle. L'idée principale de la simulation symbolique consiste à introduire des symboles à l'entrée du circuit à la place des valeurs numériques. La simulation symbolique fut proposée pour la première fois par IBM [Dar79] fin des années soixante-dix. L'idée est d'utiliser l'exécution symbolique des programmes pour les circuits. Les difficultés rencontrées à l'époque sont les suivantes :

- Les expressions symboliques croissent exponentiellement avec le nombre de cycles de simulation.
- En présence de branchements conditionnels où la condition est un terme symbolique, tous les chemins possibles doivent être explorés. Ceci produit un arbre de simulation qui croît de manière exponentielle.
- Sans simplification, les résultats de la simulation symbolique sont des termes illisibles.

La raison de ces difficultés est l'absence d'un modèle mathématique efficace pour la représentation symbolique du circuit.

La simulation symbolique a été ressuscitée après l'introduction du ROBDD par Bryant [Bry89]. En effet, c'est Bryant qui a présenté un premier simulateur symbolique des circuits décrits au niveau transistor [BBC87].

Une deuxième approche de simulation symbolique a été proposée par Burch et Dill [BuD94] en 1994 pour la vérification de microcontrôleurs. L'approche est en deux étapes. La première étape génère par compilation une formule logique pour la description comportementale du circuit. La logique utilisée est une logique du 1er ordre sans quantificateur. La deuxième étape transmet cette formule à un autre logiciel dit « Validity Checker » pour vérifier si la formule est valide ou non. Cette technique en deux étapes est adaptée pour les descriptions qui contiennent des parties données importantes. Dans le même esprit, Moore a proposé une méthode de simulation symbolique dans le démonstrateur de théorème ACL2 [Moo98].

1.3.1 La simulation symbolique par ROBDD

L'algorithme propose de modéliser les nœuds du circuit en utilisant les ROBDD [Bry92]. La simulation symbolique selon cette approche est une exploration itérative de l'espace d'états du circuit simulé. Ces états sont les valeurs des mémoires du circuit. L'exploration est effectuée sur la description au niveau des portes du circuit. Pour simuler symboliquement les descriptions de plus haut niveau, un compilateur ou une synthèse logique sont utilisés. A chaque pas de simulation, les entrées et les variables d'état sont affectées par des expressions booléennes qui peuvent être des valeurs numériques ou symboliques. La simulation se déroule en dérivant une expression symbolique (booléenne) pour chaque signal interne du circuit. Les expressions symboliques des variables d'état et des fonctions booléennes associées à chaque porte logique sont calculées en fonction des entrées.

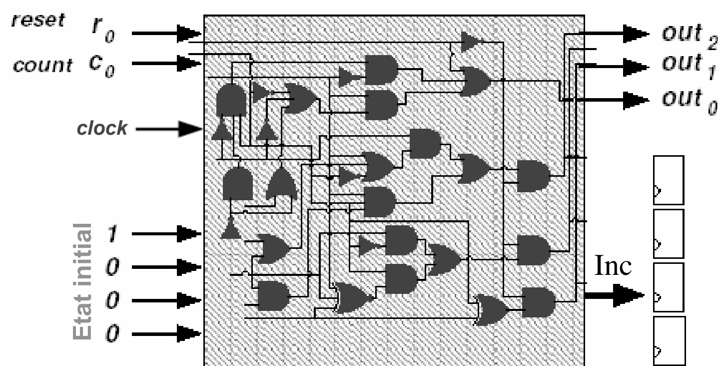


Figure 1.3 : vue abstraite de réseaux de portes d'un compteur 3 bits.

Considérons la simulation symbolique d'un compteur sur 3 bits avec deux possibilités :

incrémentation ou décrémentation. La figure 1.3 montre une vue abstraite du compteur dans laquelle les portes logiques ne représentent pas nécessairement la synthèse logique du compteur. L'état du compteur est mémorisé avec quatre registres : trois pour sa valeur courante et un pour savoir s'il incrémente ou décrémente (le signal *Inc*). À chaque coup d'horloge (*clock*), le circuit met à jour les registres internes si le signal *count* est activé. Si le compteur arrive à la valeur maximale (7 pour 3 bits), le compteur commence à se décrémenter jusqu'à obtenir zéro dans les registres. Le signal *reset* remet le compteur à zéro. Les sorties Out_0, Out_1, Out_2 donnent la valeur actuelle du compteur.

La simulation symbolique du compteur commence par spécifier un état initial, par exemple « 0001 » qui correspond à zéro pour les registres contenant les valeurs du compteur (x_0, x_1, x_2), et 1 pour la direction du comptage (*inc*) : une incrémentation. Le *reset* est affecté par r_0 et *count* par c_0. Les sorties du compteur après un cycle (ou une étape) de simulation symbolique sont :

$Out_0 = x_0 = not(r_0)\ and\ c_0$

$Out_1 = x_1 = Out_2 = x_2 = 0$

Dans le cycle suivant, *reset* est affecté par r_1 et *count* par c_1, et les registres internes contiennent les expressions symboliques calculées dans le premier cycle. Les sorties du compteur sont :

$Out_0 = x_0 = ((not(r_0)\ and\ c_0)\ xor\ c_1\ )\ and\ not\ (r_1)$

$Out_1 = x_1 = (((not(r_0))\ and\ (not(r_1))\ and\ c_0\ and\ c_1$

$Out_2 = x_2 = 0$

Soit $Spec_i$ l'expression booléenne attendue du signal Out_i comme définie par la spécification du circuit. Une erreur dans le circuit est détectée lors de la comparaison entre $Spec_i$ et Out_i, pour tous les objets du circuit.

Nous remarquons que la complexité des expressions symboliques a augmenté dans le deuxième cycle. Rappelons que les expressions symboliques sont représentées avec les ROBDD. Malgré les heuristiques, la simulation symbolique avec ROBDD explose par rapport à la mémoire nécessaire pour manipuler les expressions. Plusieurs techniques sont proposées pour contourner ce problème, citons par exemple :

- Dans [WiD00], une taille maximale est fixée pour chaque BDD. Si le BDD atteint cette

limite, une ou plusieurs variables booléennes sont remplacées par des valeurs numériques et le circuit est simulé symboliquement plusieurs fois pour couvrir toutes les valeurs numériques de ces variables.

- Dans [BeH99], Bergmann propose une approche basée sur l'abstraction. L'idée est d'éviter les parties non utilisées du circuit et de les remplacer par une valeur abstraite. Ensuite, abstraire une autre partie du circuit jusqu'à obtenir une couverture totale.

Des techniques basées sur STE (Symbolic Trenery Trajectory Evaluation) [BBC91] telles que l'exploitation de symétrie dans la simulation [PaB97], ou des techniques sur la combinaison d'un moteur de simulation numérique avec le simulateur symbolique [HSH00], s'inscrivent dans l'école de simulation symbolique par ROBDD. Le but de ces travaux est de garder la simulation symbolique compétitive dans l'industrie malgré la complexité croissante des circuits.

1.3.2 La simulation symbolique par démonstration de théorème

Dans [Gre98], Greve a proposé l'utilisation du démonstrateur de théorème PVS [OSR92] comme un simulateur symbolique pour le microprocesseur JEM1 qui exécute le code JAVA. Tout d'abord, il a formalisé un modèle du microprocesseur dans la logique d'ordre supérieur de PVS. Ensuite, il a exécuté numériquement le modèle avec les vecteurs de tests utilisés par les concepteurs pour la validation numérique. Son but est d'acquérir une première confiance dans le modèle avant la simulation symbolique. Ensuite, il a exécuté symboliquement le modèle en utilisant les algorithmes définis dans PVS (réécriture, généralisation, élimination…etc.). L'utilisateur définit les théorèmes que le système doit satisfaire, ils expriment une vérification de conformité ou des propriétés mathématiques. Finalement, il a prouvé les théorèmes qui montrent la validité du circuit sur ces résultats symboliques. L'approche a révélé plusieurs erreurs de conception non trouvées avec la simulation numérique. Moore a généralisé l'approche de Greve dans [Moo98] en utilisant la logique du 1er ordre d'ACL2.

Dans cette approche de simulation symbolique, le modèle (ACL2 ou PVS) est écrit manuellement. La performance de l'exécution symbolique selon Moore dépend du moteur de réécriture d'ACL2. Donc l'efficacité du modèle dépend de l'utilisateur et de ses connaissances en ACL2, sachant que les experts dans ce genre de démonstrateur sont rares.

Dans notre équipe de recherche, Georgelin a proposé une approche pour la simulation symbolique avec ACL2 pour des circuits décrits en VHDL [Ge001]. Le sous-ensemble considéré est une restriction du standard [IEEE99] où tous les processus sont synchronisés par une seule horloge. Le circuit est modélisé comme une machine d'états finis :

- Un état du circuit est défini dans ACL2. C'est une liste qui contient les valeurs des variables et signaux du circuit VHDL pour un cycle d'horloge.
- Une fonction de transition exprime le comportement du circuit pour un cycle d'horloge. L'entrée de la fonction est l'état actuel du circuit (dans un cycle d'horloge t), et la sortie est l'état futur (dans un cycle d'horloge t+1). Cette fonction est une traduction directe du VHDL en LISP.
- La modélisation au niveau du cycle d'horloge a réduit la capacité de modéliser des descriptions hiérarchiques : le modèle est applicable seulement pour une classe particulière de circuits, où la partie combinatoire est ordonnable statiquement par compilation.

Un noyau de simulation est défini sous la forme d'une fonction récursive. La fonction peut être exécutée dans ACL2 numériquement ou symboliquement. La performance de l'exécution symbolique avec cette approche est très limitée. En réalité, la taille de la fonction de transition manipulée symboliquement devient énorme. Par conséquent, le temps d'exécution explose en appliquant les algorithmes tels que la réécriture et la généralisation par ACL2, même pour un nombre relativement réduit de cycles de simulation.

1.3.3 Positionnement de notre approche de simulation symbolique

La simulation symbolique par ROBDD est idéale pour les circuits décrits au niveau logique en décrivant le circuit bit à bit. Par contre, elle est inapplicable sur les descriptions abstraites où le circuit est décrit en termes d'opérations sur des mots de grandes tailles et des entiers non bornés. Aussi, cette approche ne considère pas les circuits définis par des paramètres génériques. Notre approche est conçue pour ce genre de circuit.

La simulation symbolique par démonstration de théorèmes est capable de traiter des circuits

décrits au niveau abstrait. Par contre, la modélisation est manuelle. La thèse de Georgelin a donné une ouverture pour résoudre ce problème. Notre approche part de ses travaux, en analyse les limites et adapte les principes suivants :

- Nous proposons un modèle mathématique du circuit plus abstrait par rapport aux types de données et plus détaillé au niveau de l'exécution.
- Notre modélisation est au niveau du cycle de simulation et non pas au niveau du cycle d'horloge. Donc aucune restriction sur les descriptions hiérarchiques.
- Nous utilisons la technique « diviser pour régner » pour la fonction de transition du système. Nous générons une fonction pour chaque objet du circuit (variable ou signal). La fonction de transition du circuit est la combinaison de toutes ces fonctions.
- Nous utilisons un outil de calcul symbolique : Mathematica à la place du démonstrateur de théorème. L'idée est de séparer la partie calcul des expressions symboliques de la partie raisonnement. Donc seule la réécriture est utilisée pour la simulation symbolique. Les techniques telles que la généralisation sont utilisées après la simulation sur les résultats symboliques.

Avec ces principes, nous visons une simulation symbolique de haut niveau, où le modèle est généré automatiquement et avec une performance acceptable.

1.4 La contribution de la thèse

Nous étudions dans cette thèse le problème de la simulation symbolique des circuits décrits à l'aide du langage VHDL synthétisable standardisé en 1999, puis en 2004. Après la sortie fin 2004 du standard final [IEEE04], nous avons défini le sous-ensemble de VHDL que nous traitons dans ce manuscrit dans [Als04]. Les circuits décrits avec ce sous-ensemble sont des circuits synchrones qui contiennent plusieurs horloges, et des signaux ou des variables de types abstraits tels que les entiers.

Notre objectif est de répondre aux exigences suivantes :

- Satisfaire le standard VHDL décrit dans [IEEE04], y compris l'existence de plusieurs horloges dérivées d'une horloge mère.

- La simulation du circuit se déroule indépendamment de l'existence ou non d'erreurs de conception.
- Aucune assistance n'est nécessaire pour le simulateur symbolique. Le concepteur fournit le circuit en VHDL, précise l'horloge mère du circuit et le nombre de cycles de simulation. Le simulateur symbolique s'occupe du reste. Donc, l'approche est automatique et aucune expertise en démonstration de théorèmes n'est demandée.
- Les circuits ne sont pas synthétisés pour la simulation symbolique. Donc, un signal de type entier reste entier et n'est pas transformé en un vecteur de bits.
- La validation de propriétés du circuit est indépendante de sa simulation symbolique.
- La méthode est applicable à des circuits avec un chemin de donnée de taille importante.

Nous avons défini une sémantique de simulation symbolique pour le sous-VHDL considéré et un prototype en utilisant l'outil Mathematica. Le flot de traitement et d'exécution est présenté dans la figure 1.4. A partir d'un fichier VHDL, nous générons un modèle mathématique de son comportement sous la forme d'un Système d'Equations de Récurrence (SER). Nous appliquons l'algorithme de simulation VHDL sur ce modèle pendant *n* cycles de simulation (n est donné par le concepteur). Pendant l'exécution du modèle, nous appliquons des vecteurs de test symbolique et des règles de simplification. Nous obtenons ensuite les résultats : une expression symbolique pour chaque objet du circuit. Nous avons utilisé cette approche pour vérifier des circuits académiques et industriels.

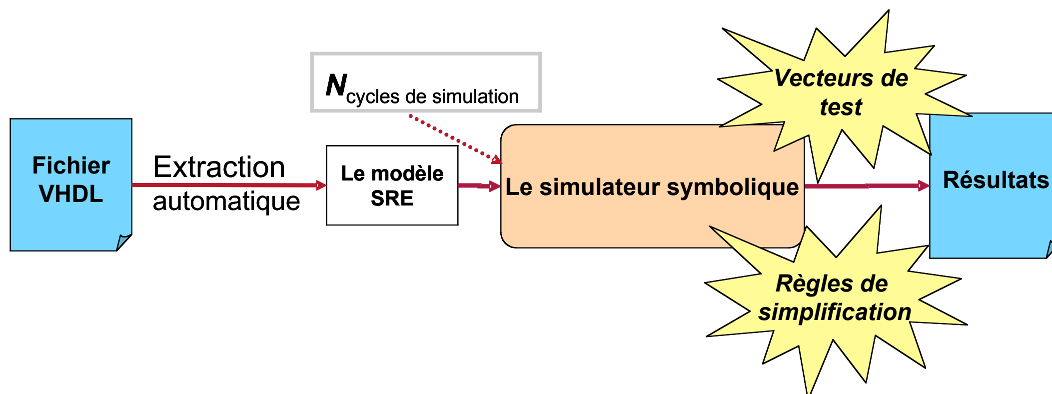


Figure 1.4 : le flot de simulation symbolique.

1.5 L'organisation de la thèse

Le chapitre 2 introduit l'outil de calcul et de programmation symbolique Mathematica. Nous avons aussi présenté les algorithmes qui sont utiles pour définir les travaux dans le reste du manuscrit.

Le chapitre 3 définit le modèle mathématique : le système d'équations récurrentes ou SER. Nous présentons aussi dans ce chapitre comment modéliser un circuit VHDL avec ce modèle et comment l'extraire automatiquement. Ce chapitre est le fruit d'une collaboration proche avec un autre membre de l'équipe : Diana Toma.

Le chapitre 4 décrit l'algorithme de simulation symbolique basé sur le système SER, les règles de simplification symbolique et la méthodologie de simulation à l'aide des vecteurs de test symbolique. Le chapitre s'achève sur une application.

Le chapitre 5 propose une méthodologie de vérification autour de notre simulateur symbolique. Nous présentons aussi dans ce chapitre deux études de cas, une académique sur un filtre numérique, et l'autre industrielle sur une spécification de haut niveau d'une mémoire conçue chez STMicroelectronics.

Le chapitre 6 inclut la conclusion et les perspectives de ce manuscrit.

Chapitre 2

2 Introduction au système Mathematica

Ce chapitre introduit la structure de Mathematica et les concepts particuliers comme la représentation interne des objets, les expressions, les séquences d'évaluation, la programmation fonctionnelle et les motifs. Le but est de permettre au lecteur de pouvoir comprendre les choix qui sont faits dans l'implémentation et la modélisation de notre simulateur symbolique de VHDL.

2.1 Préliminaire

La réalisation de la première version de Mathematica, en 1988, est considérée comme le début d'une nouvelle ère dans le monde des logiciels de calculs mathématiques (voir [Wol88]). Certes, depuis le début des années soixante, plusieurs paquetages et outils ont été créés pour des tâches variées comme les calculs numériques, algébriques, et pour les manipulations graphiques [CoW81]. Mais le nouveau concept de Mathematica est de traiter tous les aspects du calcul mathématique dans le même système et de façon cohérente [Wol85]. La clef de cette réussite réside dans son langage de programmation basé sur les expressions symboliques inspirées des langages tels que le LISP (voir [McC60] et [McC63]). Cela a permis la manipulation d'objets variés en utilisant un ensemble réduit de primitives. Aujourd'hui, Mathematica est le logiciel le plus utilisé pour les calculs symboliques (80 % des parts du marché mondial) et parmi les plus utilisés pour les calculs mathématiques en général [EGK95].

Le système Mathematica est composé de deux parties :

- Le langage de programmation : il permet la manipulation des symboles et l'écriture des algorithmes symboliques.
- Le noyau d'évaluation : il est basé sur la réécriture des définitions et l'application des règles de remplacement.

2.2 Le langage de programmation

Le langage de Mathematica est un langage multiparadigme, fonctionnel, procédural et orienté objet (voir [Mae96] et [Mae00]). Néanmoins, son langage multiparadigme est une encapsulation d'un langage basé sur une syntaxe fonctionnelle et un mécanisme d'évaluation transformationnel (à base de règles).

2.2.1 La syntaxe et les objets de base

La syntaxe est très proche de la notation algébrique traditionnelle. La distinction est faite entre les lettres majuscules et minuscules. Le tableau 1 résume quelques séparateurs et leurs usages.

Définition 3.1 : Symbole
Un symbole est une suite de lettres, de formes littérales et de chiffres. Le nom du symbole ne commence pas par un chiffre. Les noms des symboles prédéfinis respectent certaines règles générales :

- Le nom comporte des mots anglais complets ou des abréviations mathématiques standard. La graphie américaine est utilisée.
- La première lettre de chaque mot est une lettre majuscule.
- Les symboles dont les noms se terminent par Q sont généralement des prédicats qui renvoient True ou False.

Tableau 1 : les séparateurs et leurs significations	
(…)	Les parenthèses servent à grouper des termes.
[…]	Les crochets suivent une fonction ou un opérateur et encadrent ses arguments, séparés par des virgules.
{…}	Les accolades servent à définir des listes, les éléments d'une liste sont séparés par des virgules.
[[...]]	Les doubles accolades servent à encadrer les indices d'une liste.
f[*expr1*, *expr2*]	Les virgules sont les séparateurs entre les paramètres d'une fonction.

Les différentes formes d'objets tels que nombres, symboles, listes, formules, etc... sont représentées intérieurement sous une forme unifiée que l'on appelle expression symbolique.

Définition 3.2 : expression symbolique

Une expression symbolique est de la forme h[e_1, e_2 , ...] où h est un symbole que nous appelons en-tête de l'expression. Les e_i sont les éléments de l'expression. Les éléments et l'en-tête peuvent être eux-mêmes des expressions symboliques.

Voici quelques exemples d'expressions symboliques :

- Fonc[a,b,1]
- Real[2.5]
- foo
- foo[a,b,d[2]]

Une expression symbolique est composée ou atomique. Le tableau 2 explicite les types d'expressions atomiques dans Mathematica. Toute autre expression est considérée comme composée. Il existe d'autres types comme les rationnels et les complexes. Ils sont considérés comme expressions symboliques composées car ils sont construits à partir des types atomiques.

Définition 3.3 : une fonction

Une fonction est une expression symbolique *f[e_1, e_2 , ...]* qui satisfait les caractéristiques suivantes :

- l'en-tête « f » est une expression symbolique atomique du type *Symbol*, c'est le nom de cette fonction.
- Au moins un élément e_i *existe*. Les e_i sont appelés les arguments de la fonction *f*.

Dans une fonction comme foo[x], le nom de la fonction est foo : une expression symbolique du type *Symbol*, et *x* est son argument. La nature symbolique du langage Mathematica permet de traiter les noms de fonctions comme des expressions et rend possible l'ensemble des opérations fonctionnelles qui existent dans d'autres langages tels que LISP. Le tableau 3 donne une liste des opérateurs fonctionnels les plus utilisés dans Mathematica.

Généralement, la syntaxe est préfixée. Mais, pour faciliter la lisibilité de quelques expressions comme les affectations, les opérations arithmétiques et les tests logiques, une forme d'écriture non préfixée est associée. Elle correspond à une syntaxe plus compacte empruntée au langage C ; par exemple la fonction qui teste l'égalité *Equal[x,y]* peut être écrite aussi sous la forme *x == y*. Bien sûr, les deux formes sont équivalentes, et pour obtenir la forme standard d'une expression *expr*, nous pouvons utiliser la fonction *FullForm[expr] :* FullForm[3] donne Integer[3].

Tableau 2 : les expressions atomiques en Mathematica	
Symbol	Un symbole.
Integer	**LES ENTIERS.**
Real	Les réels.
Character	Un caractère.
String	Une chaîne de caractères.

Tableau 3 : les opérateurs fonctionnels en Mathematica	
`Map[`*f*`,` *expr*`]`	Applique *f* à chaque élément du premier niveau de *expr*.
`MapAll[`*f*`,` *expr*`]`	Applique *f* à chaque sous-expression de *expr*.
`Apply[`*f*`,` *expr*`]`	**REMPLACE L'EN-TÊTE DE *EXPR* PAR *F*.**
`Head[`*expr*`]`	Renvoie le type ou l'en-tête de *expr*.
`Part[`*expr*`,` *i*`]`	Envoie la *i*-ème élément de *expr*
`FullForm[`*expr*`]`	Imprime la forme complète de *expr*, sans syntaxe spéciale.
`First[`*expr*`]`	Renvoie le premier élément de *expr*.
`Last[`*expr*`]`	Donne le dernier élément de *expr*.
`Rest[`*expr*`]`	Donne *expr* sans son premier élément.
`Drop[`*list*`,` *n*`]`	Renvoie *list* avec ses *n* premiers éléments supprimés.

2.2.2 Les motifs et la reconnaissance de forme

Une forme (pattern en anglais) représente une classe d'expressions. Elles sont composées de motifs élémentaires (pattern objects en anglais) qui représentent des ensembles possibles d'expressions. Un motif en Mathematica joue le même rôle qu'une déclaration de variable dans un langage de programmation comme C. Nous les utilisons donc dans les définitions des fonctions et aussi pour définir les règles de transformation.

Définition 3.4 : le motif quelconque « _ »
Le motif « _ » représente toute expression symbolique non vide. Nous pouvons le nommer en le précédant par un symbole par exemple « *x_* », et nous le lisons « le motif x quelconque ».

Définition 3.5 : le motif suite d'expressions symboliques quelconques « _ _ »
Le motif « __ » représente une suite d'expressions symboliques non vide. Nous pouvons le nommer en le précédant par un symbole comme « *x__* », et nous le lisons « le motif x suite d'expressions symboliques quelconques ». De sorte que *f[x__]* peut par exemple représenter *f[a, b, c],* avec *x* égal à la suite *[a, b, c].*

Définition 3.6 : le motif expression symbolique typée « *x_h* »
Le motif « x_h » représente une expression symbolique non vide qui a l'en-tête *h*. Le type *h* n'est pas nécessairement restreint aux types présentés dans le tableau 2. Le type *h* peut être une expression symbolique, mais ne peut pas être lui-même un motif. Par exemple f[x_List] représente f[{x1, x2}], mais elle ne représente pas f[3], car 3 n'est pas une liste mais un entier.

Définition 3.7 : le motif expression symbolique contrainte « *x_?P* »
Le motif « *x_?P* » représente toute expression symbolique non vide qui satisfait le prédicat *P*.
Par exemple nous définissons le prédicat :
PosQ[x_Integer] := x>0 ;
Ensuite, si nous écrivons f[x_ ?PosQ] ; alors cette forme représente seulement les entiers strictement positifs comme f[2] et ne représente pas f[-1].

Définitions 3.8 : une forme
Une forme est une expression symbolique constituée par au moins un motif élémentaire. Il est important de noter que les formes représentent des classes d'expressions ayant une structure déterminée. Le tableau 3 montre quelques exemples.

Définitions 3.9 : Correspondance de formes (Pattern matching)
Soit *f* une forme, construite à l'aide des motifs définis en 3.4 à 3.7. Une expression symbolique *exp* correspond à la forme *f* si on peut obtenir *exp* en remplaçant chaque motif de *f* par une sous expression de *exp* qu'elle représente.

Tableau 3 : exemples de quelques formes d'expressions algébriques.	
x_ + y_	Une somme de deux expressions symboliques.
a_Integer * x_	Une expression avec un multiplicateur entier explicite.
x_^ n_	L'expression x^n.
Real	**LES RÉELS.**

Il faut remarquer que même si deux expressions sont mathématiquement égales, elles ne peuvent être représentées par la même forme Mathematica que si leur structure est identique. Ainsi, par exemple, la forme *(1+x_)^2* peut représenter les expressions *(1+a)^2* ou *(1+b^3)^2* qui ont une structure identique. Toutefois, elle ne peut pas remplacer l'expression *1+2a +a^2*. Bien que cette expression soit mathématiquement égale à *(1+a)^2*, sa structure diffère de celle de la forme définie par *(1+x_)^2*.

Lorsque plusieurs motifs élémentaires de même nom apparaissent dans une forme, tous les objets doivent représenter le même élément. Ainsi *f[x_,x_]* peut représenter *f[2, 2]* mais pas *f[2, 3]*.

Les fonctions Mathematica suivantes aident à tester la correspondance entre une expression et une forme définie par des motifs.

- Cases[*{e1,e2,...}, pattern*] : donne la liste des éléments e_i qui correspondent à la forme *pattern*. Exemple : Cases[{2, x, 4}, _Integer] donne{2,4}.
- Count[*{e1,e2,...}, pattern*] : donne le nombre d'éléments e_i qui correspondent à forme *pattern*. Exemple : Count[*{* [{2, x, 4}, _Integer] donne 2.
- Position[*{e1,e2,...}, pattern*] : donne les positions des éléments e_i qui correspondent à la forme *pattern*. Exemple : Position[{x^2, 5, x^4}, x^_] donne {1,3}.
- MemberQ[*{e1,e2,...}, pattern*] : renvoie « True » si un élément e_i correspond à la forme *pattern*, et « False » dans le cas contraire. Exemple : MemberQ[{x^2, y^2}, x^_] donne « True ».
- Select[list, crit] : extrait tous les éléments e_i de la liste *list* pour lesquels crit[e_i] est « True ». Exemple : Select[{1,4,2,7,6,} EvenQ] donne {4,2,6}.

2.2.3 Les définitions des fonctions et des règles de transformation

Dans Mathematica, le fait que les motifs spécifient une forme générique pour la structure des expressions rend possible la définition de règles de transformation capables de modifier la structure des expressions, tout en les laissant mathématiquement égales. Il permet d'associer des définitions avec les fonctions et les règles de transformation. Par exemple, nous souhaitons définir la fonction mathématique suivante :

- f de R dans R.
- f(0) = 1
- $f(x) = 3x^2 + 2$

En Mathematica nous écrivons :

- *f[0] :=1 ;* cette définition indique que chaque fois que l'expression particulière f[0] apparaît, elle doit être remplacée par 1. Toutefois, la définition ne dit rien sur des expressions comme f[y] ou f[2].
- *f[x_] :=3 x^2 +2;* cette définition indique la valeur de la fonction *f* pour une expression de type quelconque. Donc, la valeur de f[y] est 3 y^2 + 2 et la valeur de f[2] est 14.

Si nous comparons f à une application, lorsque nous définissons des valeurs pour f[0], nous précisons l'image de cette application en un point particulier de son domaine. Le fait de définir une valeur pour f[x_] précise comment calculer l'image par f d'un point quelconque de son domaine .

Mathematica donne la possibilité de préciser explicitement le domaine de l'expression symbolique. Cette notion de domaine est purement algébrique et il ne faut pas la confondre avec la notion du type (l'en-tête d'une expression symbolique) qui est plutôt une notion informatique.

Définition 3.10 : le domaine d'une expression symbolique
Supposons qu'une expression symbolique $f[e_1,e_2,\ldots,e_n]$ est une fonction de n variables et supposons que : $y=f[e_1,e_2,\ldots,e_n]$, où chaque e_i prend ses valeurs dans un ensemble U_i et y prend ses valeurs dans un ensemble V. Nous écrivons alors : $f : U_1 \times U_2 \times \ldots \times U_n \rightarrow V$. l'ensemble $U_1 \times U_2 \times \ldots \times U_n$ de n-tuplets $(e_1,e_2,\ldots,e_n)$ est appelé le domaine de définition de la fonction f et l'ensemble V est appelé le domaine des valeurs de la fonction f. Mathematica prédéfinit les

domaines suivants :

- `Booleans` : le domaine des booléens {`True, False`}.
- `Integers` : le domaine des entiers Z.
- `Rationals` : le domaine des nombres rationnels Q.
- `Reals` : le domaine des nombres réels R.
- `Complex` : le domaine des nombres complexes C.
- `Primes` : le domaine des nombres premiers.
- `Algebraics` : le domaine des nombres algébriques A.

Par exemple l'expression de type `Symbol : y`. Nous pouvons l'associer au domaine `Rationals`. Nous montrons dans la section 1.6 comment cette association est utile dans la simplification algébrique.

Définition 3.11 : la définition d'une fonction

L'association d'une expression *expr* avec une fonction *f* est appelée la définition de la fonction, où les e_i sont des expressions symboliques ou des formes, et *expr* est nécessairement une expression symbolique. Nous pouvons définir une fonction de deux manières :

- Définition immédiate : $f[e_1,e_2,\ldots,e_n] = expr;$ dans ce cas l'expression symbolique *expr* est évaluée pendant la définition.
- Définition différée : $f[e_1,e_2,\ldots,e_n] := expr$; dans ce cas l'expression symbolique *expr* est évaluée seulement quand la fonction est invoquée.

La définition immédiate correspond au passage d'un argument par valeur, la définition différée correspond au passage d'un argument par référence.

Exemple :

Nous donnons ci-dessous quelques lignes d'un dialogue interactif entre l'usager et le noyau du système Mathematica. Les requêtes sont numérotées par Mathematica. `In[i]` est fourni par le noyau pour saisir la i[ème] commande. `Out[i]` renvoie la i[ème] évaluation.

```
In[1]:= x=100
Out[1]= 100
In[2]:= fx=x;
In[3]:= fy:=x;
```

```
In[4]:= fx==fy
Out[4]= True
In[5]:= x=1000
Out[5]= 1000
In[6]:= fx
Out[6]= 100
In[7]:= fy
Out[7]= 1000
In[8]:= fx==fy
Out[8]= False
```

Nous affectons dans `In[1]` la variable `x` par la valeur `100` en utilisant la définition immédiate. Dans `In[2]` `fx` est affectée par `x` en utilisant la définition immédiate, par contre `fy` dans `In[3]` est affectée aussi par `x` mais en utilisant la définition différée. La comparaison entre `fx` et `fy` donne `True` dans `Out[4]`. Ensuite, dans `In[5]` nous affectons `x` par la valeur `1000`. Nous remarquons que `fx` étant affectée immédiatement a gardé la valeur `100`. Par contre, `fy` étant définie différemment est à jour par la nouvelle valeur de `x`. La sortie `Out[8]` montre que `fx` et `fy` ne contiennent plus la même valeur.

Définition 3.12 : l'expression symbolique « Module »
L'expression symbolique Module[{x_1,x_2,...,x_n}, *body*] est utilisée pour regrouper un ensemble de définitions dans *body* de la manière suivante :

$$body ::= expr_1 \; ; expr_2 \; ;...expr_n \; ; Return[expr_f].$$

Toute définition d'une expression symbolique x_i est considérée comme locale dans *body*. La définition de toute autre expression est globale. L'expression $expr_f$ est la sortie de la fonction « Module ». Cette fonction est considérée comme une façon d'écrire des procédures dans Mathematica. Exemple :

```
g[x_] := Module[{t,u}, t = x+2; u = t^2 ;  return(2 u)]
```

g(2) donne 32.

Une fonction peut être aussi définie récursivement. Par exemple la factorielle est décrite facilement dans Mathematica :

- f[n_] := n f[n-1];
- f[1] = 1.

Ces deux définitions indiquent que f[n] doit être remplacée par n f[n-1], sauf dans le cas où n est

égal à 1, pour lequel f[1] doit tout simplement être remplacée par 1.

Définition 3.13 : une règle de transformation
Une règle de transformation est écrite :

$$lhs \ \text{->} \ rhs$$

lhs est une forme ou une expression symbolique, par contre *rhs* est exclusivement une expression symbolique. Cette règle transforme *lhs* en *rhs*. Comme dans le cas d'une fonction, les motifs et les formes sont utilisés pour spécifier une classe d'expressions. Par exemple, pour définir la règle de transformation mathématique ($R : x \rightarrow x^2$), nous écrivons en Mathematica :

R = (x_ -> x^2) ;

L'association de la programmation fonctionnelle, des motifs, et de la reconnaissance des formes, donnent au langage de Mathematica une grande puissance d'expression des fonctions symboliques et des systèmes de réécriture. La différence fondamentale entre une fonction et une règle est leur interprétation dans le noyau. La prochaine section décrit ce noyau d'évaluation et la différence entre une règle de transformation et une fonction.

2.3 Le noyau d'évaluation

2.3.1 Valeurs des symboles

Les langages de programmation traditionnels qui n'acceptent pas le calcul symbolique autorisent l'utilisation de variables seulement comme noms d'emplacements mémoires. Le but est de les utiliser pour stocker des valeurs purement numériques.

Dans Mathematica, un symbole comme x peut être utilisé comme variable affectée par une autre expression et comme un symbole autonome. Si nous associons dans Mathematica explicitement une définition pour x, comme x = 3, x sera alors toujours remplacé par 3, et ne pourra plus être utilisé comme un symbole autonome.

En réalité, la flexibilité de Mathematica provient en grande partie de sa capacité à combiner ces deux natures d'un symbole. Toutefois, il faut procéder avec rigueur dans ces différentes utilisations d'un symbole pour écarter tout risque d'erreur.

2.3.2 Principes de l'évaluation

L'évaluation est l'opération fondamentale que le noyau Mathematica exécute. Chaque fois qu'une expression symbolique est entrée, le noyau la réécrit en utilisant une procédure d'évaluation standard non bornée, jusqu'à l'obtention d'une forme standard ou canonique. Le noyau utilise pour cela toutes les définitions des fonctions qu'il connaît jusqu'à l'obtention d'un résultat auquel il ne peut plus appliquer de définitions. À ce moment-là, le noyau renvoie l'expression qu'il a obtenue comme résultat. Habituellement, ce résultat est une expression dans laquelle certains objets sont représentés sous une forme symbolique (symboles non interprétés).

Il est essentiel de mettre les expressions sous forme canonique, car la détermination de l'égalité de deux expressions devient évidente par correspondance de formes. On pourrait penser que toutes les expressions mathématiques sont réécrites automatiquement en une seule forme canonique standard, d'une manière ou d'une autre. Cependant, sauf pour les expressions les plus simples, il n'est pas toujours souhaitable d'avoir toujours la même forme standard. Par exemple, pour les polynômes, il existe deux formes standard adaptées à différents besoins. La première forme est une simple somme de termes, telle qu'elle serait produite dans Mathematica en appliquant la fonction *Expand*. C'est la forme la plus appropriée si nous avons besoin d'ajouter et de soustraire des polynômes. Toutefois, l'application de *Factor* permet d'écrire un polynôme sous forme d'un produit de facteurs irréductibles. Cette forme canonique est utile si nous souhaitons effectuer des opérations comme la division.

Une autre manière pour évaluer une expression est d'utiliser la substitution par règles de transformations. Dans ce cas, l'utilisateur définit un ensemble de règles et un algorithme d'évaluation basé sur la substitution pour transformer un ensemble d'expressions vers leurs formes canoniques.

Dans cette section, nous avons formalisé les terminologies et les fonctions prédéfinies dans le noyau Mathematica et qui sont destinées à cette tâche.

Définition 3.14 : la substitution de 1er ordre
Considérons une expression symbolique *expr* et une règle de transformation R=(*t* ->*v*), la substitution est le remplacement de toute occurrence de *t* par *v* dans *expr*. La fonction *Replace[expr, R]* est la fonction de substitution dans Mathematica. Nous pouvons écrire aussi

dans Mathematica *(expr /. R)* pour exprimer la substitution. Par exemple la fonction *Replace[x+1, x -> 2]* donne 3. Nous dirons aussi que nous appliquons la règle R sur expr.

Définition 3.15 : la substitution d'ordre *n*
Considérons une expression symbolique *expr* et *n* règles de transformation $\{R_1,R_2,...,R_n\}$, la substitution d'ordre *n* est l'expression Expr(n+1) obtenue par l'algorithme suivant :

Expr[1]= expr ;
For t=1 to n :
  Expr[t+1]= *Replace[Expr(t),* R_t*]*
End for ;
La fonction ReplaceListe[expr, $\{R_1,R_2,...,R_n\}$] implémente dans Mathematica la substitution d'ordre *n*.

Définition 3.16 : le point fixe d'une substitution
Considérons une expression symbolique *expr* et une règle de transformation *R*, le point fixe d'une substitution est atteint quand :
Replace[expr, R] == Replace[Replace[expr, R], R]
Nous dirons alors que l'expression *expr* demeure inchangée par *R : FP[expr, R]* .

Définition 3.17 : la substitution répétitive
Considérons une expression symbolique *expr* et une règle de transformation R, la substitution répétitive est l'algorithme suivant :

Expr = expr ;
Do
  $Expr_1$*= Replace[Expr, R] ;*
  Expr = $Expr_1$ *;*
Until *FP[*$Expr_1$*, R]*

Autrement dit, c'est l'application de R jusqu'à l'arrivée à un point fixe de la substitution. La fonction *ReplaceRepeated[expr, R]* ou *(expr //. R)* est la fonction de substitution répétitive dans Mathematica. Nous dirons qu'une substitution répétitive *ReplaceRepeated[expr, R]* est finie si l'algorithme précédente termine.

Définition 3.18 : l'évaluation simple d'une fonction

Soit une fonction *f* associée avec une seule définition : $f[e_1,e_2,\ldots,e_n] := expr$; l'évaluation simple de $f[x_1,x_2,\ldots,x_n]$ dans le noyau est équivalente à :

ReplaceRepeated[$f[x_1,x_2,\ldots,x_n]$, $f[e_1,e_2,\ldots,e_n]$ -> *expr]*

Définition 3.19 : l'évaluation d'une fonction multi définie

Soit *f* une fonction associée avec un ensemble de *m* définitions de la manière suivante :

$\{f[e_{11},e_{12},\ldots,e_{1n}] := expr_1, f[e_{21},e_{22},\ldots,e_{2n}] := expr_2, \ldots, f[e_{m1},e_{m2},\ldots,e_{mn}] := expr_m\}$

L'évaluation de la fonction $f[x_1,x_2,\ldots,x_n]$ dans le noyau est la réécriture de l'expression $f[x_1,x_2,\ldots,x_n]$ en utilisant l'ensemble de *m* règles de transformation suivante :

$R_{nf}=\{f[e_{11},e_{12},\ldots,e_{1n}] \text{ -> } expr_1, f[e_{21},e_{22},\ldots,e_{2n}] \text{ -> } expr_2, \ldots, f[e_{m1},e_{m2},\ldots,e_{mn}] \text{ -> } expr_m\}$

Le résultat de cette opération est une expression *expr*. Nous notons $expr=Evaluation[f,R_{nf}]$, cette évaluation est faite en Mathematica soit par la stratégie d'évaluation standard (algorithme d'évaluation par défaut) ou par une stratégie définie par l'utilisateur.

Nous pouvons aussi dire que l'évaluation de *f* est un problème de réécriture d'un terme *f* en utilisant l'ensemble des règles de réécriture R_{nf}. Deux considérations sont à prendre en compte pour ce genre de système : la convergence et la terminaison.

Définition 3.20 : la confluence d'une évaluation

Soient x et y deux expressions symboliques mathématiquement équivalentes : x ⇔ y. Une évaluation est confluente s'il existe *z* tel que $z=Evaluation[x, R_{nx}]$ et $z=Evaluation[x, R_{ny}]$. Nous appelons *z* la forme canonique de *x* et de *y*. Cette définition traduit la propriété de Church-Rosser pour un système de réécriture convergente (voir [Wol02] page 1036).

Définition 3.21 : la terminaison d'une évaluation

La terminaison d'une évaluation est produite quand l'algorithme d'évaluation termine.

Définition 3.22 : la convergence

Si pour tout couple d'expressions symboliques équivalentes leur évaluation termine et est confluente, l'évaluation est dite convergente.

2.4 Définir une stratégie d'évaluation

Mathematica donne la possibilité de définir des algorithmes d'évaluation dits non standard ou une stratégie d'évaluation. Cela est fait en utilisant des fonctions qui agissent comme les structures de contrôle telles que « While » et « For ». Nous pouvons aussi définir des attributs des fonctions.

2.4.1 Les structures de contrôle

Nous allons définir les structures de contrôle d'évaluation utilisées pendant cette étude *CompoundExpression*, *For,* et *While*.

Définition 3.23 : la fonction « CompoundExpression »

La fonction *CompoundExpression[*$expr_1$*,* $expr_2$*,...,*$expr_n$*]* évalue séquentiellement les $expr_i$ et renvoie comme résultat l'évaluation de la dernière expression $expr_n$. Nous pouvons utiliser l'écriture $expr_1$; $expr_2$; ...; $expr_n$ pour exprimer aussi l'exécution séquentielle.

Définition 3.24 : la fonction « For »

La fonction *For[start, test, incr, expr]* est la fonction de boucle. Elle est équivalente à l'instruction « for » du langage C : *for (start ; test ; incr) {expr}*.

Exemple :

```
In[1]:= For[i=0, i<3, i++, Print[X+i]]
Out[1] :=  1 + X
           2 + X
           3 + X
```

Définition 3.25 : la fonction « While »

La fonction *While[test, expr]* évalue *test*, puis *expr*, successivement jusqu'à ce que *test* ne renvoie plus True. Nous pouvons introduire la fonction *Break[ ]* à l'intérieur de la boucle. Dans ce cas, l'évaluation est abandonnée et Mathematica quitte la boucle *While*. Cela n'a d'intérêt que s'il y a une condition avec la fonction If.

Définition 3.26 : la fonction If

La fonction *If[cond, $expr_1$, $expr_2$]* évalue la condition *cond*, si *cond* est vraie elle évalue $expr_1$, si *cond* est fausse elle évalue $expr_2$. La deuxième expression est optionnelle.
Exemple :

```
In[1]:=
x=1;
While[x>0,x= x+1; If[x>5,Break[],Print[x]]]
Out[1] :=
2
3
4
5
```

2.4.2 Les attributs des fonctions

Ces sont des directives qui dirigent le noyau pendant l'évaluation de la fonction. La fonction « SetAttributes[*f, attributes*] » affecte à l'expression f l'ensemble d'attributs dans *attributes*. Lors de l'évaluation d'une fonction (simple ou non) la première opération que le noyau applique est de suivre les directives données par ses attributs.

Nous définissons seulement les attributs que nous avons utilisés.

Définition 3.27 : affectation d'un attribut
Soit *f* le nom d'une fonction et un attribut *attr*, nous affectons *attr* à *f* en utilisant la fonction *SetAttributes[f, attr].*

Définition 3.28 : l'attribut « Flat »
Lors de l'évaluation d'une fonction f affectée par l'attribut « Flat » la règle suivante est appliquée : *f[a…, f[c,d],b…] = f[a…,c,d, b…]*
Pendant l'évaluation, les parenthèses (ou les appels imbriqués de la fonction) seront supprimées pour obtenir la fonction sous une forme standard dite "Flat". Donc si une fonction est associée à l'attribut « Flat », elle est associative. L'exemple le plus connu est la fonction Plus. Si nous écrivons :
Exp1=Plus[x,Plus[y,z]] ;
Exp2=Plus[Plus[x,y],z] ;

Lors de l'évaluation les deux expressions seront transformées en Plus[x, y, z] car Plus est associé à l'attribut « Flat ».

Définition 3.29 : l'attribut HoldAll
Lors de l'évaluation d'une fonction de la forme $f[e_1,e_2,\ldots,e_n]$ affectée par l'attribut « HoldAll », les définitions et les valeurs associées aux e_i sont ignorées. L'attribut HoldAll empêche le noyau d'évaluer les arguments de la fonction.

La stratégie que nous avons implémentée pour la simulation symbolique est présentée dans le chapitre 5.

2.5 La stratégie d'évaluation standard

L'algorithme d'évaluation est décrit de façon très informelle dans le manuel de Mathematica [Wol03] et [Wol88] première-édition en 1988. Aussi, nous n'avons pas trouvé de détails techniques sur cet algorithme. Dans son livre « A New Kind of Science » [Wol02], le fondateur de Mathematica, S.Wolfram, a indiqué quelques informations et références sur cet algorithme d'évaluation standard. De même que le site de l'encyclopédie mathématique de E. W. Weisstein [Wei05] donne aussi quelques informations. Nous résumons ici toutes ces informations et donnons les références citées par ces deux sources.

L'algorithme d'évaluation de Mathematica est basé sur deux algorithmes classiques dans le domaine de la réécriture :

- L'algorithme de Knuth-Bendix publié pour la première fois dans [KnB70] : cet algorithme essaie de transformer un ensemble fini d'équations sous formes d'égalités des termes (x == y) vers un système de réécriture convergent. L'algorithme peut :
  - Terminer avec succès et donner un ensemble de règles convergent.
  - Terminer sans succès.
  - Boucler sans fin.
- L'algorithme de Buchberger [Buc70] pour la construction de la base de Gröbner : il est utilisé particulièrement pour résoudre les équations symboliquement par élimination des variables en respectant l'ordre lexicographique. Il est utilisé aussi pour obtenir la forme canonique pour un système de polynômes [BeW93].

Les détails techniques de ces deux algorithmes dépassent largement le cadre de notre étude.

De façon informelle, lors de l'évaluation d'une expression $f[x_1,x_2, \dots x_n]$, les étapes suivantes sont exécutées en respectant l'ordre :

- Évaluer l'en-tête *h* de l'expression, si *h* n'est pas une fonction.
- Évaluer chaque élément de l'expression $(x_1,x_2, \dots)$, sauf si *h* est affecté par un attribut qui contrôle l'évaluation telle que HoldAll.
- Si h a l'attribut Flat, aplatir toutes les expressions imbriquées ayant l'en-tête h.
- Prendre en considération tout autre attribut affecté à *h*.
- Utiliser les règles de transformation internes associées aux fonctions prédéfinies et qui sont des arguments pour h.
- Utiliser les définitions qui ont été données par l'utilisateur pour h, dans l'ordre spécifié par l'utilisateur.

2.6 Les fonctions de calculs algébriques

Le noyau simplifie automatiquement toutes les expressions symboliques de base telles que x+x ou 1-2 en appliquant des définitions algébriques internes (les propriétés de l'espace vectoriel des polynômes $K^n[x]$). Mais pour les expressions plus complexes, aucune simplification n'est faite de façon standard. Par exemple le noyau ne procède pas à la simplification du quotient de deux polynômes :

$$\frac{1 - x}{1 - x^2}$$

Des fonctions prédéfinies telles que *Simplify* appliquent des algorithmes d'évaluation spéciale et des règles de transformation algébriques pour obtenir la forme la plus simple d'une expression mathématique [Gra97]. Pour l'expression précédente, *Simplify* donne :

$$\frac{1}{1 + x}$$

Décider la forme "la plus simple" d'une expression est un problème complexe et généralement non résolu pour un nombre considérable de problèmes mathématiques. C'est à l'utilisateur de choisir cette forme simple et de définir les règles pour l'obtenir.

Définition 3.30 : la fonction « Simplify »
La fonction *Simplify[expr, assum]* effectue des simplifications sur *expr* en utilisant les hypothèses *assum*. Elle essaie de développer, de factoriser et d'effectuer d'autres transformations sur les expressions, et garde la forme la plus simple qu'elle peut obtenir.

Dans ce cas, l'utilisation des domaines est bénéfique. Prenons l'exemple suivant :
In[1]:= Simplify[Cos[2 x π] ,Element[x,Reals]]
Out[1]= Cos[2 π x]
In[2]:= Simplify[Cos[2 x π] ,Element[x,Integers]]
Out[2]= 1

Définition 3.31 : la fonction « FindInstance »
La fonction *FindInstance[equ, $\{x_1,x_2,\ldots,x_n\}$, dom]* applique un ensemble de règles de transformation dans le but de trouver un ensemble de solutions $S=\{x_1\text{->} val_1, x_2 \text{->} val_2, \ldots, x_n \text{->} val_n\}$ pour l'ensemble des équations *equ*, où $\{val_1, val_2, \ldots, val_n\} \in$ dom. Si l'ensemble *S* est vide, la fonction essaie de prouver que *equ* n'est pas satisfaisable. FindInstance Donne « *Fails* » dans le cas où la preuve est impossible.

La simplification et les preuves sur les résultats de simulation symbolique avec Simplify et Findinstance sont traitées dans le chapitre 6.

2.7 Conclusion

Pour résumer, Mathematica est un système de calculs symboliques avec un moteur de réécriture basé sur deux algorithmes bien établis dans ce domaine. C'est à l'utilisateur de vérifier la convergence de ses définitions et pas à l'outil. Nous avons pris conscience de cette difficulté dans l'utilisation de Mathematica durant l'implémentation de TheoSim.

Nous précisons dans chaque chapitre les règles et les algorithmes d'évaluation que nous utilisons pour la simulation symbolique et la vérification des circuits.

Chapitre 3

3 Modélisation de VHDL par un Système d'Equations Récurrentes (SER)

3.1 Le sous-ensemble VHDL

VHDL est un langage riche autant par sa syntaxe et sa sémantique que par sa capacité à exprimer, en terme de simulation dirigée par événements, des comportements séquentiels et concurrents. Nous désirons simuler symboliquement les circuits décrits à haut niveau d'abstraction. Pour atteindre cet objectif en trois ans, nous avons décidé de limiter notre modélisation à un sous-ensemble acceptable au niveau scientifique et au niveau industriel. Ainsi, ce sous-ensemble prend en compte le standard VHDL synthétisable finalisé en 2004 [réf], en écartant néanmoins quelques variétés syntaxiques qui ne sont pas très utilisées. Par contre, nous ajoutons quelques éléments qui le rendent plus expressif pour les descriptions abstraites. Dans cette section, nous décrivons d'une façon informelle ces éléments les plus remarquables. Une description formelle basée sur la grammaire de [IEEE04] est présentée comme rapport technique de TIMA [Als04].

Un circuit est décrit en VHDL par un couple (entité-architecture). Pour décrire un circuit hiérarchiquement, une ou plusieurs configurations peuvent être associées à l'entité-architecture. De manière rapide, les éléments syntaxiques et sémantiques qui apparaissent dans ce sous-ensemble sont :

- L'entité : les signaux d'interfaces sont déclarés dans l'entité et seules les directions (in, out) sont acceptées. Les paramètres génériques sont admis.

- Les objets : les constantes, les signaux, les variables sont considérés. Leurs valeurs initiales doivent être spécifiées dans un fichier de test séparé de la description du circuit. Nous ne considérons pas les valeurs par défaut pour permettre une valeur symbolique de l'objet lors de l'exécution.

- Les types de données : presque tous les types et sous-types sont considérés tels que les types énumérés, les booléens, les bits, les vecteurs de bits de taille fixe, les numeric bits signés et non signés, les entiers et les réels.

- Les paquetages : nous autorisons la définition des fonctions, des constantes et des types de données dans les paquetages VHDL. Par contre, les fonctions récursives et les fonctions qui contiennent l'instruction exit sont interdites.

- Les architectures : elles sont constituées de deux parties : déclarations et instructions.

    - Les déclarations sont exclusivement des déclarations de signaux, de constantes, de composants et de configurations internes. Les déclarations de fonctions et de types sont interdites : il faut les écrire dans des paquetages, et inclure ces paquetages , la puissance d'expression de VHDL est préservée.

    - La partie instructions décrit le comportement du circuit. Elle peut inclure des processus mémorisants, des processus combinatoires, des affectations concurrentes et des instanciations de composants. Ces composants ne peuvent communiquer entre eux que par l'intermédiaire de signaux déclarés dans l'architecture. Les variables partagées sont interdites. Cela garantit le fonctionnement déterministe du composant.

- Les processus : la liste de sensibilité d'un processus peut contenir tout type de signaux. La partie déclaration ne contient que des variables. Nous supposons qu'au plus une instruction « *wait until <front d'horloge>* » est présente dans la première ligne du processus. Par contre, l'attribut *event* peut être placé sans aucune restriction. Donc, la majorité des caractéristiques de la nouvelle norme pour la synthèse [IEEE 1076.6 2004] sont admises.

- Les instructions séquentielles : affectation de variables, affectation de signaux à délai nul, instructions conditionnelles if..then..else, case, et boucle for..loop sont admises dans les processus. Les appels de fonctions sont admis, mais les appels de procédures ne le sont pas.

- Les instructions concurrentes : les affectations de signaux à délai nul et les affectations conditionnelles, les appels de procédures ne contenant pas de « wait » et les instanciations de composants, où les chemins d'instanciations (port map) doivent être présents de façon explicite, sont traités.

- Les expressions : nous acceptons toutes les expressions arithmétiques et logiques comme l'addition, la multiplication, les comparaisons, et les opérateurs agissant sur les vecteurs (longueur par exemple).

3.2 Le modèle mathématique

3.2.1 Motivation

Un composant en VHDL est l'association d'une entité avec une architecture. Une entité peut avoir plusieurs architectures, où chaque composant doit être configuré avant la simulation. En effet, la configuration est la partie de la description qui relie une entité avec une architecture précise. Notre modélisation considère une entité configurée et génère la définition d'une fonction qui simule le comportement du composant pendant un cycle de simulation VHDL. Nous appelons les fichiers qui contiennent ces fonctions M-code.

La fonction porte le nom du couple entité-architecture correspondant. L'interface de cette fonction déclare l'ensemble des objets dans le composant : les signaux d'entrée, les signaux de sortie, les paramètres génériques, les signaux déclarés dans l'architecture, les variables des processus et les constantes. Chaque objet est représenté par un ou plusieurs motifs Mathematica. Tous ces motifs sont non-typés (ou de type quelconque) de façon qu'ils puissent être remplacés pendant l'exécution par un nombre ou par un symbole.

Le corps de la fonction globale de l'entité-architecture contient l'ensemble des fonctions de transitions, un pour chaque objet de la conception. L'idée est d'extraire automatiquement ces fonctions. Ces fonctions sont basées sur la notion de fonction conditionnelle et sur la notion d'équation de récurrence.

Notre modélisation est basée sur la simulation au niveau cycle de simulation. Donc, un objet S (un signal ou une variable) est observable sur un intervalle de temps : S(n) est la valeur d'un objet à la fin d'un cycle de simulation n. Dans la sémantique de VHDL synthétisable, la valeur future S(n+1) est calculable en fonction des valeurs précédentes de S. Donc, nous pouvons voir une affectation d'un objet sous la forme :

```
S (n+1) <= f(S(n), S(n-1), …)
```

L'idée que nous avons développée pour décrire l'évolution au cours du temps d'un ensemble d'objet $S_i(n)$ est de les modéliser comme un système d'équations récurrentes (SER) sur ***K***. Notre approche est inspirée d'un modèle présenté par Karp et al dans [KMW63].

Les notions de cette section seront utilisées pendant ce chapitre, ainsi que les chapitres suivants.

Définition 4.1 : domaines primitifs

Dans la suite, ***K*** est un symbole qui représente l'un des domaines primitifs :

- Les entiers ***Z***
- Les réels ***R***
- Les booléens ***B*** : l'ensemble des valeurs booléennes {1, 0} représenté aussi par {T,F} ou par {True, False}.
- Les vecteurs de bits $\boldsymbol{B}^n$: est l'ensemble $\{1, 0\}^n$ avec $n \in \boldsymbol{N}$.

3.2.2 Définitions et propriétés de la fonction Ife

Définition 4.2 : la fonction Ife

la fonction *Ife(x,y,z)* : $\boldsymbol{B} \times \boldsymbol{K} \times \boldsymbol{K} \rightarrow \boldsymbol{K}$ est définie de la manière suivante :

- *Ife(T, y, z) = y*
- *Ife(F, y, z) = z*

Cette fonction est l'équivalent mathématique des instructions séquentielles if-then-else. Nous la retrouvons dans les articles de McCarthy sur le langage LISP [McC60], ainsi que dans le rapport de recherche de Moore [Moo92]. Prenons l'exemple du morceau de code suivant :

```
If  x='1' then
 V := y ;
else

  V := z;
End if;
```

La fonction Ife modélise la valeur de la variable V : `V = Ife( x=1, y, z)`

La modélisation en fonctions Ife nous permet de profiter des résultats d'autres travaux dans le domaine de la vérification. Dans le rapport de J Strother Moore [Moo92] sur l'implémentation des BDD de Bryant [Bry86] dans le démonstrateur de théorème Nqthm [BoM97], nous retrouvons les théorèmes suivants :

Théorème 4.1 : la distribution de Ife

La fonction Ife est distribuable sur toute fonction f de la manière suivante :

$f[A_1,\ldots,Ife[x,y,z],\ldots,A_n] = Ife[x, f[A_1,\ldots,y,\ldots,A_n], f[A_1,\ldots,z,\ldots,A_n]]$

Exemple : si f est la fonction d'addition «*Plus* », et que nous avons une expression comme : $Plus[A_1,A_2,Ife[x,y,z]]$

Cette expression est égale à : $Ife[x, Plus[A_1,A_2,y], Plus[A_1,A_2,z]]$

Théorème 4.2 : la réduction de Ife

Dans une expression de la forme $Ife[x, a, b]$, où $x \in \boldsymbol{B}$, toute occurrence de x dans a est remplacée par *True* et toute occurrence de x dans b est remplacée par *False*.

Théorème 4.3 : IF-Y-Y :

L'expression de la forme $Ife[x, y, y]$ est remplacée par y.

La démonstration de ces théorèmes est immédiate à l'aide de preuves par cas [BoM97].

3.2.3 Système d'équations récurrentes (SER)

Définition 4.3 : une équation récurrente d'ordre n_o

Une équation récurrente d'ordre $n_o \in \boldsymbol{N}$ est une formule qui calcule la valeur d'une suite $U_n \in \boldsymbol{K}$, en un instant $n \in \boldsymbol{N}$, en fonction de n_o valeurs passées :

$U_n = f(U_{n-1}, U_{n-2},\ldots,U_{n-n_o})$

Définition 4.4 : un système d'équations récurrentes

Un système d'équations récurrentes d'ordre n_o, $\{S_i(n)\}_{0<i\leq m}$ est un ensemble fini d'équations récurrentes interdépendantes de la manière suivante :

$\{S_i(n)\}_{0<i\leq m} =\{ S_i(n) : n_o, i \in \boldsymbol{N}, 0 < i \leq m, S_i(n) = f_i(S_1(n-1), S_2(n-1), \ldots,S_m(n-1), S_1(n-2), S_2(n-2), \ldots,S_m(n-2), \ldots ,\ldots, S_1(n-n_o), S_2(n-n_o), \ldots,S_m(n-n_o))\}$

Pour la modélisation d'un composant VHDL, l'ordre maximal des équations est $n_o = 2$:

$\{S_i(n)\}_{0<i\leq m} = \{ S_i(n) : i \in \boldsymbol{N}, 0 < i \leq m, S_i(n) = f_i(S_1(n-1), S_2(n-1), \ldots,S_m(n-1), S_1(n-2), S_2(n-2), \ldots,S_m(n-2))\}$.

Une description d'un circuit VHDL synthétisable est en général représentée par un modèle de machine d'états finis de Mealy [Dus99]. Ce modèle est défini, pour une description synchrone, par FSM = $<I, O, E, s_0, \delta, \lambda>$ où :

I : est l'ensemble des signaux des entrées

O : est l'ensemble des signaux des sorties

E : est l'ensemble des variables d'états, c'est-à-dire les signaux et variables mémorisants dans le système

s_0 : est l'ensemble des valeurs initiales des états dans E

δ : est la fonction de transition d'état définie comme $\delta : I \times E \rightarrow E$

λ : est la fonction de la sortie définie comme $\lambda : I \times E \rightarrow O$

Le modèle SER d'un circuit $C = < I_c, K_I, S, K_S, \{S_i(n)\}_{0<i\leq m} >$ ressemble beaucoup au modèle FSM, avec $I = I_c$, $S = E \cup O$. Cependant, nous remarquons les différences suivantes :

- Dans une FSM, nous distinguons une fonction de transition et une fonction de sortie ; dans le SER, nous parlons d'un ensemble d'équations de récurrence.
- La FSM nécessite un ensemble d'états finis ; le SER est plus abstrait et les domaines peuvent être infinis.

La comparaison mathématique entre ces deux modèles dépasse le cadre de notre étude. Nous pouvons dire que le SER est une vision plus abstraite par rapport aux domaines et une vision plus détaillée par rapport au temps. Le passage d'un SER à une FSM est fait par une exécution de quelques cycles de simulation symbolique. Nous montrons ce passage dans 4.4.1.

3.3 Représentation d'une description VHDL

3.3.1 Les affectations des objets

L'affectation d'un signal calcule sa valeur future en connaissant les valeurs actuelles de tous les objets de la conception et l'existence d'événements sur un ou plusieurs signaux de la liste de sensibilité de l'affectation. Autrement dit, l'architecture est un système d'équations récurrentes d'ordre 2. Le choix de cet ordre est justifié car la norme VHDL pour la synthèse [IEEE 1076.6 1999] limite l'écriture des chronogrammes en partie droite de l'affectation des signaux à une valeur future (la clause « after » est ignorée). En outre, la nouvelle norme pour la synthèse [IEEE 1076.6 2004]] permet de prendre en compte des horloges multiples dans un processus, à l'aide de l'attribut event. Donc au plus trois valeurs sont nécessaires pour modéliser un objet S : une

pour la valeur future S(n+1), une pour la valeur actuelle S(n) et une pour la valeur précédente S(n-1).

Définition 4.5 : l'expression symbolique NextObj

Dans une architecture VHDL décrite à l'aide d'un ensemble de *m* objets $\{S_i(n)\}_{0<i\leq m}$ (signal ou variable), $S_i(n)$ est la valeur de l'objet *i* à la fin d'un cycle de simulation *n*. Cette valeur est décrite à l'aide des valeurs des objets dans $\{S_i(n)\}_{0<i\leq m}$ aux instants *n-1* et *n-2*. Nous écrivons : $NextObj(S_i(n),\ f_i(S_1(n-1),\ S_2(n-1),\ \ldots,S_m(n-1),\ \ S_1(n-2),\ S_2(n-2),\ \ldots,S_m(n-2)))$, pour représenter l'équation de récurrence $S_i(n) = f_i(S_1(n-1), S_2(n-1), \ldots,S_m(n-1),\ S_1(n-2), S_2(n-2), \ldots,S_m(n-2))$.

Nous associons la fonction *NextSig* à l'affectation d'un signal et la fonction *ChangeVar* pour l'affectation d'une variable. Cela rend le M-code plus lisible et permet de se reporter plus facilement au code VHDL d'origine.

Définition 4.6 : la fonction Event

La fonction Event modélise l'attribut *event* en VHDL, elle est définie de la manière suivante :

Event(S(n)) : $\boldsymbol{K} \rightarrow \boldsymbol{B}$

Event(S(n)) = S(n) ≠ S(n-1)

Donc, pour détecter un événement sur un signal *S* dans une affectation, la fonction Event impose que nous prenions en compte les valeurs *Sig(n-1)* et *Sig(n-2)*.

Exemple : nous considérons une affectation VHDL d'un signal Sig, de type booléen :

```
Sig <= clock'event and clock='1' ;
```

Nous écrivons alors :

Sig(n) = And(Event(clock(n-1)), clock(n-1)=1)

3.3.2 Les composants VHDL

Définition 4.7 : représentation SER d'un composant VHDL

Soit C =<I, K_I ,S ,K_S, $\{S_i(n)\}_{0<i\leq m}$ > un composant VHDL où :

I = {I1, I2,... Im1} : est l'ensemble des entrées. K_I est l'ensemble des domaines des entrées avec K_I =$\{K_{I1}, K_{I2}, \ldots K_{Im1}\}$.

Soit O = {O1, O2,... Om2} l'ensemble des sorties. K_O est l'ensemble des domaines des sorties avec $K_O = \{K_{O1}, K_{O2}, ...K_{Om2}\}$.
Soit V = {V1, V2,... Vm3} l'ensemble des variables et signaux internes. K_V est l'ensemble des domaines des variables et signaux internes avec $K_V = \{K_{V1}, K_{V2}, ...K_{Vm3}\}$.
$S = V \cup O = \{S_1, S_2, ... S_m\}$ avec m = m2+m3, et $K_S = K_V \cup K_O$

$\{I_i\}_{0<i\leq m1}$ est un ensemble de m1 valeurs en fonction seulement du temps de simulation *n*.
$\{S_i\}_{0<i\leq m}$ est un ensemble de m équations récurrentes par rapport au temps de simulation *n* et aux valeurs à *(n-1)* et *(n-2)* des éléments dans I et V.
Nous écrivons :
$\{S_i(n)\}_{0<i\leq m} = \{ S_i(n) : m, i \in \mathbf{N}, 0 < i \leq m, S_i(n) = f_i(I1(n-1), I2(n-1), ..., Im1(n-1), V1(n-1), V2(n-1), ..., Vm3(n-2), ... ,..., V1(n-2), V2(n-2), ..., Vm3(n-2))\}$, *où* f_i est la fonction de transfert de l'objet S_i.
L'exécution de l'ensemble des f_i représente un cycle de simulation.

3.3.3 Les types de données

Nous avons pris en compte, pendant la représentation des types, l'environnement de l'implémentation de l'outil : Mathematica. L'idée est de définir ou retrouver dans Mathematica la sémantique de chaque type de données VHDL, en conservant l'aspect non typé d'une expression symbolique. Pour ceci, nous ajoutons le type comme une information supplémentaire : domaine avec des contraintes. Dans le modèle, les types ne sont pas nommés en tant que tels. Seuls les objets d'un type sont représentés, sous la forme d'une liste associant l'identificateur de l'objet, un domaine primitif de valeurs, et éventuellement une ou plusieurs contraintes permettant de restreindre ce domaine.

Nous procédons comme en VHDL, deux catégories de types sont distinguées : les types scalaires et les types composés.

Les objets des types scalaires

Les types considérés sont les entiers, les réels et les types énumérés. Il existe des domaines proches dans Mathematica. Pour faire la correspondance entre Mathematica et les types VHDL, nous ajoutons davantage de contraintes. Ces contraintes sont utilisées ultérieurement pour simplifier les expressions de ce type pendant la simulation symbolique ou pendant les preuves sur les résultats symboliques.

Définition 4.8 : une contrainte

Soit ***K*** le domaine des entiers ***Z*** ou des réels ***R***. Une contrainte *c* est une expression symbolique de la forme c = ($expr_1 \diamond expr_2$), où $\diamond \in \{=, \neq, <, \leq, >, \geq\}$, et $expr_1, expr_2 \in K$. Donc, c représente une égalité ou une inégalité entre deux expressions symboliques qui sont construites à partir d'opérateurs arithmétiques (+, -, /, ×).

Définition 4.9 : représentation d'un objet de type scalaire VHDL

Un scalaire est représenté par (*S*, *Domaine*, *Contraintes*) où S est le nom de l'objet, *Domaine* est l'un parmi les domaines ou les types prédéfinis dans Mathematica, et *Contraintes* est un ensemble de contraintes *{c1, c2,...}*, si Contraintes ={} nous écrivons (*S*, *Domaine*).

Nous avons représenté les types suivants :

- **Les types entiers :** le domaine *Integer* en Mathematica a la même signification que celui de type integer en VHDL. Donc nous écrivons seulement (S, *Integer*). Le type natural ainsi que tout autre sous-type entier en VHDL est défini comme entier (*Integer*) avec une borne supérieure ou inférieure, ou les deux. Par exemple: une variable *b* de type Natural est définie par (*b*, *Integer*, {0≤ b}}.

- **Les réels :** le domaine *Real* est utilisé et nous écrivons : (S, *Real*).

- **Les types énumérés :** sont codés comme les sous-types entiers. De même, chaque élément est représenté par une constante du même nom. Autrement dit, un élément d'un type énuméré est un entier entre 0 et le nombre des éléments dans ce type. Par exemple, en VHDL nous écrivons :

```
type MULTI_LEVEL_LOGIC is (LOW, HIGH, RISING, FALLING, AMBIGUOUS);
```

Si b est une variable de ce type :

```
Variable b : MULTI_LEVEL_LOGIC;
```

Nous écrivons : (b, *Integer*, {0≤b, b≤4}). Les éléments symboliques de ce type (`LOW, HIGH, RISING, FALLING, AMBIGUOUS`) définissent des constantes comprises entre `0` et `4` dans l'ordre d'apparition :

```
LOW = 0 ;
HIGH = 1 ;
RISING = 2 ;
FALLING = 3 ;
AMBIGUOUS = 4 ;
```

- **Les bits :** en suivant le même raisonnement, un bit est modélisé comme un entier entre `0` et `1`. Une variable b de type bit est définie par (b, *Integer*, { 0 ≤ b, b ≤ 1}).
- **Les booléens :** le domaine des booléens est utilisé et les deux constantes prédéfinies en Mathematica sont « True » et « False » .

Les objets des types composés

Ce sont essentiellement les vecteurs de bits et les tableaux définis sur des types scalaires ou sur des vecteurs de bits. Nous avons défini l'expression symbolique `indexed` qui contient deux arguments : l'objet composé *S* et un naturel *i*, avec (*i* < la longueur de l'objet). Cette expression `indexed[S,i]` représente dans le M-code le $i^{ème}$ élément de l'objet *S* compté à partir de `0`. Les affectations par *NextSig* ou *ChangeVar* d'élément d'un objet composé se font uniquement avec cette expression. Cela est pratique pour l'implémentation de l'outil plutôt que l'utilisation directe de la fonction prédéfinie en Mathematica `Part` qui donne l'élément *i* dans une liste. Car contrairement à `indexed`, `Part` génère une erreur quand *i* est symbolique, ce qui est restrictif pour notre approche de simulation symbolique.

Définition 4.10 : représentation d'un objet composé VHDL
Un objet composé est représenté par (*indexed(S,i)*, *Domaine*, *Contraintes*) où *S* est le nom de l'objet composé, *i* est l'indice des éléments de *S*, *Domaine* est un parmi les domaines ou les types prédéfinis dans Mathematica et *Contraintes* est un ensemble de contraintes *{c1, c2,…}*, si *Contraintes* ={} nous écrivons (*indexed(S,i), Domaine*).

Exemple : en VHDL nous écrivons :

```
Signal S : bit_vector(3 downto 0);
```

S est représenté par :
(indexed(S,i), Integer, {0 ≤i, i ≤ 3, 0 ≤ indexed(S,i), indexed(S,i) ≤1}

Il existe des types composés définis dans les bibliothèques de VHDL comme le type numeric_bit. Pour utiliser ces types dans un composant, nous devons d'abord assurer que le paquetage qui les contient est acceptable par notre sous-ensemble. Ensuite, nous devons le compiler en M-code pour l'utiliser.

En VHDL, il est aussi possible d'affecter un objet de type composé sans préciser ses éléments un par un. Par exemple, nous affectons le signal déclaré au-dessus par :
S <= ″0001″;
Dans ce cas, nous affectons à S une liste d'entiers entre 0 et 1. Pour l'exemple précédent nous écrivons : NextSig(S(n+1), {0,0,0,1}). Dans le cas où un élément du vecteur de bits est utilisé, le lien entre cette affectation et la représentation par objet indexé `indexed(S,i)` est fait automatiquement et le code équivalent est :

```
NextSig(S(n+1), {0,0,0,1})
⇔
{NextSig(indexed(S(n+1),0), 1),
NextSig(indexed(S(n+1),1),  0),
NextSig(indexed(S(n+1),2),  0),
NextSig(indexed(S(n+1),3),  0)}
```

3.3.4 Illustration sur un exemple

Prenons l'exemple d'un composant *Counter* qui implémente un compteur simple. Le code VHDL est présenté dans la figure 4.1. La représentation SER de *Counter* est :

- Les entrées : I={clock, clear, count}
- Les domaines des entrées K_I={(Z,{clock≥0,clock≤ 1}),(Z, {clear≥0, clear≤ 1}),(Z, {count≥0, count≤ 1})}
- S= {Pre_Q, Q}

KS={(Z,{Pre_Q ≥0,Pre_Q ≤ 2^{len-1}}),(Z,{Q ≥0,Q ≤ 2^{len-1}})}
Le vecteur Pre_Q est traité en tant qu'un objet entier. Donc nous ne le découpons pas bit à bit et la fonction `indexed` n'est pas utilisée dans ce cas.

Les équations récurrentes du circuit sont :

```
{Si}0<i≤2 =
{NextSig(Pre_Q(n+1),
```

```
        Ife(Event(clock(n)
            ,Ife(clear(n)=1
                ,Pre_Q(n)-Pre_Q(n)
                 ,Ife( And(clock(n)=1, event(clock(n))
                       ,Ife(count(n) = 1
                           ,Pre_Q(n)+ 1
                           ,Pre_Q(n)
                           )
                      ,Pre_Q(n)
                      )
                 )
             ,Pre_Q(n)
             )
         )
,
NextSig( Q(n+1), Ife(event(Pre_Q(n)), Pre_Q(n), Q(n)   )
}
```

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_bit.all;

entity counter is
 generic(n: natural :=2);
 port(clock:      in std_logic;
      clear:      in std_logic;
      count:      in std_logic;
      q    :      out unsigned(n-1 downto 0)
     );
end counter;

architecture behv of counter is
  signal pre_q: unsigned(n-1 downto 0);
begin
  process(clock)
    begin
      if clear = '1' then
          pre_q <= pre_q - pre_q;
      elsif (clock='1' and clock'event) then
          if count = '1' then
            pre_q <= pre_q + 1;
          end if;
      end if;
  end process;
  q <= pre_q;
end behv;
```

Figure 4.1 : code VHDL d'un compteur simple

3.4 Extraction du modèle

3.4.1 Principe de la méthode d'extraction

L'objectif est de représenter l'architecture VHDL comme un système d'équations récurrentes, où chaque équation est écrite en utilisant une expression IF. Une expression IF peut être une constante, une expression booléenne, une expression arithmétique ou une fonction conditionnelle Ife. Nous donnons dans la figure 4.2 la définition de la liste d'affectations et la définition de l'expression IF. Pour des raisons de simplification, les syntaxes abstraites de l'expression booléenne (`expression _bool`) et de l'expression arithmétique (`expression _arith`) sont omises. La définition complète est écrite dans le rapport technique TIMA [Als04].

```
Seq_Assign ::=  nil
             | NextSig(s, IF)
             | ChangeVar(v, IF)
             | Seq_Assign1 ; Seq_Assign2

 IF ::= expression _bool
      | expression_arith
      | Ife(IF, IF1, IF2)
      | Fonction_VHDL(Ife1,…,Ifen)
```

Figure 4.2 : définition de l'expression IF.

Afin d'attaquer la complexité de VHDL, nous procédons par étapes. La première de ces étapes est l'extraction des domaines et des contraintes à partir des déclarations des objets dans le fichier VHDL. Ensuite, les équations de récurrence sont extraites à partir des :

- instructions séquentielles (wait, assert, affectation du signal, affectation de variable, for-loop, case et appels des fonctions).
- instructions concurrentes (affectations concurrentes, instanciation des composants, appels concurrents des procédures).

3.4.2 Normalisation des instructions séquentielles

Le but de la normalisation des instructions VHDL est de simplifier l'algorithme d'extraction. Un bloc d'instructions séquentielles (contenant des affectations d'objets, wait, Case, if-then-else, ...) est transformé en un bloc équivalant qui ne contient que des affectations d'objets et des if-then-else. La simplification est immédiate puisqu'un ensemble réduit d'instructions est considéré.

```
Seq ::=
target <= expression
| target := expression
| [label] assert condition [report expression] [severity expression]
| if condition then Seq [elsif_statement] endif
| case expression is case_alternative end case
| for identifier in expression1 direction expression2 loop Seq end loop
| null
| Seq1 ; Seq2
| [label] wait on sensitivity_list ; Seq
| [label] wait until condition ; Seq

elsif_statement ::=
else Seq
| elsif condition then Seq elsif_statement

case_alternative ::=
when choice => Seq
| when choice => Seq case_alternative

direction ::=
  to
| downto

choice ::= expression | others
```

Figure 4.3 : syntaxe abstraite des instructions séquentielles

Soit *expr* une expression définie selon la grammaire VHDL [IEEE 1076.6 2004], nous appliquons les deux principes suivants :

- Les affectations des objets :
  - L'affectation d'un signal (`S <=expr`) est écrite *NextSig(S(n+1), expr)*.
  - L'affectation d'une variable (`V <=expr`) est écrite *ChangeVar(V, expr)*.
- Toutes les autres instructions sont réécrites en if-then-else. La syntaxe des instructions séquentielles est donnée par la figure 4.3. Nous appliquons la fonction *Seq_rewrite* sur le

texte source, ce qui a pour effet d'appliquer récursivement un ensemble de règles de transformation pour obtenir les instructions normalisées (figure 4.4).

```
Seq_Norm ::=
NextSig(target(n+1), expression)
| ChangeVar(target, expression)
| if condition then Seq_Norm1 [else Seq_Norm2] endif
| Seq_Norm1 ; Seq_Norm2
```

Figure 4.4 : syntaxe abstraite d'instructions normalisées

Définition 4.11 : la fonction de normalisation *Seq_rewrite*

Soit $Sequentiel_{VHDL}$ le sous-langage des instructions séquentielles. C'est le langage engendré par le symbole non terminal *Seq* en appliquant la grammaire de la figure 4.3. Soit $Seq_Normalise_{VHDL}$ le sous-language construit par l'ensemble des instructions séquentielles primitives réduites. $Seq_Normalise_{VHDL}$ est le langage qu'on peut dériver à partir du non terminal *Seq_Norm*.

Seq_Rewrite : $Sequential_{VHDL}$ → $Seq_Normalise_{VHDL}$, avec:

1. Pour les affectations :
   - ***Seq_Rewrite***`(S <= expression)` → `NextSig(S ,expression)`
   - ***Seq_Rewrite***`(V := expression)` → `ChangeVar(V, expression)`
2. Pour les assertions : ***Seq_Rewrite***`([label]` **`assert`** `condition [`**`report`** `expression]` **`severity`** `expression])` → **`if`** `condition` **`then`** `label:=true` **`else`** `label:= false` **`endif`**
3. Pour les instructions conditionnelles :
   - ***Seq_Rewrite***`(`**`if`** `condition` **`then`** `Seq [`**`elsif`**`_statement]` **`endif`** `)` → **`if`** `condition` **`then`** ***Seq_Rewrite***`(Seq) [`**`else`** ***Seq_Rewrite***`(elsif_statement)]` **`endif`**
   - ***Seq_Rewrite***`(else Seq)` → ***Seq_Rewrite***`(Seq)`

- ***Seq_Rewrite*(elsif** condition **then** Seq elsif_statement) →
  if condition **then** ***Seq_Rewrite***(Seq) **else** ***Seq_Rewrite***(elsif_statement)
  endif

- ***Seq_Rewrite*(case** expression **is when** choice => Seq **end case**) →
  if (expression = choice) **then** ***Seq_Rewrite***(Seq) **endif**

- ***Seq_Rewrite*(case** expression **is when** others => Seq **end case**) →
  Seq_Rewrite(Seq)

- ***Seq_Rewrite*(case** expression **is when** choice => Seq case_alternative
  end case) → **if** (expression = choice) **then** ***Seq_Rewrite***(Seq) **else**
  ***Seq_Rewrite*(case** expression **is** case_alternative **end case**) **endif**

4. Pour une suite d'instructions **:** ***Seq_Rewrite***(Seq_1 ; Seq_2) →
   Seq_Rewrite(Seq_1) ; ***Seq_Rewrite***(Seq_2)

5. Pour l'instruction de synchronisation **wait** :

- ***Seq_Rewrite***([label] **wait on** sensitivity_list ; Seq) →
  if ***Event***(sensitivity_list) **then** ***Seq_Rewrite***(Seq) **endif**

- ***Seq_Rewrite***([label] **wait until** condition ; Seq) →
  if ***Event***(condition) **and** condition **then** ***Seq_Rewrite***(Seq) **endif**

6. Pour les boucles (le cas où $exp_1>exp_2$) : ***Seq_Rewrite*(for** identifier **in** $expression_1$ to $expression_2$ **loop** Seq **end loop**) →

   identifier := $expression_1$; ***Seq_Rewrite***(Seq);
   identifier := succ($expression_1$); ***Seq_Rewrite***(Seq);
   …
   identifier := $expression_2$; ***Seq_Rewrite***(Seq)

   Quand la direction est **downto**, la fonction prédécesseur pred remplace la fonction successeur succ.

L'implémentation dans Mathematica de cette normalisation est faite à l'aide des opérateurs et fonctions que nous avons définis dans le chapitre 3. L'équivalence entre les instructions

séquentielles originales et normalisées est prouvable par analyse de cas. Nous pouvons dire aussi que la normalisation se termine car le nombre de transformations appliquées est fini (les règles sont appliquées de façon finie).

3.4.3 Extraction des équations récurrentes à partir des instructions normalisées

Cette section présente la méthode d'extraction des équations récurrentes qui constituent la sémantique d'un ensemble d'instructions normalisées. Nous prenons un point de vue plutôt implémentation que théorique.

Définition 4.12 : la liste d'instructions simple
Une liste d'instructions est dite simple si elle contient exclusivement une des deux instructions suivantes :

- NextSig(S(n+1), expr)
- ChangeVar(V(n),expr)

Définitions 4.13 : la liste d'instructions composée
Une liste d'instructions est dite composée si elle est de la forme {inst1,inst2,...instn}, où insti est une des instructions suivantes:

- NextSig(S(n+1), expr)
- ChangeVar(V(n),expr)
- If(cond, liste1,liste2), liste1 et liste2 sont aussi des listes normales.

Définition 4.14 : la réduction d'une liste d'instructions simple
Considérons une liste d'instructions simple L = {NextObj(Obj1, expr1), NextObj(Obj2, expr2), ..., NextObj(Objn, exprn)}. La réduction de L est l'application de la sémantique VHDL. Pratiquement, nous appliquons les deux principes suivants :

- L'affectation d'une variable prend son effet immédiatement.
- Dans le cas de multiples affectations d'un même signal, seule la dernière affectation est considérée.

L'algorithme de réduction est le suivant, nous utilisons dans ce pseudo code la syntaxe et les fonctions de Mathematica décrites dans le chapitre 2 :

L = {NextObj(Obj₁, expr₁), NextObj(Obj₂, expr₂), ..., NextObj(Objₙ, exprₙ)} ;

```
L_réduite ={NextObj(Obj_1, expr_1)}
L_règles ={Obj_1 -> expr_1}
For i=1 to n-1
    Expr_réduite = ReplaceList[exp_i+1, L_règles] ;
    AddToListe[L_réduite, NextObj(Obj_i+1, Expr_réduite)] ;
    UpdateListe[L_règles, Obj_i+1 -> Expr_réduite] ;
End for ;
L_réduite, simplifiée =
   ReplaceRepeated[Liste_réduite,
   {x___, NextObj(Obj,expr_1),y___, NextObj(Obj,expr_2), z___} -> {x, y, NextObj(Obj,expr_2)}] ;
```

Nous commençons avec une liste *L* contenant *n* affectations séquentielles. Au début, la liste réduite $L_{réduite}$ contient la première affectation dans L et la liste de règles de substitution $L_{règles}$ contient la règle obtenue de cette première affectation. Nous exécutons une boucle qui parcourt la liste L. A l'étape *i* de la boucle :

- l'expression exp_{i+1} de l'affectation (i+1) dans L est réduite en appliquant $L_{règles}$. L'expression obtenue est $Expr_{réduite}$
- L'affectation, *NextObj*(Obj_{i+1}, $Expr_{réduite}$) est ajoutée à $L_{réduite}$
- La fonction UpdateList ajoute la règle Obj_{i+1} -> $Expr_{réduite}$ dans l'ensemble de règles courant $L_{règles}$, et ôte de toute règle ayant $_{i+1}$ en partie gauche qui s'y trouvait déjà.

Le résultat de la boucle est la liste $L_{réduite}$ qui contient les affectations réduites mais pas simplifiées. L'étape suivante consiste à garder une seule affectation pour chaque objet. Cela revient selon la sémantique VHDL à garder la dernière affectation de l'objet. L'application de cette simplification sur la liste $L_{réduite}$ donne la liste $L_{réduite,\ simplifiée}$. C'est le résultat final de l'algorithme. La fonction *SimplifyAndReduce[L]* applique cet algorithme sur une liste *L*.

Définition 4.15 : la simplification d'une liste d'instructions composée

La simplification d'une liste d'instructions composée consiste à la transformer en une liste d'instructions simple ; pour cela il faut transformer chaque instruction *If(cond,* $liste_1$*,*$liste_2$*)* en une liste d'instructions simple. L'algorithme de simplification considère toutes les combinaisons possibles d'instructions. Il est constitué des actions suivantes :

- Réduire les listes simples.
- Prendre en considération la mémorisation de VHDL.
- La distribution de l'expression *Ife*.

Exemple 1 : pour expliquer l'algorithme, nous commençons par donner un exemple de mémorisation dans une instruction if-then-else :

```
If  clock'event and clock ='1'  then
  Sig <= val ;
end if ;
```

Dans un cycle de simulation *n*, si la condition de l'instruction *if* est vraie, la valeur du signal *Sig* au cycle de simulation *n+1* est la valeur du *val* au cycle *n*, et nous écrivons : *NextSig(Sig(n+1), val(n))*. Par contre si la valeur de la condition est fausse, une mémoire est créée et nous écrivons *: NextSig(Sig(n+1), Sig(n))*. Pour représenter les deux comportements quelle que soit la valeur de la condition, nous utilisons la fonction Ife avec la distribution et nous obtenons : *NextSig(Sig(n+1), Ife(And(Event(clock), Equal(clock(n),1)), val(n), Sig(n))).*

Exemple 2 : nous modifions l'exemple précédent et nous ajoutons une valeur par défaut de la manière suivante :

```
Sig <= '0';
 If  clock'event and clock ='1'  then
   Sig <= val ;
end if;
```

Dans ce cas, la mémorisation de la branche fausse selon la sémantique de VHDL n'est plus celle de la valeur Sig(n), mais celle de l'affectation précédente. Autrement dit, le code est équivalent à :

```
If  clock'event and clock ='1'  then
   Sig <= val ;
```

```
else
   Sig <= '0';
End if ;
```

Après la distribution de la fonction Ife nous obtenons :
NextSig(Sig(n+1), Ife(And(Event(clock), Equal(clock(n),1)), val(n), 0)).

Exemple 3 : nous compliquons l'exemple de la manière suivante :

```
If  clock'event and clock ='1' then
   V := val ;
   Sig <= '1';
   Sig <= V +1 ;
else
   Sig <= '0';
end if ;
```

Dans ce cas, nous commençons par réduire la liste de la branche vraie :

{ V :=val, Sig<='1', Sig<=V+1 } devient { ChangeVar(V(n) ,val(n)), NextSig(Sig(n+1),val(n) +1)}

Ensuite, nous considérons la mémoire créée du fait de l'absence d'une affectation de la variable V dans la branche fausse. Puis nous distribuons la fonction Ife et nous obtenons :

{ChangeVar(V(n), Ife(And(Event(clock), Equal(clock(n),1)), val(n), V(n))), NextSig(Sig(n+1), Ife(And(Event(clock), Equal(clock(n),1)), val(n), 0))}.

Nous obtenons après ces transformations l'expression future de la variable V. Donc nous remplaçons dans la partie gauche de la fonction *ChangeVar* V(n) par V(n+1). Les équations récurrentes suivantes sont obtenues :

{ChangeVar(V(n+1), Ife(And(Event(clock), Equal(clock(n),1)), val(n), V(n))), NextSig(Sig(n+1), Ife(And(Event(clock), Equal(clock(n),1)), val(n), 0))}.

Exemple 4 : nous donnons un dernier exemple avant d'expliciter l'algorithme général :

```
If  clock'event and clock ='1' then
   Sig <= '1';
   If reset = '1' then
```

```
      V := val ;
      Sig <= V + 1 ;
   end if ;
 else
   Sig <= '0';
end if ;
```

Ce bloc d'instructions est de profondeur 2 (instruction if dans une instruction if). Pour le simplifier nous commençons par le niveau de profondeur 2 (le niveau le plus profond). La première étape est l'introduction des mémoires et la réduction des listes de niveau 2:

```
If reset = '1' then
      V := val ;
      Sig <= val + 1 ;
   else
     V := V(n) ;
     Sig <= '1';
end if ;
```

Ensuite, nous simplifions le niveau 2 et nous obtenons :

{
ChangeVar(V(n), Ife(Equal(reset(n),1), val(n), V(n))),
NextSig(Sig(n+1), Ife(Equal(reset(n),1), val(n) + 1, 1))
}

Une vision de l'exemple après cette première étape :

```
If  clock'event and clock ='1' then
   Sig <= '1';
    {
      ChangeVar(V(n+1), Ife(Equal(reset(n),1),  val(n), V(n))),
     NextSig(Sig(n+1),  Ife(Equal(reset(n),1), val(n) + 1, 1))
    }
 else
   Sig <= '0';
end if ;
```

Nous recommençons les mêmes opérations au premier niveau :

- Introduction des mémoires

- Réduction des listes

```
If  clock'event and clock ='1' then
    {
      ChangeVar(V(n), Ife(Equal(reset(n),1),  val(n), V(n))),
     NextSig(Sig(n+1),  Ife(Equal(reset(n),1), val(n) + 1, 1))
    }
 else
   V := V(n)
   Sig <= '0';
end if ;
```

- Distribution de la fonction Ife.

```
{
   ChangeVar(V(n+1), Ife(And(Event(clock), Equal(clock(n),1)),
                         Ife(Equal(reset(n),1),  val(n), V(n))),
                        V(n)) ),
  NextSig(Sig(n+1), Ife(And(Event(clock), Equal(clock(n),1)),
                       Ife(Equal(reset(n),1), val(n) + 1, 1)),
                        0))
 }
```

Le résultat final est la liste simplifiée et réduite de la liste initiale. Cette liste contient les équations récurrentes que nous cherchons.

Définition 4.16 : profondeur d'une liste composée
La profondeur d'une liste est le niveau d'imbrication des instructions if-then-else. Si nous comparons la liste avec un arbre, le niveau 1 est la racine, et le niveau n est la feuille. Naturellement, la liste de niveau n est une liste d'instructions simple.

Définition 4.17 : extraction des équations récurrentes
Considérons une liste d'instructions composée L de profondeur n. Pour des raisons de simplification et de clarté nous donnons l'algorithme suivant :

$Trans_{Seq}$(L) =
For i=n downto 0
IntroductionMémoire[i, L] ;
RéductionListe[i, L] ;
DistributionIfe[i, L] ;
End for ;
Return[SimplifyAndReduce[L]]

La hiérarchie de chaque instruction If-then-else est parcourue pour obtenir une liste d'affectation simple. Ensuite, une dernière simplification et réduction est nécessaire pour obtenir les équations récurrentes. L'implémentation de cet algorithme est faite en Mathematica par des fonctions récursives.

3.4.4 Normalisation d'instructions concurrentes

La fonction *$Trans_{conc}$* est appliquée sur la partie qui contient les instructions concurrentes dans les architectures. Elle produit une liste d'instructions concurrentes en distinguant les cas suivants :

Le processus : le 1er cas est le processus avec une liste de sensibilité. Selon la sémantique VHDL, un processus est exécuté seulement si un événement est détecté sur un ou plusieurs signaux dans sa liste de sensibilité. Donc, naturellement, nous la modélisons par la forme :

```
TransConc([label:] process [(sensitivity_list)] [is] process_declarative_part
begin Seq_Norm end process [label]) →
 Ife(Event (sensitivity_list), Update(Distribute (TransSeq(Seq_Norm))))
```

Ensuite nous procédons de façon similaire à la partie précédente pour extraire les équations récurrentes : introduction des éléments mémoires et distribution de la fonction Ife.

L'instanciation du composant : le modèle SER est conçu pour traiter les descriptions structurelles comme des appels de fonctions. Chaque composant est représenté par une fonction qui regroupe un ensemble d'équations récurrentes du composant. Dans ce cas, l'effort de simulation et de vérification est divisé.

```
TransConc(label: component_name [generic map (association_list1)]
      [port map (association_list2)]) →
SER(component_name) .//  (association_list1  → association_list2)
```

La fonction est appliquée sur les objets locaux considérés dans l'instanciation. Il s'agit des raccordements entre les ports formels et les ports effectifs donnés sur la liste des associations association_list en utilisant la fonction ReplaceRepeated (.//) décrite dans le chapitre 2.

La description engendrée par For…generate : le traitement de cette instruction est déterminé comme dans le cas d'une boucle, avec la condition que sa borne ait une valeur numérique précisée avant la génération du modèle. Donc, l'instanciation générée est éclatée en substituant les i numériquement à chaque fois :

```
TransConc(label: for identifier in expression1 to expression2
      generate Conc end generate [label]) →
Remplacer identifier avec expression1 dans TransConc(Conc);
Remplacer identifier avec succ(expression1) dans TransConc(Conc);
…
Remplacer identifier avec expression2 dans TransConc(Conc)
```

Nous supposons ici que ($expression_2 > expression_1$).
De façon similaire quand la direction est downto.

L'affectation : qui est éventuellement conditionnelle. En fait, une affectation concurrente peut être remplacée par un processus équivalent. La liste de sensibilité de ce processus équivalent est la liste des signaux dans l'affectation concurrente. La fonction Sensitivity retourne la liste des signaux dans cond_expression. Donc, une instruction est modélisée par une instruction conditionnelle de la forme :

```
 TransConc(target <= cond_expression) →
NextSig(target(n+1), Ife(Event(Sensitivity
(cond_expression)),TransConc(cond_expression), target(n)))
```

Nous appliquons ensuite les règles suivantes :

```
    TransConc(expression when condition else conditional_forms) →
     Ife(condition, expression,  TransConc (conditional_forms) )
```

```
TransConc(with expression select target <= selected_forms) →
 NextSig(target(n+1),TransConc(with expression select selected_forms))

TransConc(with expression1 select expression2 when others) →
 expression2

TransConc(with expression1 select expression2 when choice) →
 Ife(expression1=choice,  expression2 ,target(n))

 TransConc(with expression1 select expression2 when choice selected_forms)
 →
 Ife( expression1 = choice , expression2 , TransConc(with       expression1
 select selected_forms) )


TransConc(expression when condition else conditional_forms) →
 Ife( condition , expression , TransConc(conditional_forms) )
```

Le résultat de l'application de ces règles est une équation récurrente qui décrit la valeur du signal affecté.

3.5 Implémentation dans Mathematica

3.5.1 Le compilateur vers TheoSim

L'outil TheoSim est basé sur la modélisation SER de VHDL décrite dans ce chapitre. Nous avons implémenté un traducteur afin de rendre automatique la génération de SER (dans un fichier appelé M-code) d'une description VHDL. Cela rend la phase d'expérimentation de la simulation symbolique plus rapide car nous évitons à chaque fois d'extraire manuellement le M-code (le SER d'un composant). La figure 4.5 montre la vue d'ensemble de ce prototype. La traduction est faite en deux étapes : la compilation frontale et l'élaboration du modèle.

La compilation frontale prend en entrée un fichier source VHDL qui a préalablement été soumis à une première phase de compilation (vérification syntaxique et contextuelle) à l'aide d'un outil

industriel disponible dans l'équipe. Il n'est donc pas utile de prévoir le traitement des erreurs statiques. La description VHDL est traduite dans le format intermédiaire LIF. C'est une représentation préfixée de VHDL, qui ressemble à la syntaxe du LISP. La définition complète de LIF est faite par notre équipe [AlT03].

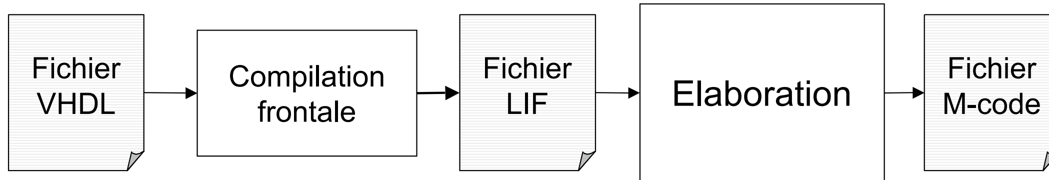


Figure 4.5 : vue d'ensemble du traducteur de VHDL vers le M-code

L'élaboration est la phase de génération du modèle simulable (le SER dans notre cas). Nous appliquons la méthode de génération du M-code que nous avons décrite dans ce chapitre pour générer ce modèle. Cependant, pour des raisons de performance, nous adoptons quelques différences entre les codes générés et le modèle SER. Par exemple, selon la sémantique VHDL, un processus est exécuté seulement si un événement est détecté sur un ou plusieurs signaux de sa liste de sensibilité. Dans l'implémentation, nous avons préféré la forme :

Process[Label, If [Event [sensitivity_liste], TransSeq [Seq_Norm]]].

plutôt que d'introduire des éléments mémoires et de distribuer la fonction Ife, car à ce niveau, la mémorisation est implicite dans l'environnement d'implémentation (Mathematica). Pour montrer un exemple de sortie de notre implémentation, nous reprenons l' exemple du compteur dans la figure 4.1. Son modèle M-code est le suivant :

```
Clear[counter$behv];  (* Initialisation du symbole counter$behv *)

SetAttributes[counter$behv, HoldAll];  (* Empêcher d'évaluer les arguments de la
fonction*)

counter$behv[clear_, clear$0_, clock_, clock$0_, count_, count$0_, n_,
pre$q_, pre$q$0_, pre$q$1_, q_, q$1_]:=
(* Déclaration de l'en-tête de la fonction qui simule le composant  *)
```

```
Module[{},

Process[process$13, If[Wait[clock, count, clear],
        (* traduction du 1er processus *)
{start[]
,
(* l'équation récurrente du signal pre_q*)
NextSig[pre$q$1,
       Ife[equal[clear, 1],
       0      ,
       Ife[and[equal[clock, 1], event[clock]],
         Ife[equal[count, 1],
           1 + pre$q          ,
           pre$q$1          ]
         ,
         pre$q$1        ]
       ]
]
}]];
(*fin de process  *)

Process[process$22, If[Wait[pre$q],
(* traduction du processus équivalent à l'instruction concurrente *)
{start[]
,
(* l'équation récurrente de q *)
NextSig[q$1,
       pre$q]
}]];   (*fin de process *)
(* Prendre en considération les types *)
UpDate[generic[natural[n]], in[std$logic[clock], std$logic[clear],
std$logic[count]], out[BitVector[q, downto[{minus, n, 1}, 0]]],
Signals[BitVector[pre$q, downto[{minus, n, 1}, 0]]]];
];
```

(* fin du fichier M-code *)

Nous remarquons dans le M-code du compteur que les instructions concurrentes sont précédées par l'en-tête Process suivi par un label unique. Il ne s'agit pas d'une instruction mais plutôt d'un étiquetage qui aide à lire et à distinguer les processus concurrents. Une autre remarque est la disparition de l'expression (Pre_Q - Pre_Q), présente dans le code VHDL de la figure 4.1, qui est remplacée par le résultat de sa simplification symbolique. Nous précisons que la fonction UpDate est utilisée pendant la simulation pour simplifier les expressions symboliques des objets de la conception selon leurs types. La lettre « _ », ayant une signification particulière en Mathematica, est remplacée par « $ ».

3.5.2 Les paquetages VHDL

Les définitions des types ainsi que les définitions des fonctions et des procédures sont souvent regroupées dans des paquetages VHDL. Cette notion de bibliothèque est très utilisée dans l'industrie. Il existe plusieurs paquetages standard qui sont à la fois nécessaires pour la compilation des nouvelles descriptions et pour leur simulation. Nous avons procédé de façon similaire pour le traitement des paquetages. Deux modèles sont générés pour chaque paquetage :

- Le premier est un modèle statique utilisé pour la génération du M-code d'un composant qui importe ce paquetage. Dans cette partie du paquetage, nous retrouvons la partie déclaration d'un paquetage VHDL : déclarations des types, des constantes et des interfaces des fonctions.
- Le deuxième modèle est dynamique, nécessaire dans le cas où on souhaiterait simuler le composant en interprétant le paquetage. Cette partie du paquetage contient exclusivement les interfaces et les corps des fonctions.

La figure 4.6 montre un exemple d'une déclaration d'un paquetage VHDL, son modèle statique est présenté dans la figure 4.7. L'annexe 3.1 montre un exemple plus important d'un paquetage VHDL. Ses modèles statiques et dynamiques sont présents dans l'annexe 3.4 et 3.5.

```
package fonctions is
function ch(e,f,g : in std_logic_vector(31 downto 0))return
std_logic_vector;
function maj(a,b,c : in std_logic_vector(31 downto 0))return
std_logic_vector;
function add2(h,e,ch1,a,maj1,wi32,ki32 : in std_logic_vector(31
downto 0))return std_logic_vector;
function shift(x : in std_logic_vector(31 downto 0))return
std_logic_vector;
function ajout2(x,y : in std_logic_vector(31 downto 0))return
std_logic_vector;


type operation is  (opodoo,kangoo,fastoo);

constant idle: std_logic_vector(2 downto 0):="000";
constant init: std_logic_vector(2 downto 0):="001";
constant sha_ini_one:std_logic_vector(2 downto 0):="010";
constant calculw_one: std_logic_vector(2 downto 0):="011";
constant calcul_abc_one: std_logic_vector(2 downto 0):="100";
constant result: std_logic_vector(2 downto 0):="101";
constant resultw:std_logic_vector(2 downto 0):="110";
end ;
```

Figure 4.6 : une définition du paquetage fonctions VHDL

```
FunctionQ[ch]:= True;
FunctionQ[maj]:= True;
FunctionQ[add2]:= True;
FunctionQ[shift]:= True;
FunctionQ[ajout2]:= True;

CodeType[opodoo, kangoo, fastoo];

ChangeVar[idle, {0, 0, 0}];
ChangeVar[init, {0, 0, 1}];
ChangeVar[sha$ini$one, {0, 1, 0}];
ChangeVar[calculw$one, {0, 1, 1}];
ChangeVar[calcul$abc$one, {1, 0, 0}];
ChangeVar[result, {1, 0, 1}];
ChangeVar[resultw, {1, 1, 0}];
```

Figure 4.7 : extrait du modèle statique du paquetage fonctions.

3.5.3 Représentation d'une fonction

Dans notre modélisation, les sous-programmes sont considérés comme des opérations et leurs principales restrictions par rapport à VHDL sont :

- Les procédures ne contiennent pas l'instruction `wait`.
- Les fonctions sont pures.
- Les fonctions ne contiennent qu'un seul `return`.

Dans la notion de SER, la définition d'une fonction ou d'une variable VHDL est mathématiquement similaire, dans le sens où toutes les deux sont définies par une équation récurrente et leur valeur est immédiatement considérée pendant l'extraction. En effet, la sémantique des instructions séquentielles est la même pour les fonctions et les processus. Dans notre modèle, une fonction donne une sortie unique qui est indiquée explicitement par : Return(sortie). Donc, modéliser une fonction revient à extraire une seule équation récurrente qui représente sa sortie.

La procédure à son tour n'est pas fondamentalement différente de la représentation d'un composant et peut être vue comme un composant qui ne contient qu'un seul processus.

Exemple : la fonction adressage qui prend un entier entre 0 et 3 et calcule le vecteur de bits équivalents :

```
function adressage(t:in integer)return std_logic_vector is
      variable x : std_logic_vector(1 downto 0);
begin
   case t  is
          when 1  =>   x := "01" ;
          when 2  =>   x := "10";
          when 3  =>   x := "11";
          when others => x := "00"
        end case;
   return x;
end adressage;
```

L'équation récurrente qui représente la sortie de la fonction est donnée par :

```
ChangeVar[adressage[t],
      Ife[equal[t, 1]
            ,{0, 1}
            ,Ife[equal[t, 2]
                  ,{1, 0}
                  ,Ife[equal[t, 3]
                         ,{1, 1}
                         ,{0, 0}
                        ]
                  ]
            ]
      ]
```

3.6 Conclusion

Dans ce chapitre, nous avons présenté les fondements de notre modélisation Mathématique de VHDL : les équations récurrentes et la modélisation par Ife. Le modèle de SER est adapté pour la simulation symbolique de haut niveau car il définit les équations sur des domaines plus abstraits tels que les entiers. Les propriétés de la fonction Ife ont permis la définition d'un algorithme d'extraction qui manipule symboliquement le code VHDL sans synthétiser les éléments abstraits. Le résultat de l'algorithme est un système d'équations récurrentes SER : une équation pour chaque objet du circuit. Ces équations sont au niveau de cycle de simulation VHDL. Donc, aucune restriction ou précision d'une horloge n'est imposée.

La prochaine étape est d'appliquer sur ces équations l'algorithme de simulation VHDL. C'est le sujet du chapitre suivant.

Chapitre 4

4 La Simulation symbolique

4.1 Le mécanisme de simulation VHDL

Comme nous l'avons dit précédemment, VHDL est défini pour la modélisation et la simulation des circuits digitaux. Le manuel de VHDL [IEEE93], ainsi que la norme pour la synthèse [IEEE04], décrivent plusieurs aspects de la sémantique de VHDL en terme de simulation.

La compréhension de notre approche de simulation symbolique suppose de connaître les principes d'exécution d'un modèle VHDL. La première étape est l'élaboration d'un modèle simulable. La deuxième étape est l'exécution dirigée par événements de ce modèle par un simulateur, selon l'algorithme de simulation de VHDL. Nous résumons ici les éléments majeurs du mécanisme de simulation VHDL qui sont extraits des références [ABO98], [Coh99] et [PKZ98].

4.1.1 Le simulateur VHDL

Un simulateur VHDL manipule essentiellement des processus. Ainsi, une description VHDL est vue comme un ensemble de processus.

- Un processus ne contient que des instructions séquentielles.
- Toute instruction concurrente est représentable par un processus combinatoire équivalent.
- L'exécution d'un processus est cyclique. Il est activé ou interrompu temporairement ou définitivement selon les signaux auxquels il est sensible (les signaux de sensibilité).

Le simulateur a accès à l'ensemble des signaux et coordonne l'activation ou non des processus de la description simulée. Lors de la simulation, il détecte les événements produits par les signaux de sensibilité. Quand tous ces signaux sont stabilisés, le temps de simulation est avancé jusqu'à la date du prochain événement et le simulateur met à jour les valeurs des objets de la conception (entrées, sorties, signaux et variables). Ensuite, le simulateur recommence ces opérations de manière cyclique. Il exécute ces opérations en deux phases : initialisation et exécution cyclique.

La phase d'initialisation : les variables qui représentent le temps de simulation courant et le temps de simulation futur sont notées T_n et T_c. La phase d'initialisation comporte les étapes suivantes :

1. La variable T_c est affectée par zéro.
2. Chaque objet de la conception est affecté par une valeur initiale. Elle est donnée explicitement dans le fichier VHDL ou implicitement par le standard VHDL [IEEE04].
3. Les processus qui ne sont pas précédés par le mot clé « `postponed` » sont exécutés.
4. Les processus précédés par le mot clé « `postponed` » sont exécutés.
5. La variable T_n est mise à jour selon l'étape 6 de la phase d'exécution cyclique.

La phase d'exécution cyclique : le temps de simulation maximal est T_{high}. Les étapes suivantes sont exécutées dans l'ordre :

1. Le temps de simulation T_c est mis à jour : $T_c := T_n$
2. Si $(T_c = T_{high})$ la simulation s'arrête.
3. Mise à jour des signaux et détections des événements.
4. Activation des processus : si un événement est détecté sur un signal de sa liste de sensibilité, le processus est actif.
5. Les processus actifs qui ne sont pas précédés par le mot clé « `postponed` » sont exécutés.
6. Mise à jour de la variable T_n, avec la date la plus proche parmi :
   - T_{high}
   - La prochaine date à laquelle un pilote de signal est actif.
   - La prochaine date à laquelle un processus est actif.

   Si $T_n = T_c$ un cycle delta est nécessaire et nous recommençons les opérations à partir de l'étape 1.
7. Les processus actifs précédés par le mot clé « `postponed` » sont exécutés.
8. La variable T_n est mise à jour selon l'étape 6. Si $T_n = T_c$ alors une erreur est détectée, sinon nous bouclons avec l'étape 1.

4.1.2 Le cycle delta

Le simulateur peut exécuter le modèle simulable sans avancer le temps de simulation courant t_c. Ce cycle de simulation particulier est appelé le cycle delta. Il est présent dans le cas où un signal de sensibilité est modifié au cours d'un cycle de simulation.

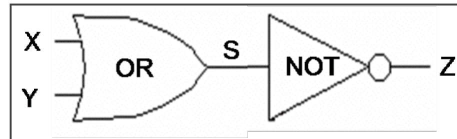


Figure 4.1 : circuit simple.

Par exemple, considérons le circuit simple de la figure 4.1 décrit par le VHDL suivant :

```
Entity exemple_delta is
  Port( X,Y : in bit ; Z : out bit);
End exemple_delta;
Architecture a of exemple_delta is
  Signal s : bit;
Begin
   Porte_ou : Process (X,Y)
   Begin
      S <= X or Y ;
   End Process ;
Porte_non : Process (S)
Begin
   Z <= not (S) ;
End Process ;
End a ;
```

Si pendant un cycle de simulation tc, un événement est détecté sur l'une des entrées X ou Y, le simulateur exécute le processus Porte_ou. Dans ce même cycle le simulateur n'exécutera pas le processus Porte_non car aucun événement n'est détecté sur le signal S. Cependant, le signal S est affecté avec un délai nul dans le processus Porte_ou. Cette affectation prend effet seulement lors de la mise à jour des signaux à la fin du cycle de simulation. Le simulateur teste ensuite si l'un des signaux de sensibilité est affecté (dans ce cas c'est le signal S). Il exécute alors un cycle de simulation sans avancer le temps de simulation t_c : un cycle delta. Pendant ce cycle, seul le processus Porte_non est exécuté car un événement est présent sur le signal S et aucun événement n'est détecté sur les signaux X et Y. Le signal de sortie Z est affecté par la valeur à

jour de S : Z <= Not(X or Y). Après l'exécution de ce cycle delta et la mise à jour des signaux, aucun nouvel événement n'est détecté sur les signaux de sensibilité, et le simulateur avance t_c.

Finalement, le simulateur a obtenu la valeur de Z après deux cycles de simulation, mais en gardant la même valeur de t_c. Ce mécanisme de cycle delta permet la propagation des événements entre les entrées et les sorties du système. Ceci est une particularité intéressante pour la synthèse des matériels à partir de VHDL, puisque le comportement du système numérique est décrit en supposant que les délais nécessaires pour la propagation physique des signaux sont nuls pendant une unité de temps de simulation.

4.2 La simulation symbolique avec un SER

Nous développons dans cette section l'algorithme de simulation VHDL avec les considérations suivantes :

- Les valeurs de tous les objets de la conception sont des expressions symboliques comme décrit dans le chapitre 2. Les valeurs numériques forment alors des cas particuliers de ces expressions symboliques.
- Une description VHDL est modélisée en SER, comme défini dans le chapitre 3.
- Les symboles qui représentent les noms des objets de la conception sont strictement différents des valeurs symboliques que nous utilisons pendant la simulation symbolique.

Pour décrire plus facilement l'algorithme de simulation, nous définissons les notions ci-dessous.

Définition 4.1 : un cycle de simulation

Comme en VHDL, un cycle de simulation est l'exécution du modèle SER une seule fois. Le cycle delta est un cycle de simulation.

Définition 4.2 : un cycle temporel

Nous appelons cycle temporel l'exécution répétée du modèle SER à une date fixée jusqu'à la stabilisation du modèle à cette date. Un cycle temporel est l'exécution d'un ou plusieurs cycles de simulation.

4.2.1 Exécution d'un cycle de simulation

La figure 4.2 montre une vue d'un cycle de simulation symbolique. L'exécution du modèle consiste à calculer les expressions symboliques dans les NextSig et ChangeVar en utilisant les règles de transformation statiques prédéfinies dans le simulateur, ou dynamiques créées lors de la mise à jour des S_i, et les règles de simplifications liées à la normalisation Ife. Les résultats sont les expressions de chaque objet S_i de la conception après le cycle de simulation n.

Les règles de simplification standard de Mathematica : ces règles sont celles abordées dans le chapitre 2. Elles sont définies pour simplifier les opérations arithmétiques de base sur $K^n[x]$. L'utilisateur peut désactiver une ou plusieurs règles, mais modifier une règle standard peut affecter le simulateur symbolique, et nous le déconseillons. Nous notons ces règles Reg_{Math}.

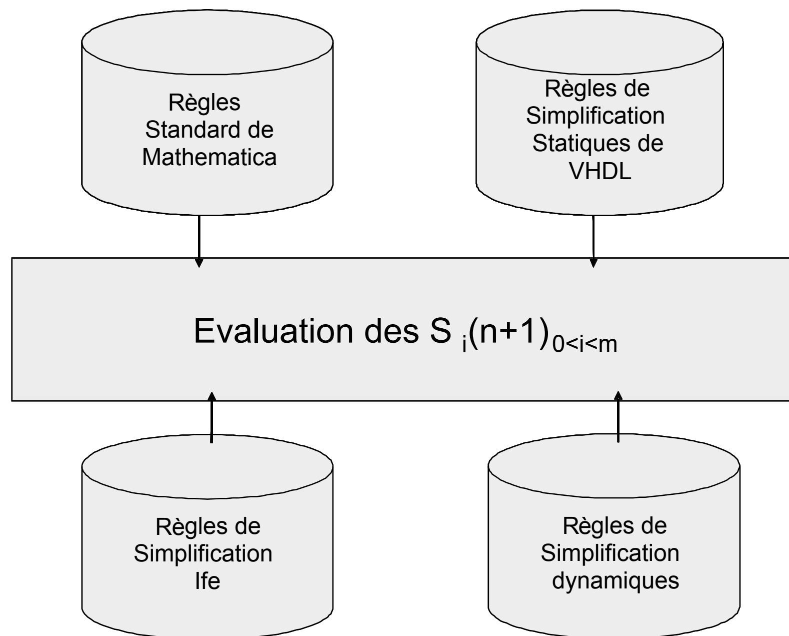


Figure 4.2 : vue d'un cycle de simulation symbolique avec SER.

Les règles de simplification statique de VHDL : les bits et les types booléens ne sont pas définis dans Mathematica. Nous avons avancé dans le chapitre 2 qu'un bit est représenté par une expression du domaine des entiers qui sont compris entre 0 et 1. Pour simplifier ces expressions, nous avons défini un ensemble de règles de simplification. Il ne s'agit pas de normalisation en utilisant ces règles, mais d'une phase de simplification pour réduire les

expressions simples avant toute normalisation possible. Nous notons ces règles Reg_{Stat}. Les exemples suivants montrent un extrait de ces règles :

```
and[a_,1]:=a;
and[1,a_]:=a;
and[a_,a_]:=a;
and[0,___]:=0;
not[not[x_]]:=x;
or[_,1]:=1;
or[1,_]:=1;
or[a_,0]:=a;
or[0,a_]:=a;
xor[0,a___]:=a;
xor[a___,0]:=a;
xor[a_,a_]:=0;
```

Les règles de la normalisation par Ife : l'utilisation de la fonction Ife nous permet de profiter des théorèmes et définitions présentés dans le chapitre 3 : théorème de distribution, théorème de réduction et théorème de IF-Y-Y. Nous notons ces règles Reg_{Ife}.

Les règles de simplification dynamique : les valeurs des S_i calculées dans des cycles temporels précédents sont considérées comme des règles qui entrent dans le calcul des valeurs futures des S_i. Nous notons ces règles $Reg_{n,dyn}$ avec :

$Reg_{n,dyn}=\{ S_i(n) \text{ -> } expr_i(n), S_i(n-1) \text{ -> } expr_i(n-1) \}_{0<i\leq m}$.

Définition 4.4 : l'exécution symbolique d'un SER

Considérons un composant VHDL C décrit par un SER : $C =<I, K_I ,S ,K_s ,\{S_i\}_{0<i \leq m} >$.

Soit Reg_n l'ensemble des règles de transformation dans un cycle de simulation n défini par :

$Reg_n = Reg_{Math} \cup Reg_{Stat} \cup Reg_{Ife} \cup Reg_{n,dyn}$

L'exécution symbolique du C est :

$Execute\ (\{S_i\}_{0<i \leq m}) = ReplaceRepeated(\{S_i\}_{0<i \leq m}, Reg_n)$;

La convergence de l'exécution symbolique du modèle dépend des règles utilisées. La convergence de l'évaluation d'une expression avec les règles Reg_{Ife} est montrée dans [PZ00]. Les règles

définies dans Reg_{Stat} transforment une expression en une constante ou en une expression plus simple. Le but de Reg_{Stat} est de supprimer des expressions évidentes telles que *And[a,a]*. La preuve de convergence de l'évaluation avec Reg_{Stat} est immédiate. Les règles dynamiques Reg_n définissent une substitution simple. La partie droite de chaque règle ne contient aucun élément d'une partie gauche. Aussi, aucune interdépendance n'existe entre ces règles car à ce niveau d'exécution, la valeur future d'un objet est calculée en fonction des valeurs anciennes sans mise à jour des signaux.

4.2.2 L'algorithme de simulation

Définition 4.3 : un scénario de test

Considérons un système C décrit en VHDL et représenté par un SER avec :

$C = \langle I, K_I, S, K_s, \{S_i\}_{0<i\leq m} \rangle$, où I est constitué de m1 éléments.

Soit U_{m1} un ensemble de m1 vecteurs. Chaque vecteur contient *n* expressions symboliques. Un scénario de test symbolique Δ sur *n* cycles temporels est la spécification des valeurs dans I par rapport au temps de simulation :

$\Delta = \{ I_1=\{I_1(1) \text{->} U_{1,1}, I_1(2) \text{->} U_{1,2},\ldots, I_1(n) \text{->} U_{1,n}\},$

$I_2=\{I_2(1) \text{->} U_{2,1}, I_2(2) \text{->} U_{2,2},\ldots, I_2(n) \text{->} U_{2,n}\},$

...

$I_{m1}=\{I_{m1}(1) \text{->} U_{m1,1}, I_{m1}(2) \text{->} U_{m1,2},\ldots, I_{m1}(n) \text{->} U_{m1,n}\}\}$

L'algorithme de simulation est une boucle dans laquelle nous calculons les expressions symboliques de $\{S_i\}_{0<i\leq m}$ au cours du temps en appliquant un scénario. Plus précisément :

```
For tc = 1 to tn do
  Do
     Get(Δ,tc)
      Execute({Si}0<i≤m)
  While Event(Ssync(tc));
End For;
```

La détection des événements est faite à l'aide de la fonction `Event` définie dans le chapitre précédent. L'ensemble $S_{sync}(t_c)$ contient les valeurs des signaux de sensibilité dans le

composant simulé. Ces valeurs sont extraites automatiquement à partir des processus et des autres instructions concurrentes.

Les résultats de la simulation symbolique sont en fonction du scénario de test symbolique Δ sur t_n cycles temporels. Ils forment l'ensemble :

$R(\Delta, t_n) = \{ S_1(t_n), S_2(t_n), \ldots, S_m(t_n) \}$.

Nous pouvons dire que le résultat de la simulation symbolique d'un objet S_i est l'équation récurrente suivante :

$S_i(t_n) = f_i(I_1(t_n -1), I_2(t_n -1), \ldots, I_{m1}(t_n -1), S_1(t_n -1), S_2(t_n -1), \ldots, S_m(t_n -1))$, où fi est la fonction de transition après t_n cycles.

Définition 4.4 : la stabilisation d'un modèle

Nous disons qu'un modèle SER simulé symboliquement se stabilise, si l'algorithme de simulation symbolique défini précédemment termine.

La sémantique VHDL interdit l'écriture sur les entrées et la lecture des sorties par le composant lui-même. Donc, si le modèle ne contient que des ports d'entrées/sorties, il se stabilise. Mais dans le cas général un système contient des signaux internes. L'ensemble des affectations concurrentes peut contenir un cycle de dépendance entre les signaux affectés. Dans ce cas, le simulateur exécute un nombre non borné de cycles delta et le modèle ne sera pas stabilisé. Cette limitation est liée à VHDL, et habituellement l'utilisateur doit vérifier l'absence de ce genre de problème avant de simuler le modèle.

Notre algorithme de simulation est fidèle à la sémantique VHDL. Par conséquent, nous ne pouvons pas prouver la convergence du système de réécriture associé à cet algorithme. En réalité, mathématiquement, cet algorithme ne termine pas nécessairement.

4.2.3 Le simulateur symbolique : TheoSim

Nous avons implémenté notre approche de simulation symbolique dans Mathematica. Le prototype appelé TheoSim est l'implémentation de l'algorithme de simulation symbolique présenté dans la section précédente. TheoSim fournit aussi un environnement de simulation symbolique pour VHDL. Cet environnement est un ensemble de fonctions et de variables globales destinées à contrôler le simulateur symbolique pour atteindre deux buts :

- Permettre aux utilisateurs de choisir les différentes configurations de simulation symbolique.
- Donner la possibilité de définir des stratégies de tests à l'aide de symboles.

Les symboles utilisés pendant la simulation peuvent être indexés en utilisant le temps de simulation ou l'horloge du système. Le tableau 4.1 explicite une partie de ces variables et fonctions que nous avons définies, en donnant une explication sur leurs rôles.

Exemple :
La fonction suivante définit un vecteur de test pour le signal **sig** à l'aide de l'environnement de simulation symbolique :

```
NextSig[sig$1, Ife[Event[Clock[$SimulationTime,2]],
,Affect[{1,0},{2,1},{5,INPUT[$SimulationTime]}],sig$0]]
```

Le signal **sig** est affecté, si un événement est détecté sur l'horloge Clock, par 0 au premier cycle de simulation (l'instant 1), par 1 à l'instant 2 et par une valeur symbolique indexée par le temps de simulation à partir de l'instant 4. Aux instants 3 et 4 la valeur est mémorisée (donc égale à 1).

Évidemment, toute fonction native Mathematica peut être utilisée pendant la simulation. Cela est pratique pour générer des valeurs de test. Par exemple la fonction Mathematica **Sin[x]** peut être utilisée pour tester un circuit de traitement du signal.

Dans notre approche de simulation symbolique, les objets de la conception ne sont pas initialisés par les valeurs par défaut fournies par le standard VHDL. L'utilisateur doit fournir explicitement la valeur initiale de chaque objet de la conception. Le but est de pouvoir attribuer une valeur symbolique à la place de la valeur numérique par défaut. Les fonctions NextSig et ChangeVar sont utilisées pour cette affectation.

Par exemple, pour initialiser l'objet sig avec la valeur symbolique SIG nous écrivons :

```
NextSig[sig, SIG]] ;
```

Tableau 4.1 : les fonctions et les variables destinées à contrôler le simulateur symbolique	
`$SimulationTime`	La variable globale qui représente le temps de simulation.
`$SimulationTimeMax`	La variable globale qui est fixée par l'utilisateur pour donner le nombre de cycles de simulation désirés.
`NoDelta`	La fonction qui teste la stabilisation des signaux de synchronisation du système pendant un cycle de simulation.
`$DeltaMax`	La variable globale qui fixe la limite supérieure des itérations delta nécessaires pour le calcul de point fixe de stabilisation de synchronisation. La valeur *Infinity* est acceptable.
CLOCK[t,p]	La fonction qui donne la valeur d'une horloge de demi-période p à l'instant t.
Affecte[{t_1,v_1},{t_2,v_2},…,{t_n,v_n}]	La fonction qui donne la valeur v_1 à l'instant t_1, v_2 à l'instant t_2,…et v_n à l'instant t_n, avec la condition : $t_1<t_2<\ldots<t_n$. La valeur donnée entre t_{n-1} et t_n est la valeur v_{n-1}. Cette fonction est pratique pour définir un scénario d'exécution. Pour affecter un objet `Obj` avec cette fonction, nous écrivons : NextSig[obj,Affecte[{t_1,v_1},{t_2,v_2},…,{t_n,v_n}]]

Dans cet objectif, TheoSim intègre un outil de génération de test. Il génère une maquette de test que l'utilisateur est capable de modifier, puisque l'outil affecte par défaut chaque objet par un symbole qui est le nom de l'objet écrit en lettres majuscules.

4.3 La méthodologie de test symbolique

Le but de cette méthodologie est d'utiliser efficacement la simulation symbolique. L'utilisateur peut alors simuler une partie du système en définissant des vecteurs de test qui portent des valeurs numériques ou symboliques. Ainsi, l'utilisateur peut donner des valeurs numériques ou un scénario d'exécution pour la partie contrôle, et observer les symboles propagés dans la partie opérative.

L'idée de partitionner le système en une partie contrôle et une partie opérative n'est pas nouvelle et nous trouvons plusieurs définitions dans la littérature VHDL et dans le domaine de la synthèse de haut niveau [Dus99]. Dans cette section, nous donnons d'autres définitions pour la partie contrôle et la partie opérative. Le but est d'exprimer de manière précise notre méthodologie de test symbolique. Donc ces définitions ne sont pas nécessairement généralisables dans un autre contexte.

4.3.1 Partie contrôle et partie opérative du système

Nous partons de l'idée que toutes les instructions de contrôles en VHDL (case, For-loop, l'affectation conditionnelle de signal et l'affectation sélective de signal) sont normalisées en *Ife* comme expliqué dans chapitre précédent.

Prenons d'abord le cas d'un composant VHDL dans le niveau hiérarchique le plus bas ; son architecture ne contient pas une instanciation d'un composant.

Définition 4.4 : un élément local de décision
Un élément local de décision est un objet qui apparaît dans la condition d'une instruction de contrôle. Donc, pour une fonction *Ife(cond, x, y),* les éléments locaux de décision sont les objets qui constituent *cond*.

Définition 4.5 : un élément local de donnée
Un élément local de donnée est un objet qui n'apparaît dans aucune condition des instructions de contrôle de son architecture.

Définition 4.6 : la partie contrôle d'un composant
La partie contrôle d'un composant VHDL est un ensemble qui contient tous les éléments locaux de décision dans son architecture.

Définition 4.7 : la partie opérative d'un composant
La partie opérative d'un composant VHDL C est un ensemble qui contient tous les éléments locaux de données dans son architecture. Soit C décrit par le SER, $C=<I, K_I, S, K_s, \{S_i\}_{0<i \leq m}>$, P_{cont} est la partie contrôle et P_{opr} est la partie opérative du C. Elles satisfont les relations suivantes :

- $I \cup S = P_{opr} \cup P_{cont}$
- $P_{opr} \cap P_{cont} = \phi$

Considérons le cas général d'un composant défini structurellement par l'interconnexion de plusieurs composants. Nous parlons alors d'un système.

Définition 4.8 : la partie contrôle d'un système
La partie contrôle d'un système est l'union de toutes les parties contrôles des composants qui le constituent. Nous la notons P_c.

Définition 4.9 : la partie opérative d'un système
La partie opérative d'un système est l'intersection de toutes les parties opératives des composants qui le constituent. Nous la notons P_o.

4.3.2 Les modes de vecteurs de test symboliques

Avec le partitionnement du système en partie opérative et en partie contrôle, nous pouvons distinguer trois modes de vecteurs de test symboliques. Dans tous les modes, l'horloge mère du système est affectée par des valeurs numériques.

1. ***Le mode raisonnement*** : les deux parties opérative et contrôle sont affectées par des valeurs symboliques. Le but est d'extraire les équations récurrentes du système après n cycles de simulations.

2. ***Le mode exécution*** : seule la partie opérative du système est affectée par des valeurs symboliques.

3. ***Le mode mixte*** : au moins un élément local de décision de la partie contrôle est affecté par des valeurs numériques pendant la simulation.

Le choix du mode de simulation dépend du circuit à simuler et des propriétés que nous souhaitons vérifier.

4.3.3 Illustration sur un exemple

Pour illustrer les différents concepts développés dans cette section, nous utilisons l'exemple d'un registre à décalage décrit de la façon suivante :

```
entity regdec is
     port( int_in     : in std_ulogic_vector(7 downto 0) ;
           ad_rd      : in std_ulogic_vector(1 downto 0) ;
           l_e_rd     : in std_ulogic ;
           reset      : in std_ulogic ;
           clk        : in std_ulogic ;
           int_rd     : out std_ulogic_vector(7 downto 0)) ;
end regdec;

architecture a of regdec is
      signal x0, x1, x2, x3 : std_ulogic_vector(7 downto 0);
begin
  process (clk,reset)
  begin
    if clk'event and clk='1' then
     if reset='1' then
        x0 <="00000000";
        x1 <="00000000";
        x2 <="00000000";
        x3 <="00000000";
     end if;
     if reset='0' and l_e_rd='0' then
        x3     <= x2;
        x2     <= x1;
        x1     <= x0;
        x0     <= int_in;
     end if;
     case ad_rd is
        when "00" => int_rd <= x0;
        when "01" => int_rd <= x1;
        when "10" => int_rd <= x2;
```

```
        when "11" => int_rd <= x3;
      end case;
     end if;
  end process;
end a;
```

La partie contrôle du registre à décalage précédent est {clk, reset, l_e_rd , ad_rd} , et la partie opérative est {x0, x1, x2, x3, int_in, int_rd}. Nous simulons ce circuit en utilisant les trois modes de vecteurs de test symboliques.

Mode raisonnement

Tout d'abord, nous affectons l'horloge (le signal clk) en utilisant la fonction *CLOCK* qui génère une horloge classique : {clk (0) -> 0, clk (1) -> 1, clk (2) -> 0, clk (3) -> 1,..., clk (2 n) -> 0, clk (2 n +1) -> 1 } où la période est égale à 2 unités temporelles.

Ensuite, nous initialisons les deux parties contrôle et opérative par des valeurs symboliques identiques à leurs noms mais en utilisant des lettres majuscules. Ces symboles représentent les valeurs des S_i à un cycle temporel (n-1). L'initialisation d'un objet VHDL est écrite de la manière suivante:

```
NextSig[ad$rd, AD$RD];
```

Pour exprimer la valeur actuelle du signal ad_rd, nous utilisons le symbole AD$RD : le caractère « _ », ayant une signification particulière en Mathematica, est remplacé par « $ ». La figure 4.3 montre un vecteur de test symbolique en mode raisonnement.

Les résultats de simulation après une période d'horloge sont les fonctions de transfert de chaque élément du système au niveau du cycle d'horloge, et après la stabilisation du modèle. Le tableau 4.2 explicite les résultats de simulation symbolique après un cycle d'horloge en utilisant la syntaxe de Mathematica. L'expression symbolique du signal x0 est équivalente à l'équation mathématique suivante :

$X_0(n+2) =$

Ife(

and (equal(rest(n),0),equal(l_e_rd(n),0)) ,

int_in(n),

Ife(

equal(reset(n),1),

{0, 0, 0, 0, 0, 0, 0, 0},

$X_0(n)$

)

)

```
NextSig[clk,CLOCK[$SimulationTime,1]];
NextSig[ad$rd, AD$RD];
NextSig[int$in, INT$IN];
NextSig[int$rd, INT$RD];
NextSig[l$e$rd, L$E$RD];
NextSig[reset, RESET];
NextSig[x0, X0];
NextSig[x1, X1];
NextSig[x2, X2];
NextSig[x3, X3];
```

Figure 4.3 : vecteur de test symbolique en mode raisonnement.

Pour obtenir l'expression de `x0` donnée par le tableau 4.2, il faut remplacer chaque variable dans l'équation par son équivalent dans l'implémentation. Dans ce but, la liste de règles de substitution suivante peut être employée : `{`reset(n) `-> RESET,` l_e_rd (n) `-> L$E$RD,` int_rd(n) `-> INT$RD,` int_in(n) `-> INT$IN,` $X_0(n)$ `-> X0}`. Un cycle d'horloge est équivalent à deux cycles temporels.

TABLEAU 4.2 : RÉSULTATS DE LA SIMULATION SYMBOLIQUE EN MODE RAISONNEMENT.

L'objet	La valeur calculée après un cycle d'horloge
x0	`if[and[RESET == 0, L$E$RD == 0], INT$IN, if[RESET == 1, {0, 0, 0, 0, 0, 0, 0, 0}, X0]]`
x1	`If[and[RESET == 0, L$E$RD == 0], X0, if[RESET == 1, {0, 0, 0, 0, 0, 0, 0, 0}, X1]]`
x2	`if[and[RESET == 0, L$E$RD == 0], X1, if[RESET == 1, {0, 0, 0, 0, 0, 0, 0, 0}, X2]]`
x3	`if[and[RESET == 0, L$E$RD == 0], X2, if[RESET == 1, {0, 0, 0, 0, 0, 0, 0, 0}, X3]]`
int_rd	`if[AD$RD == {0, 0}, X0, if[AD$RD == {0, 1}, X1, if[AD$RD == {1, 0}, X2, if[AD$RD == {1, 1}, X3, INT$RD]]]]`

Mode exécution

Nous utilisons le même modèle d'horloge que dans le mode raisonnement. La partie opérative est affectée par des valeurs symboliques. Par contre, les valeurs de la partie contrôle sont définies comme pour une simulation numérique classique. Les résultats obtenus forment alors un cas particulier et ne sont plus les équations qui décrivent le système après n cycles temporels, mais plutôt les équations qui décrivent le système dans une configuration particulière après n cycles de simulation. La figure 4.4 montre un vecteur de test possible. Les résultats de la simulation de ce vecteur (pendant 9 cycles temporels) sont présentés dans le tableau 4.3.

```
NextSig[clk,CLOCK[$SimulationTime,1]];
NextSig[ad$rd, {0,0}];
NextSig[int$in,INT$IN[$SimulationTime]];
NextSig[l$e$rd,0];
NextSig[reset,0];
```

Figure4.4 : un vecteur de test en mode exécution.

TABLEAU 4.3 : RÉSULTATS DE LA SIMULATION SYMBOLIQUE EN MODE EXÉCUTION.

Cycle temporel *i*	x0 (i)	x1(i)	x2(i)	x3(i)	int_rd(i)
1	X0	X1	X2	X3	INT$RD
3	INT$IN[1]	X0	X1	X2	X0
5	INT$IN[3]	INT$IN[1]	X0	X1	INT$IN[1]
7	INT$IN[5]	INT$IN[3]	INT$IN[1]	X0	INT$IN[3]
9	INT$IN[7]	INT$IN[5]	INT$IN[3]	INT$IN[1]	INT$IN[5]

Mode mixte

Il est intermédiaire entre les deux modes précédents. Le but est d'exécuter numériquement un scénario de test, pendant *p* cycles temporels, pour arriver à un mode opératoire particulier du circuit. Ensuite, nous affectons des valeurs symboliques pour certains objets du circuit et nous observons le comportement pendant n cycles temporels. La simulation en mode mixte a duré alors *(p+n)* cycles temporels au total. Cette application est inspirée des travaux de Dumitrescu [Dum03] sur l'utilisation de la simulation symbolique comme stratégie de simplification pour la vérification.

Pour appliquer cette méthode à notre exemple, nous pouvons considérer un scénario où le reset est à '0' pendant la simulation, et nous observons les résultats sur le vecteur de test de la figure 4.5. Les résultats de la simulation sont présentés dans le tableau 4.4.

TABLEAU 4.4 : RÉSULTATS DE LA SIMULATION EN MODE MIXTE APRÈS 2 CYCLES TEMPORELS.

L'objet	La valeur calculée de l'objet
x0	`if[L$E$RD$ == 0, INT$IN[1], X0]`
x1	`if[L$E$RD$ == 0, X0, X1]`
x2	`if[L$E$RD$ == 0, X1, X2]`
x3	`if[L$E$RD$ == 0, X2, X3]`
Int_rd	`if[AD$RD$[1] == {0, 0}, X0, if[AD$RD$[1] == {0, 1}, X1, if[AD$RD$[1] == {1, 0}, X2, if[AD$RD$[1] == {1, 1}, X3, INT$RD]]]]`

RÉSULTATS DE LA SIMULATION EN MODE MIXTE APRÈS 4 CYCLES TEMPORELS.

L'objet	La valeur calculée de l'objet
x0	`if[L$E$RD$ == 0, INT$IN[3], X0]`
x1	`if[L$E$RD$ == 0, INT$IN[1], X1]`
x2	`if[L$E$RD$ == 0, X0, X2]`
x3	`if[L$E$RD$ == 0, X1, X3]`
Int_rd	`if[ADRD[3] == {0, 0}, if[LERD$ == 0, INT$IN[1], X0], if[AD$RD$[3] == {0, 1}, if[L$ERD == 0, X0, X1], if[ADRD[3] == {1, 0}, if[LERD$ == 0, X1, X2], if[ADRD[3] == {1, 1}, if[LERD$ == 0, X2, X3], if[ADRD[1] == {0, 0}, X0, if[ADRD[1] == {0, 1}, X1, if[ADRD[1] == {1, 0}, X2, if[ADRD[1] == {1, 1}, X3, INT$RD]]]]]]]]`

Nous remarquons l'absence du `reset` et les valeurs initiales données. En l'absence des valeurs initiales, la taille des registres en termes de bits est inexistante. Donc la vérification sur ces équations est indépendante de la taille.

```
NextSig[ad$rd,AD$RD$[$SimulationTime]];
NextSig[clk,CLOCK[$SimulationTime]];
NextSig[int$in,INT$IN[$SimulationTime]];
NextSig[l$e$rd,L$E$RD];
NextSig[reset,0];
```

Figure 4.5 : un vecteur de test symbolique en mode mixte

4.3.4 Utilité de la méthodologie de simulation symbolique

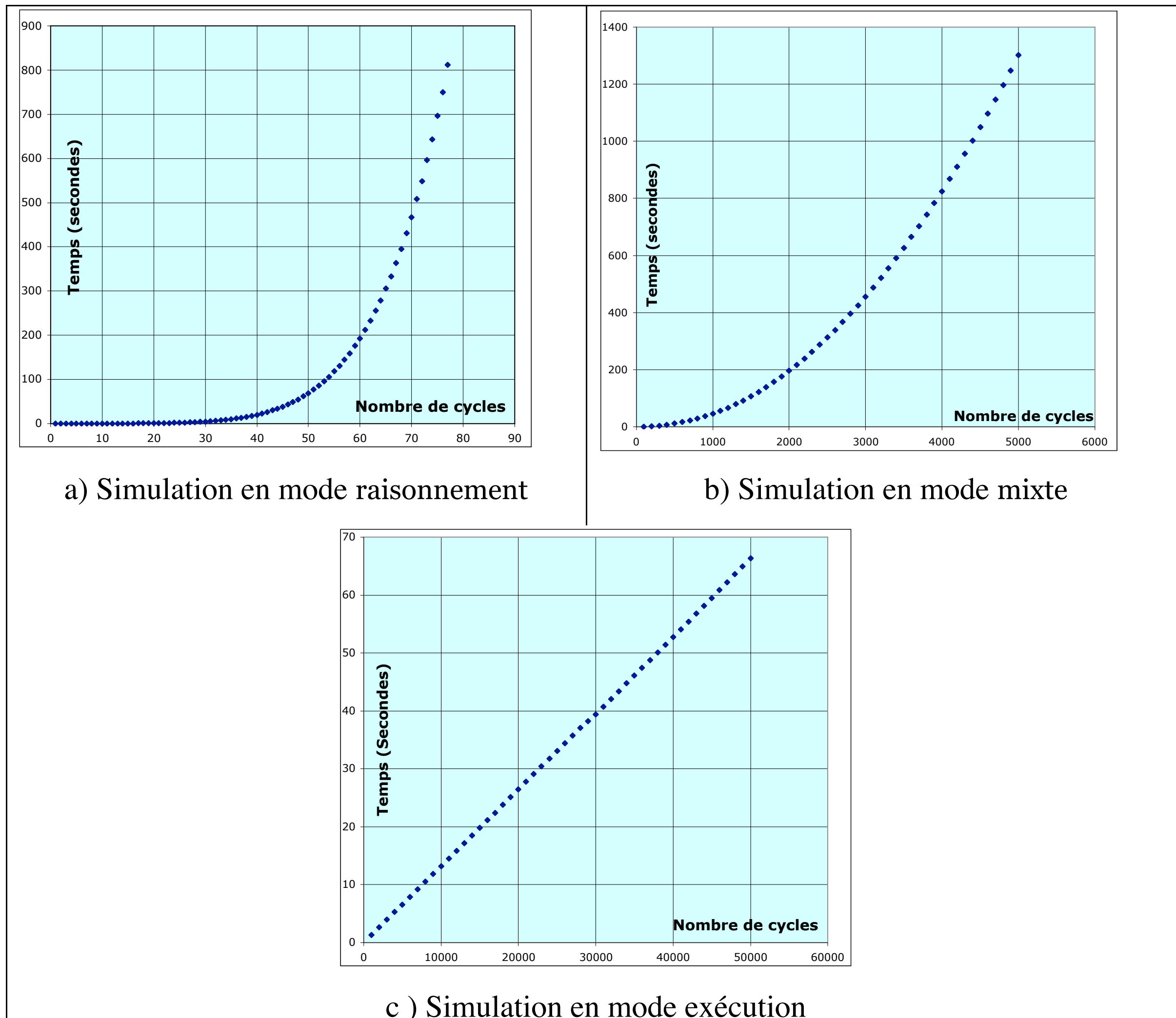
La simulation par vecteurs de test est une technique bien comprise des ingénieurs de conception matérielle. Écrire des vecteurs de test en utilisant la syntaxe de Mathematica et les éléments définis dans notre environnement TheoSim requiert peu de formation si l'on compare aux techniques de vérification formelle, malgré les notions fonctionnelles qui en font leur base. La méthodologie de simulation avec des vecteurs de test symboliques est un compromis entre la complexité des méthodes formelles qui tend à définir le système en termes de logiques, et la simplicité de la simulation numérique.

Le mode raisonnement est le plus général, avec lequel les équations résultantes expriment le comportement du système sans aucune restriction. Donc, la vérification avec ce mode est une vérification formelle. Cependant, la taille des expressions croît exponentiellement avec le temps de simulation (même pour un petit circuit). Prenons l'exemple de registre à décalage précédant, la figure 4.6.a montre le temps de simulation de l'exemple en mode raisonnement.

Le mode mixte est proposé pour contourner ce problème. Le concepteur écrit les vecteurs de test symboliques pour observer un comportement particulier. Dans le cas de l'exemple précédant, le mode mixte est écrit pour tester que le registre fonctionne correctement en absence de `reset` et en gardant un seul mode opératoire. Les équations expriment un cas particulier tout en restant beaucoup plus générales par rapport à la simulation numérique. La taille des expressions symboliques de certains objets croît exponentiellement avec le temps de simulation, mais la

pente est considérablement plus douce, presque polynomiale, si l'on compare avec le mode raisonnement. Pour cet exemple, nous gagnerons 60 fois plus de performance.

La figure 4.6.b montre l'évaluation du temps de simulation avec le nombre de cycles. Certes, la vérification est semi-formelle mais symbolique et fournit un niveau de vérification acceptable, notamment pour valider des propriétés mathématiques. Le mode exécution est proposé pour vérifier le cas d'un circuit qui contient une partie donnée (définition 4.8) de taille importante et une partie contrôle préprogrammée à l'avance. Un cas typique est le circuit du traitement de signal. Dans ce cas, La simulation symbolique est presque linéaire (voir figure 4.6.c pour le registre à décalage). Nous précisons que les résultats de la figure 4.6 sont obtenus en utilisant une machine SUN-Blade 100 Sparc avec 640 MO de mémoire vive.



a) Simulation en mode raisonnement

b) Simulation en mode mixte

c) Simulation en mode exécution

Figure 4.6 : évaluation de la performance de simulation symbolique de l'exemple.

4.4 La réduction du modèle

L'algorithme de simulation symbolique répète l'exécution du modèle SER pendant plusieurs cycles de simulation, pour stabiliser le modèle ou pour balayer l'ensemble des vecteurs de test symboliques. L'idée de la réduction du modèle est montrée dans la figure 4.7. Nous partons d'un SER et d'une stratégie de réduction pour obtenir un autre SER dit réduit. Ensuite, nous simulons symboliquement le modèle réduit, ce qui implique un gain de performance. Cette réduction est utile dans certaines situations comme celle où une seule exécution symbolique est suffisante pour stabiliser le modèle statiquement. Dans cette section, nous proposons deux techniques de réduction : par scénario et par abstraction.

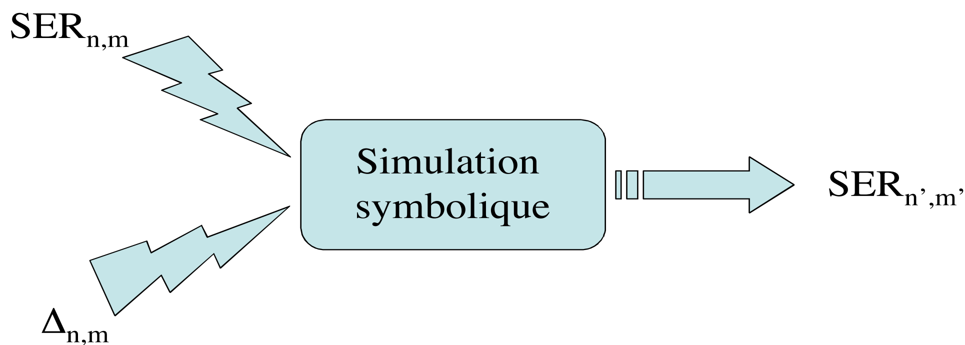


Figure 4.7 : réduction du modèle par scénario.

4.4.1 La réduction par scénario

Nous partons d'un SER et d'un scénario de simulation symbolique pour obtenir le SER réduit. Formellement, considérons un composant VHDL décrit par un SER :

$C=\langle I, K_I, S, K_s, \{S_i\}_{0<i\leq m} \rangle$

Les équations sont écrites en fonction du cycle temporel n. Considérons également un scénario de test Δ décrit aussi au niveau du cycle temporel n sur n_o cycles:

$\Delta = \{ I_1=\{I_1(1) \text{->} U_{1,1}, I_1(2) \text{->} U_{1,2},..., I_1(n) \text{->} U_{1,no}\},$

$I_2=\{I_2(1) \text{->} U_{2,1}, I_2(2) \text{->} U_{2,2},..., I_2(n) \text{->} U_{2,no}\},$

...

$I_{m1}=\{I_{m1}(1) \text{->} U_{m1,1}, I_{m1}(2) \text{->} U_{m1,2},..., I_{m1}(n) \text{->} U_{m1,no}\}\}$

Le scénario est défini en mode mixte ou en mode raisonnement dans le but de réduire la complexité des résultats symboliques. Après n_o cycles de simulation, les résultats obtenus forment un nouveau SER réduit :

$C' = <I', K_I, S', K_s, \{S'_i\}_{0<i\leq m'}>$, avec : $m' \leq m$ et $I' \subset I$,

$\{S'_i\}_{0<i\leq m'} = \{ S'_i(n+n_0), 0<i\leq m', S'_i(n+n_0)= f_i(I_1(n), I_2(n),..., I_{m1}(n), S_1(n), S_2(n),..., S_m(n))\}$

Application : extraction des fonctions de transition de la machine d'états finis

Nous nous plaçons dans le cas d'un circuit synchronisé par une horloge. Nous définissons un scénario de simulation en mode raisonnement ; seule l'horloge est affectée par des valeurs numériques. Ensuite, nous simulons pendant une période d'horloge en détectant tous les signaux de sensibilité. Toute présence d'événement (appel de la fonction Event) est remplacée par une valeur vraie ou fausse. Le résultat symbolique est la fonction de transition de chaque objet du circuit selon le modèle de la machine d'états finis. La transition est une période d'horloge, et un état du système est l'ensemble des valeurs de ces objets après une transition.

Les équations récurrentes obtenues sont au 1er ordre car toute occurrence de la fonction `Event` est remplacée par une valeur (vraie ou fausse). Donc, ces signaux peuvent être supprimés du modèle.

Pour illustrer, nous reprenons l'exemple de compteur décrit dans le chapitre 3 (le code dans la figure 3.2). Nous le simulons dans le mode raisonnement où la période d'horloge est de 2 cycles temporels. Le SER réduit est alors :

$I' = \{clear, count\}$, $K_I = \{B, B, B^{len}\}$, $S = \{Pre_Q, Q\}$, $K_S = \{B^{len}, B^{len}\}$.

$\{S'_i\}_{0<i\leq m'} =$

```
{NextSig(Pre_Q(n+1),
       Ife(clear(n) = 1,
             Pre_Q(n) - Pre_Q(n),
            Ife(count(n) = 1
                  ,Pre_Q(n)+ 1
                  ,Pre_Q(n)
               )
```

```
          )
        )
    , NextSig( Q(n+1), Ife(clear(n) = 1,
           Pre_Q(n) - Pre_Q(n),
          Ife(count(n) = 1
                ,Pre_Q(n)+ 1
                ,Pre_Q(n)
             )
          )
        )
)
}
```

Le *n* dans les équations précédentes représente un cycle d'horloge et non pas un cycle temporel. Nous remarquons l'absence des signaux de sensibilité dans le système réduit. Nous remarquons aussi que les signaux Pre_Q(n+1) et Q(n+1) sont les mêmes. Donc, dans ce cas nous, pouvons supprimer l'équation du signal Pre_Q du modèle.

D. Toma [TBA04] a utilisé notre simulateur symbolique et la technique de réduction par scénario pour extraire automatiquement les fonctions de transitions de la machine d'états finis. Elle a utilisé ensuite ces fonctions pour définir un modèle adapté pour le démonstrateur de théorèmes inductif ACL2 [KMM00a]. Puis ce modèle a servi à prouver des propriétés mathématiques sur le SHA-1 (un circuit cryptographique) [TBP04].

K. Morin-Allory a utilisé le simulateur de la même manière que D. Toma mais pour générer un modèle pour le démonstrateur de théorèmes PVS [OSR92]. Le but de ses travaux en cours est de prouver qu'une bibliothèque de composants est sémantiquement correcte et que la méthode de connexion de ces composants est elle aussi correcte. La bibliothèque implémente les opérateurs PSL [Acc04a] en VHDL synthétisable [BLO05].

4.4.2 La réduction par abstraction

La réduction par abstraction d'un SER est une intervention humaine pour simplifier les expressions symboliques. Considérons un SER $C=<I, K_I ,S ,K_S ,\{S_i\}_{0<i \leq m} >$, et un ensemble de règles de simplification *R* fourni par l'utilisateur. Le modèle SER réduit est :

$C' = \langle I', K_I, S', K_s, \{S'_i\}_{0<i\leq m'} \rangle$, avec :

$m' \leq m$

$I' \subset I$,

$\{S'_i\}_{0<i\leq m'} \subset \{S_i\}_{0<i\leq m}$ en conservant la relation suivante :

$\forall\, i,j \in N$, avec $i \leq m'$, si $i=j$ alors ReplaceList[S'_i[n+1], R] = S_j[n+1]]

L'abstraction ne doit pas influencer les valeurs calculées au cours du temps, mais seulement les compacter. De manière pratique, cela revient à introduire des variables supplémentaires pour abstraire certaines expressions symboliques et par conséquent diminuer leurs tailles.

Prenons l'exemple de la mémoire ROM suivante :

```
entity ent is
  port (
    clk    : in  bit;
    rst    : in  bit;
    sel    : in  integer;
    int    : in  integer;
    output : out integer);
end ent;


architecture a of ent is
  signal memory  : integer;
begin  -- a

  p_memory: process (clk, rst)
  begin
    if rst = '0' then
          memory <= 0;
    elsif clk'event and clk = '1' then
      case sel is
        when 0  => memory <= 1 ;
        when 1  =>  memory <= 10 ;
        when 2  =>  memory <= 100 ;
        when 3  =>  memory <= 1000 ;
        when others =>  memory <= 0;
      end case;
   end if;
  end process p_memory;

 output <= int * memory;

end a;
```

Les équations récurrentes après la simulation symbolique en mode raisonnement et 2 cycles temporels (une période d'horloge) sont :

```
{NextSig(memory(n+2),
            if(rst(n) = 0,
              0,
             if(sel(n) = 0,
                1,
                if(sel(n) = 1,
                  10,
                  if(sel(n) = 2,
                    100,
                    if(sel(n) = 3,
                      1000,
                      0)
                  )
                )
              )
      ) ,
NextSig(output(n+2),
            int(n) ×
            if(rst(n) = 0,
              0,
             if(sel(n) = 0,
                1,
                if(sel(n) = 1,
                  10,
                  if(sel(n) = 2,
                    100,
                    if(sel(n) = 3,
                      1000,
                      0)
                  )
                )
              )
      )
}
```

Nous remarquons la duplication de l'expression symbolique dans le signal `memory`. Un utilisateur décide de réduire les équations obtenues par abstraction en introduisant la règle suivante :

$$memory(n+2) \rightarrow Func_memory(rst, sel)$$

L'idée est de remplacer l'expression symbolique de `memory` par une fonction non interprétée *Func_memory*, tout en gardant comme arguments de la fonction les paramètres de la mémoire (`rst` et `sel`). Les équations deviennent alors :

{

NextSig(memory(n+2), Func_memory(sel(n), rst(n)))

,

NextSig(output(n+2), int(n)) × Func_memory(sel(n), rst(n)))

}

Certes, le choix de l'abstraction dépend des résultats et des propriétés qu'on souhaite vérifier. Dans l'exemple précédent, nous pouvons procéder autrement en bloquant l'évaluation du signal interne `memory`.

4.5 Conclusion

Nous avons présenté notre algorithme de simulation symbolique qui utilise le modèle SER. L'algorithme est implémenté en Mathematica. Pour obtenir une performance optimale nous avons utilisé ses techniques de simplification par substitutions. Aussi, nous avons défini une méthodologie de test symbolique basée sur la distinction entre la partie contrôle et la partie donnée du système.

Trois modes de simulation sont proposés (raisonnement, exécution et mixte). Nous avons illustré ces modes sur un registre à décalage. Les modes influencent la simulation en modifiant l'entrée. Nous avons montré deux techniques pour réduire le modèle SER. Donc la simulation se déroule sur un modèle plus compact et la performance est améliorée.

Chapitre 5

5 La méthodologie de Vérification

5.1 Présentation générale

Les résultats de la simulation symbolique sont les expressions symboliques, une pour chaque objet de la conception, qui sont calculées par le modèle SER. Elles sont souvent très difficiles à lire et à comprendre. Aussi, ces résultats ne montrent pas directement le comportement désiré du système. Dans ce but, nous proposons une méthodologie de vérification autour de notre approche de simulation symbolique.

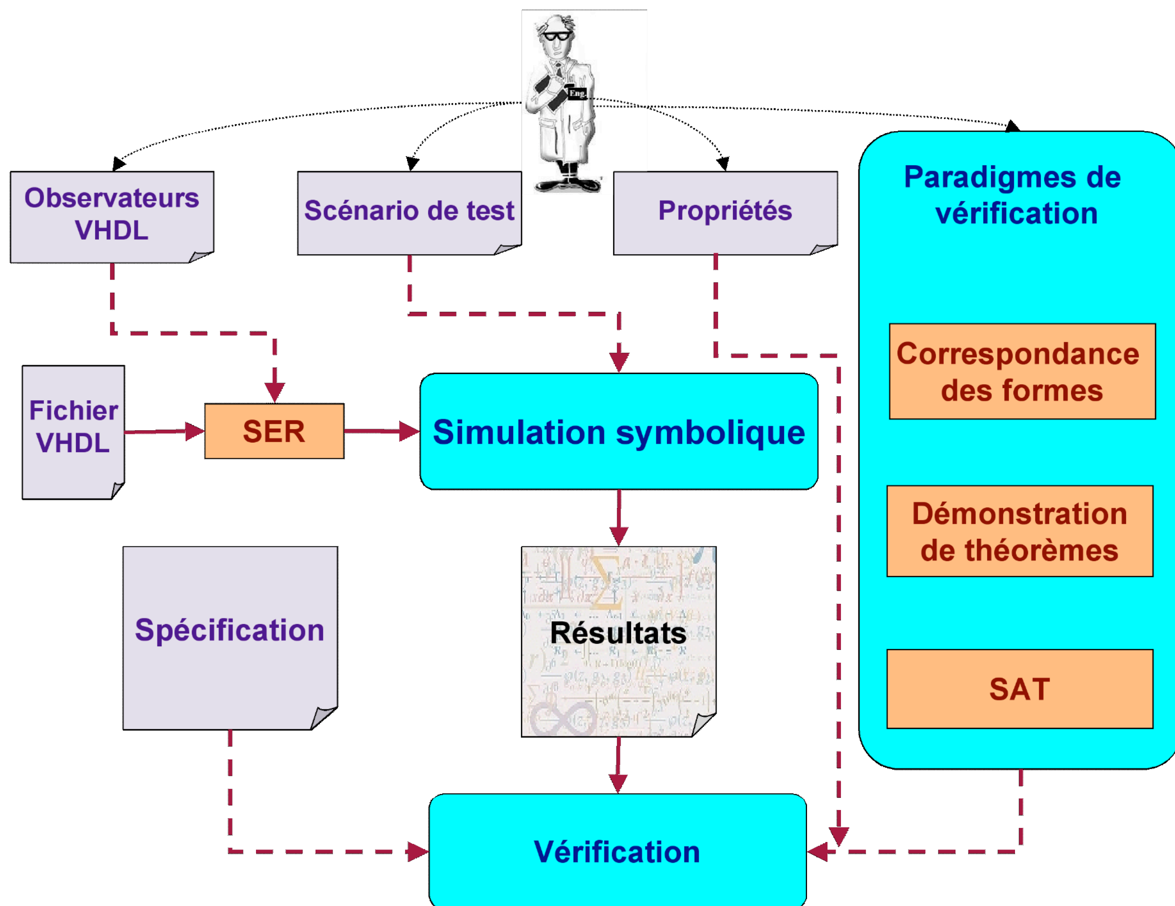


Figure 5.1 : vue de la méthodologie de vérification.

La figure 5.1 montre une vue de la méthodologie. L'utilisateur choisit le scénario de test approprié pour obtenir les expressions symboliques. Ensuite, il décrit soit des observateurs VHDL qui regardent le comportement attendu du système, soit des propriétés que le modèle doit satisfaire. Le fichier et les observateurs VHDL sont traduits en SER et ensuite simulés

symboliquement avec le scénario donné. Puis les résultats symboliques sont utilisés pour valider les observateurs ou pour prouver les propriétés en utilisant les spécifications de la conception et les contraintes de l'environnement. L'idée est d'appliquer un ou plusieurs paradigmes de vérification : la correspondance de forme, la démonstration de théorèmes ou SAT.

Nous allons détailler dans ce chapitre cette méthodologie appuyée par des exemples réels et des applications sur des circuits industriels.

5.2 Le paradigme de correspondance de formes

5.2.1 La vérification par simulation

Lors d'une simulation numérique traditionnelle, l'utilisateur compare les résultats de la simulation avec des valeurs numériques calculées par un modèle de référence. Ce modèle est écrit dans un langage de plus haut niveau que le VHDL tel que C ou Matlab. Il est considéré comme la spécification du système. Ayant la possibilité d'introduire des expressions symboliques, nous généralisons ce principe.

Définition 5.1 : la vérification par simulation
Considérons un composant C=<$I, K_I, S, K_s, \{S_i\}_{0<i\leq m}$>, un scénario de simulation Δ sur n cycles de simulation et un cycle de simulation n_o où $n_o \leq n$. Considérons aussi une spécification $\{U_i\}_k$: un ensemble de k expressions symboliques données par l'utilisateur. La vérification par simulation consiste à vérifier l'expression P_E calculée par la relation suivante :

$$P_E = \underset{i=1}{\overset{i=k}{AND}} (U_i == S_i(n_o)) : k \in N, i \in N, 0<i \leq k \leq m$$

Donc, la vérification par simulation est une sorte de vérification d'équivalence ou une vérification de conformité. Ceci dépend de l'écart du niveau d'abstraction entre la spécification et les résultats de la simulation symbolique. La simulation numérique est un cas particulier de cette définition quand les expressions U_i sont exclusivement des valeurs numériques.

La vérification symbolique de ces propriétés est réalisée par la correspondance des formes que nous avons présentées dans le chapitre 2. Cette vérification est adaptée à des circuits qui contiennent des parties données de taille importante. D'abord, nous décrivons la spécification U_i en utilisant les motifs et les formes. Nous appliquons les fonctions de correspondance de forme

(SameQ, MemberQ, FreeQ…etc.) pour identifier les expressions symboliques résultant du calcul avec leur spécification.

Dans ce chapitre, nous montrons deux études de cas sur la vérification par ce paradigme : un sur un circuit de traitement du signal et un autre sur une mémoire.

5.2.2 Étude d'un filtre RIF

Un filtre est un système visant à modifier la distribution fréquentielle des composantes d'un signal. Les circuits intégrés permettent désormais de réaliser en temps réel les fonctions de transfert des filtres. Généralement, le cahier des charges définit le gabarit fréquentiel G(f) du filtre à réaliser. C'est la réponse fréquentielle du filtre: si X(f) et Y(f) sont les transformées de Fourier des signaux d'entrée et de sortie, alors :

Y(f) = G(f).X(f)

Ceci traduit bien le fait que le système atténue plus ou moins les différentes composantes fréquentielles du signal d'entrée. Ramenée dans le domaine temporel, cette relation devient le produit de convolution suivant :

$$y(k) = \sum_{l=-\infty}^{+\infty} g(l).x(l-k)$$

où g(k) est la réponse impulsionnelle du filtre. Sa transformée en Z, G(z), constitue sa fonction de transfert. Pour que le filtre soit réalisable, il doit être causal. Ceci implique que la sommation précédente ne s'effectue que sur les $l>0$ puisque $g(l)=0$ pour $l<=0$. On peut diviser les filtres numériques en deux catégories : ceux dont la réponse impulsionnelle est infinie (RII) et ceux à réponse impulsionnelle finie (RIF).

Le filtre RIF (Réponse Impulsionnelle Finie), est utilisé dans plusieurs applications actuelles : transmission numérique, synthèse fréquentielle, codage du son, MP3,…etc.

La spécification du RIF

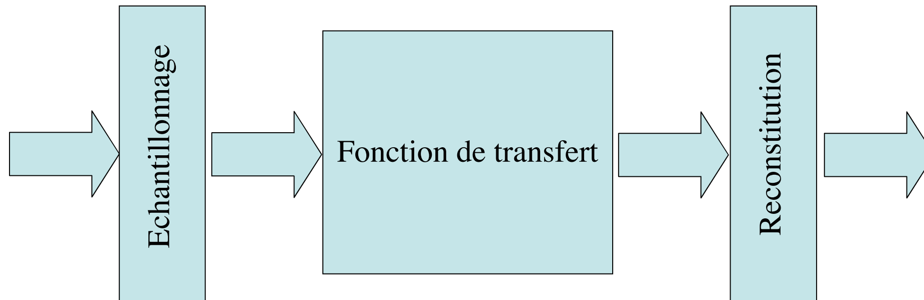


Figure 5.2 : filtrage du signal.

La figure 5.2 montre le principe du filtre. Un signal est d'abord échantillonné par rapport au temps. Ensuite, un algorithme de calcul est appliqué sur ces échantillons pour obtenir les échantillons filtrés. Ils sont finalement reconstitués pour donner le signal filtré. La spécification mathématique d'un filtre RIF d'ordre L est décrite par la série suivante :

$$y(n) = \sum_{i=0}^{L-1} x(n-i)h(i)$$

Avec :

y(n) : est l'échantillon filtré à l'instant *n*.

x(n-i) : est l'entrée à l'instant *(n-i)*.

h(i) : est le coefficient *i* du filtre.

Nous avons dans notre équipe une implémentation VHDL d'un filtre RIF. L'ordre du filtre est 32, et chaque échantillon est codé sur 16 bits. Le but de notre étude est de vérifier qu'une implémentation VHDL du filtre satisfait sa spécification mathématique.

L'implémentation VHDL

La figure 5.3 montre la structure de l'implémentation VHDL du RIF. Le circuit est synchronisé par une horloge `CLK` et dispose d'un signal `reset`. Les entrées (les échantillons) arrivent par le signal `Filter_IN`. Les signaux `ADC_Busy`, `ADC_Convstb` et `ADC_Rd_cs` forment l'interface entre le circuit et les convertisseurs (analogique/numérique et numérique/analogique).

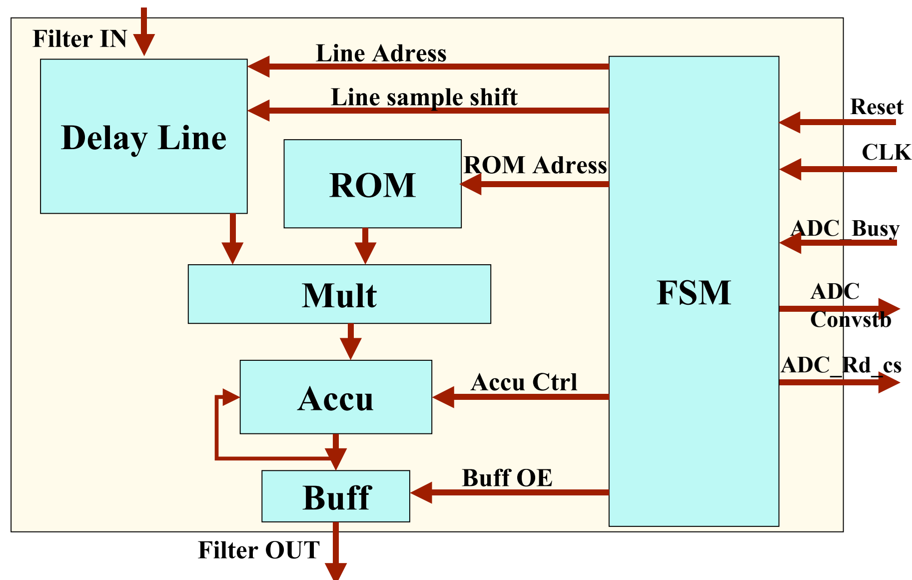


Figure 5.3 : vue de l'implémentation du RIF.

Le code VHDL est donné dans l'annexe 1. Il contient les composants suivants :

- **Delay_Line** : un registre à décalage qui mémorise les échantillons arrivant par `Filter_IN`.
- **ROM** : une mémoire morte où les coefficients du filtre sont écrits.
- **Mult** : un multiplieur qui donne le résultat de la multiplication sur 24 bits.
- **Accu** : un accumulateur où le résultat de chaque multiplication avec Mult est accumulé et stocké.
- **Buff** : un registre qui stocke le résultat final de l'algorithme.
- **FSM** : un composant qui implémente un automate d'états finis. Il est le responsable de la coordination entre les différents composants pour implémenter correctement l'algorithme. Il regroupe les états suivants :
    - Initialisation : la sortie *filter_out* et les registres internes dans les composants (Delay_Line, Accu et Buff) sont initialisés à zéro.
    - Décalage : *Delay_Line* est décalé d'une case et un nouvel échantillon est stocké à la place.
    - Additionner et multiplier : pendant 32 cycles d'horloge, les valeurs stockées dans *Delay_Line* sont multipliées par les valeurs écrites dans *ROM*. Le résultat est accumulé dans *Accu*.

- Sortie-prêt : est l'état final, le résultat du filtrage est mis dans *Buff* et Accu est initialisé.

La simulation symbolique du RIF

Nous simulons le filtre dans TheoSim (le M-code du filtre sont donnés dans l'annexe 2). Le scénario de test en mode mixte est présenté dans la figure 5.4. Tout d'abord, le système est initialisé avec `reset` pour un cycle d'horloge (2 cycles temporels). Ensuite, le filtre est simulé symboliquement en supposant que les signaux de contrôle de l'interface avec les convertisseurs restent désactivés et inchangés. Par contre, tous les autres signaux sont affectés symboliquement.

```
NextSig[adc$busy,0];
NextSig[clk,CLOCK[$SimulationTime]];
NextSig[filter$in,FILTER$IN[$SimulationTime]];
NextSig[reset,If[$SimulationTime<3,1,0]];
```

Figure 5.4 : le scénario de simulation du filtre.

Nous pouvons voir ce filtre comme s'il était préprogrammé à l'avance. Le scénario d'exécution et les séquences de contrôle sont décrits par le composant FSM. Donc nous exécutons ces séquences (l'automate de la *FSM*) et nous observons la partie opérative qui est le contenu des registres dans *Delay_Line*, *Accu* et *Buff*. La simulation nécessite 72 cycles temporels pour parcourir la totalité de l'automate de la *FSM* : 2 cycles d'initialisation + 2 cycles de décalage + 64 cycles d'addition et multiplication + 2 cycles de Sortie-prêt = 70. La valeur du *filter_out* est observable après un cycle d'horloge, nous ajoutons deux cycles supplémentaires et le compte est bon.

L'exécution est linéaire (voir figure) malgré l'utilisation du mode mixte. Ceci est dû au fait que toutes les valeurs des signaux contrôlant les composants (Mult, Rom, Accu,...etc.) sont données numériquement dans FSM. Donc la simulation symbolique du filtre a la même performance que la simulation symbolique en mode exécution. Les résultats de la figure 5.5 sont obtenus en utilisant une machine SUN-Blade 100 Sparc avec 640 MO de mémoire vive.

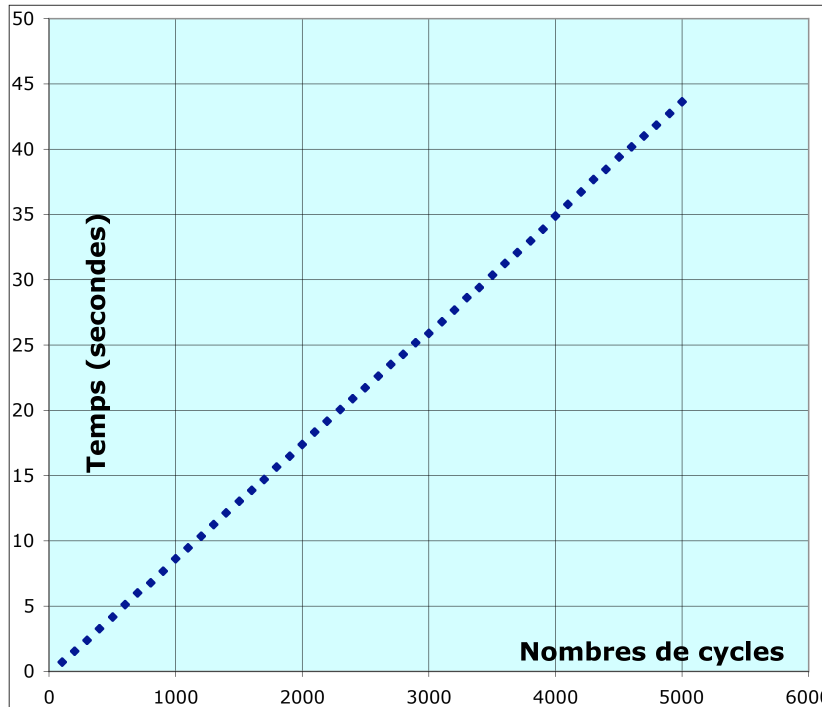


Figure 5.5 : évaluation du temps de la simulation symbolique du filtre.

Vérification de la fonctionnalité globale

La vérification de la fonctionnalité globale du filtre correspond à montrer que l'expression symbolique du signal Filter_out après 72 cycles temporels est conforme à la spécification mathématique :

$$y(n) = \sum_{i=0}^{L-1} x(n-i)h(i)$$

L'expression symbolique du signal `filter_out` est illisible. Donc, pour vérifier que cette expression est conforme à la spécification, nous utilisons la correspondance de forme.

Nous avons décrit la spécification en syntaxe Mathematica en utilisant les signaux de la description VHDL. La valeur *y(n)* est présentée par la variable **Spec**. La somme est réalisée par la fonction Mathematica **Sum**[*expr*, {*i*, *min*, *max*}] où *i* varie entre 0 et 31. L'expression *expr* est la multiplication du coefficient *i* de la mémoire Rom (**indexed**[***filter$rom***, ***i***]) par l'élément *i* du registre à décalage Delay_Line (**indexed**[**u2$x**, ***i***]). Nous rappelons que l'élément *i* du registre à décalage correspond à l'entrée (*n* - *i*) dans la spécification. La formule compactée de **Spec** est :

```
Spec :=
  conv$std$logic$vector[
        Sum[
             Times[indexed[filter$rom,i]
                   ,indexed[u2$x,i]]]]
             ,{i,0,31}]
       ,21]
```

Dans la formule complète, ***convstdlogic$vector*** est appliquée quasiment sur chaque expression et sous-expression de la fonction **Sum**. Pour des raisons de lisibilité, nous avons gardé seulement son application sur la somme. La fonction ***convstdlogic$vector*** transforme les entiers en vecteurs de 21 éléments de type standard_logic.

Nous avons utilisé la fonction SameQ pour prouver que *Spec* est identique à la valeur calculée du signal filtrer_out :

```
SameQ[Spec, filter$out] ;
```

La fonction renvoie « True », et la fonctionnalité globale de l'implémentation par rapport à la spécification mathématique est vérifiée.

Cette propriété ne peut être vérifiée automatiquement avec une autre approche de vérification formelle.

5.2.3 Vérification de la mémoire SPSMALL

La mémoire SPSMALL est fabriquée par STMicroelectronics. Elle est conçue initialement au niveau transistor, et pour l'incorporer dans de nouveaux circuits, une abstraction RTL est produite par l'outil TLL de Transeda [BDP01]. Cette version RTL est l'équivalent structurel de la synthèse de SPSMALL. Une autre version dite *hdl* de plus haut niveau est utilisée dans la conception des systèmes sur puce. Le comportement de la mémoire (voir Figure 5.6) est simple. Il existe deux modes de fonctionnement : lecture et écriture.

1. Quand le circuit est en mode « lecture », la sortie Q doit contenir le mot sélectionné par l'adresse d'entrée, si le signal de contrôle OEN est à '1'.
2. Quand le circuit est en mode « écriture », la donnée à l'entrée est enregistrée dans la bonne adresse spécifiée à l'entrée, si le signal de contrôle OEN est à '0'.

Les deux versions du SPSMALL (RTL et *hdl*) sont vérifiées formellement par l'outil Innologic de Synopsys [LN03], qui est basé sur la simulation symbolique de STE [BBS91]. Le temps de vérification augmente exponentiellement avec la taille de mémoire. Donc la vérification complète est impossible.

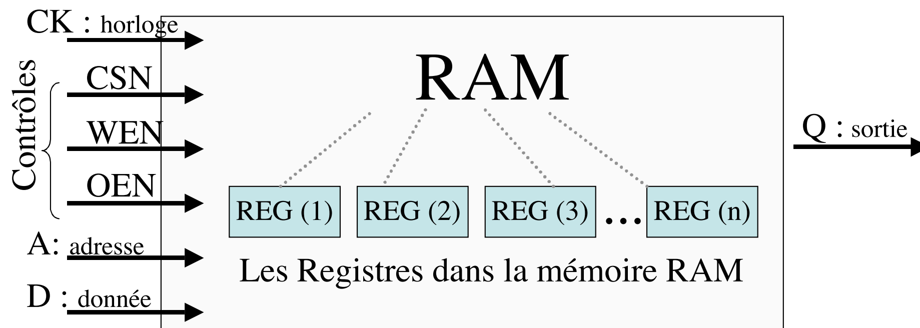


Figure 5.6 : la mémoire SPSMALL

Après la génération du modèle simulable (SER), nous avons simulé le niveau *hdl* du SPSMALL avec le mode raisonnement. Le résultat de la simulation symbolique est un ensemble d'équations récurrentes qui forme la fonction de transition de haut niveau. C'est-à-dire que nous avons une équation récurrente pour chaque registre et pour la sortie après un cycle d'horloge.

Le principe de la vérification consiste à injecter des symboles à l'entrée et voir si ces symboles se propagent dans la mémoire en respectant la spécification. Donc le paradigme de vérification appropriée est la correspondance de forme.

Si nous supposons qu'à l'entrée du circuit à un cycle d'horloge n, la donnée est « $D(n)$ » et l'adresse est « $A(n)$ », TheoSim calcule l'expression symbolique du registre $Reg_i(n+1)$ et la sortie Q(n+1). La spécification du SPSMALL est fournie par STMicroelectronics. Nous avons réussi à prouver la correction du SPSMALL vis-à-vis de cette spécification, indépendamment de la taille de la mémoire en formalisant les propriétés suivantes :

Propriété 5.1 : correction de l'écriture
Si (CSN = 0) et (WEN = 0), la donnée « D(n)» est sauvegardée à l'adresse « A(n) », si « A(n) » existe :

$$(CSN(n)=0 \wedge WEN(n)=0 \wedge A(n) \wedge D(n)) \Rightarrow \text{if } (0 \leq A(n) \leq Max_address) \text{ then } Reg_{A(n)}(n+1) = D(n).$$

Propriété 5.2 : correction de la lecture
Si (CSN(n) = 0) et (WEN(n) =1), et si OEN(n) = 0 et si l'adresse « A(n) » existe, la sortie est la donnée qui est stockée dans « A(n) ».

$$(CSN(n)=0 \wedge WEN(n)=1 \wedge A(n) \wedge OEN=0) \Rightarrow \text{if } (0 \leq A(n) \leq Max_address) \text{ then } Q(n+1)= Reg_{A(n)}(n+1).$$

Propriété 5.3 : l'initialisation
Pour une mémoire de taille Max_address, Si OEN(n) =1 alors chaque bit de la sortie est égal à Z :
$(OEN(n) =1) \Rightarrow (indexed[Q,i]_{0\leq i \leq Max_address}) = Z$

La troisième propriété inclut dans son énoncé un parcours de chaque bit de la sortie. Donc, nous ne pouvons pas la vérifier indépendamment du nombre de bits constituant la sortie Q(n+1).
Cette vérification (indépendamment de la taille de la mémoire) ne demande pas de ressources importantes. C'est une autre application où notre méthodologie de vérification est efficace.

5.3 Les paradigmes : démonstration de théorèmes et SAT

Nous montrons ici comment vérifier des propriétés logiques ou temporelles sur les résultats de la simulation symbolique. Ce genre de propriété est habituellement vérifié par d'autres méthodes formelles comme la vérification de modèle, pour laquelle il existe des outils industriels performants. Cependant, nous montrons qu'il est possible de vérifier le même genre de propriétés sur les expressions sortant du simulateur symbolique, mais avec une autre technique. Cela montre l'idée de multiparadigme de notre approche et donne une autre vision du domaine de la vérification des propriétés logiques et temporelles.

5.3.1 Les propriétés logiques et temporelles

L'utilisateur peut définir ces propriétés avec deux logiques : propositionnelle [Tra95] et du 1er ordre sans quantificateurs [Smu95].

Définition 5.2 : une formule logique

Considérons l'expression *expr* qui est une expression *IF* définie sur un domaine $\boldsymbol{K}$. Une formule logique construite sur $\boldsymbol{K}$ est telle que :

- Vrai et Faux sont des formules.
- Si $\boldsymbol{K}$ est $\boldsymbol{B}$, alors *expr* est une formule.
- Si $\boldsymbol{K}$ est $\boldsymbol{N}$, $\boldsymbol{Z}$, ou $\boldsymbol{R}$ et si $expr_1, expr_2 \in \boldsymbol{K}$ alors, $expr1 \lozenge expr2$ est une formule, avec $\lozenge \in \{<, =, >, \neq, \leq, \geq\}$.
- Si $expr_1$ et $expr_2$ sont des formules, $\blacklozenge$ est un opérateur booléen, alors $expr_1 \Rightarrow expr_2$, $expr_1 \Leftrightarrow expr_2$, $expr_1 \blacklozenge expr_2$ sont des formules.

Définition 5.3 : propriétés logiques

Considérons un composant C=<I, K_I ,S ,K_s ,$\{S_i\}_{0<i \leq m}$ >, un scénario de simulation Δ_n sur n cycles temporels. Une propriété logique est une formule sur K construite sur les expressions symboliques dans C et Δ_n.

Cette définition est inspirée des travaux de [BLS02] sur la logique CLU (logic of Counter arithmetic with Lambda expressions and Uninterpreted functions) présentée dans [UCLID].

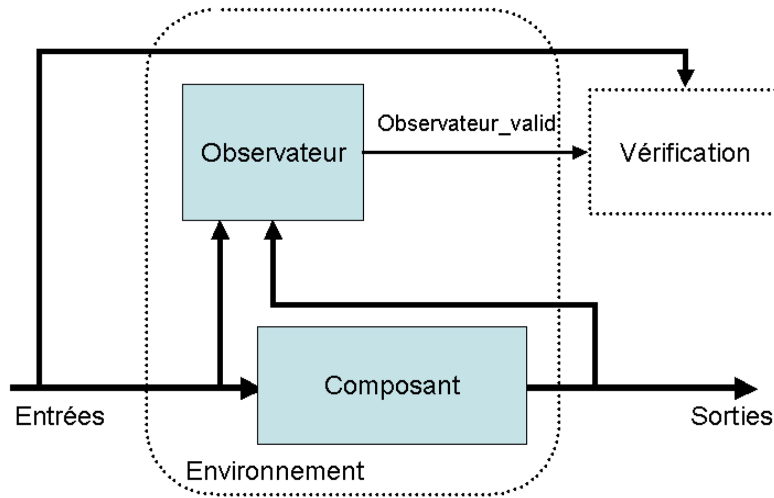


Figure 5.7 : l'observateur VHDL

Un observateur est un composant VHDL qui teste le comportement du système pendant la simulation [SDH00]. La figure 5.7 montre le concept d'un observateur. La vérification par observateur est d'assurer qu'un signal de sortie (habituellement appelé *valid*) est actif à chaque cycle de simulation. L'observateur est systématiquement composé avec le système, sans influencer son comportement.

Aujourd'hui les observateurs sont utilisés pour valider par simulation les propriétés temporelles du système en logique LTL [CGP00]. Ces propriétés peuvent être écrites en PSL [PSL04] et traduites ensuite par des outils comme FoCs [WWW01]. Nous disposons dans l'équipe VDS d'un générateur alternatif d'observateurs VHDL à partir d'une propriété écrite en PSL [BLO05]. Nous proposons dans notre méthodologie de générer un modèle SER pour le système composé (observateur + circuit). Après la simulation symbolique nous vérifions que l'expression symbolique de l'observateur est toujours vraie.

Définition 5.4 : la preuve de propriété par observateur
Soit C=<I, K_I ,S ,K_s ,$\{S_i\}_{0<i\leq m}$ >, le modèle SER de l'environnement (composant + observateur). Soit Δ_n un scénario de simulation en mode raisonnement sur n cycles temporels. Soit l'équation récurrente du signal *valid* de l'observateur :

$$S_{obs}(n+1) = f_{obs}(n)$$

La preuve de la propriété consiste à montrer que l'expression $f_{obs}(n)$ est toujours vraie : $\Delta_n \Rightarrow (f_{obs}(n) = Vrai)$.

5.3.2 Vérification par SAT

Le problème de satisfaction booléenne (SAT) consiste en la détermination d'une configuration des valeurs qui satisfait la formule ou prouve qu'elle est fausse. Ce problème est NP-complet. La plupart des techniques de résolution SAT [MSS99, Zha97, MMZ01, GZA02] sont basées sur la technique de recherche arborescente utilisant la décomposition des variables (DPLL) décrite dans [DLL62]. Pour cela, la formule est exprimée sous forme normale conjonctive (CNF).

Mathematica implémente une technique de résolution SAT dans la fonction `FindInstance` que nous avons présentée dans le chapitre 2. Nous employons cette fonction pour prouver que la formule d'un observateur VHDL est toujours vraie (définition 5.4).

Pour cela nous transformons les résultats de la simulation symbolique qui sont normalisés en Ife en combinaison des formules en utilisant la relation suivante :

$$Ife(x, y, z) = Or(And(x,y), And(Not(x),z))$$

Par exemple, le code VHDL suivant implémente un arbitre de bus à 2 clients où la priorité est pour le client 1.

```
entity two_arbiter  is
   port (   clock    : in  bit;
            reset    : in  bit;
            req1     : in  bit;
            req2     : in  bit;
            ack1     : out bit;
            ack2     : out bit);
end two_arbiter ;

architecture behavior of two_arbiter is
begin  -- behavior
  synchronous: process (clock, reset)
    begin  -- process synchronous
```

```
    if reset = '0'
      ack1<='0';
      ack2<='0';
    elsif clock'event and clock = '1' then  -- rising clock edge
       ack1<='0';
       ack2<='0';
       if req1='1' then
          ack1<='1';
       elsif req2='1' then
            ack2<='1';
       end if;
    end if;
  end process synchronous;
end behavior;
```

Nous ajoutons aussi un observateur VHDL qui assure que la priorité est donnée à ack1. Cet observateur est généré à partir de l'expression PSL suivante :

```
Always( req1 -> next(ack1) )
```

Cette expression peut s'écrire en Mathematica par : `Imply[REQ1$, ack1$1]`

`REQ1$` est req1(n) et `ack1$1` est ack1(n+1). L'opérateur `Always` traduit l'idée que nous validons l'expression pour des arguments symboliques quelconques.

La simulation symbolique en mode mixte (initialisation avec reset = 1, et ensuite complètement symbolique) donne la valeur suivante de la sortie :

```
ack1$1 = if[REQ1$ == 1, 1, 0]
```

Donc, nous souhaitons prouver :

```
Imply[REQ1$, if[REQ1$ == 1, 1, 0]]
```

En appliquant la transformation précédente et la définition logique de l'opérateur `Imply`, nous obtenons la formule suivante :

```
Or[
    Not[REQ1$],
    And[REQ1$, True],
    And[Not[REQ1$],
    False]
    ]
```

La fonction `FindInstance` est appliquée sur la négation de la formule précédente et renvoie « False ». C'est à dire que la propriété est prouvée.

5.3.3 Vérification par un démonstrateur de théorème

Mathematica n'est pas un démonstrateur de théorème. Il est vrai qu'il contient des fonctions qui effectuent des preuves élémentaires, mais des concepts comme la généralisation ou la base de données des lemmes pour la démonstration sont totalement inexistants. Par contre, il existe des démonstrateurs de théorème qui sont implémentés dans Mathematica comme Analytica [ClZ93], et THEOREMA [Buc01]. Ces outils sont restés des prototypes expérimentaux difficiles à utiliser. C'est pourquoi, nous avons préféré utiliser un démonstrateur de théorème externe à Mathematica : ACL2 [KMM00].

ACL2 est basé sur la logique du 1er ordre et écrit avec le langage LISP. Il a servi à prouver des circuits industriels de taille importante, comme des microprocesseurs [SaH02]. En conséquence, dans notre équipe de recherche, plusieurs travaux utilisent ACL2 pour la simulation symbolique des circuits décrits en VHDL [BGM01, GBO02].

Nous avons implémenté une connexion automatique entre Mathematica et ACL2, de manière à utiliser ACL2 comme un moteur de preuve pour Mathematica. La fonction Mathematica est **`ImplyAcl2`**`[`*hyp, expr*`]`. Elle formalise et envoie automatiquement le théorème équivalent à ($hyp \Rightarrow expr$) vers ACL2. Si ACL2 prouve le théorème automatiquement, la fonction renvoie True. Si ACL2 ne prouve pas le théorème, la réponse est NIL et nous ne pouvons pas savoir si la formule est fausse ou si nous avons besoin de lemmes intermédiaires pour le prouver.

Nous reprenons l'exemple de l'arbitre précédent, la propriété de causalité suivante est intéressante à prouver : $Ack1(n+1) \Rightarrow Req1(n)$. Ce théorème est écrit dans TheoSim : **`Imply`**`[ack1, REQ1]`, où `ack1` représente $Ack1(n+1)$ et `REQ1` représente $Req1(n)$. Nous utilisons la simulation symbolique pour obtenir l'expression symbolique de `ack1` : `ack1 = if[REQ1 == 1, 1, 0]`.

Nous pouvons prouver la propriété avec ACL2 en utilisant la fonction `ImpliesAcl2`.

Donc, nous formalisons en Mathematica que Req1(n) est de type bit : `bitp[REQ1]`.

Finalement, nous appelons ACL2 en Mathematica pour prouver la propriété en supposons que le **`reset`** est inactif :

```
ImpliesAcl2[And[bitp[REQ1,RESET],RESET==1,ack1==1],REQ1==1]
```

La variable ack1 est substituée automatiquement par la valeur obtenue pendant la simulation symbolique. La fonction `ImplyAcl2` envoie à ACL2 le théorème suivant :

```
(defthm theorem$5018
  (implies
            (And (integerp REQ1)
               (<   REQ1   2)
               (>   REQ1   -1)
               (integerp RESET)
               (<   RESET   2)
               (>   RESET   -1)
               (equal  RESET  1)
               (equal
                  (if   (equal   REQ1   1)   1   0)
                  1                 )
             )
          (equal  REQ1   1)
   )
)
```

La fonction `ImpliesAcl2` renvoie « True», et la propriété est prouvée avec ACL2. Nous avons utilisé cette approche, avec une version antérieure du prototype, pour prouver des propriétés sur des descriptions VHDL de haut niveau. Nous présentons ici deux exemples de ces applications.

Exemple 1 : un synthétiseur de fréquences

Les lignes suivantes décrivent en VHDL un circuit calculant le sinus de *x* pour un synthétiseur de fréquences inspiré de [JHM01]:

```
  …
      for j in 0 to n loop
       for i in 1 to 2*j+1 loop
        fact:= mult*fact;
        mult:= mult+1.0;
        exponent:= x*exponent;
       end loop ;
      accu :=-1.0*accu-(exponent/fact);
      end loop;
      y<= accu;
…
```

Le résultat de la simulation symbolique de ce circuit est de la forme :

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}$$

Cette expression est traduite en une fonction ACL2 *sym_res(x)*. Une fois que la série de Taylor pour le sinus est définie en ACL2 par la fonction récursive *taylor(x,n)*, le théorème suivant est montré par induction par ACL2 :

$$\text{Théorème} : \forall x \in Q, sym_res(x) = taylor(x,3)$$

Exemple 2 : un arbitre de bus

Nous avons vérifié un arbitre de bus dans une architecture composée de processeurs et de mémoires communiquant grâce à un bus ARM et un protocole maître-esclave (figure 5.8). L'arbitre attribue le bus d'adresses à un unique maître et le décodeur sélectionne le banc de mémoire correspondant à l'adresse d'un transfert. La vérification par démonstration de théorèmes sur les résultats symboliques permet de prouver que le décodeur choisit une unité existante, ceci pour un nombre de processeurs et de mémoires arbitraire, ainsi que pour des tailles arbitraires des bus d'adresses et de données.

Soit NB le nombre de mémoires et soit U l'unité choisie par le décodeur. Alors la propriété précédente s'exprime par la formule : U<NB. Cette propriété a été prouvée avec l'hypothèse que les adresses sont toujours inférieures au produit de NB par la taille de chaque mémoire, *i.e.* les adresses sont valides.

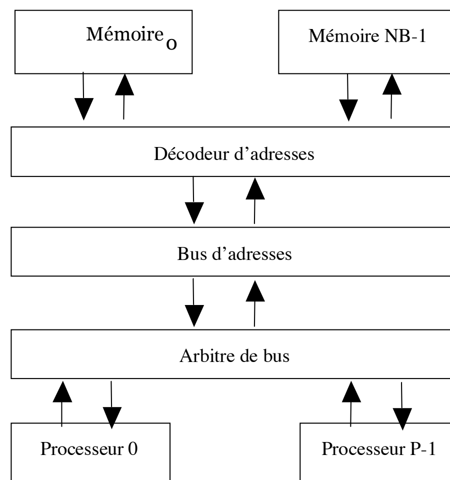


Figure 5.8 : architecture multiprocesseurs

5.4 Évaluation de la méthodologie

La méthodologie de vérification que nous avons présentée dans ce chapitre permet d'exploiter les résultats de la simulation symbolique. La vérification par correspondance de formes est efficace pour la vérification des circuits qui contiennent des parties de données importantes. Le filtre FIR et la mémoire SPSMALL sont des exemples de cette efficacité. Évidemment, les descriptions VHDL doivent être écrites au niveau algorithmique et avec types de donnés abstraits.

La technique SAT est utilisable pour vérifier des propriétés logiques ou temporelles sur des circuits de contrôle. Nous avons montré l'exemple d'un arbitre de bus. Nous n'avons pu appliquer cette technique sur des circuits de grande taille. En réalité, la fonction Mathematica `FindInstance` permet de résoudre des équations logiques de taille réduite. Donc pour traiter des circuits de taille importante, nous devons implémenter une connexion entre Mathematica et un moteur SAT performant. Cette tâche dépasse le cadre de la thèse. Pourtant, nous la considérons comme une perspective technique pour le prototype TheoSim.

La connexion entre Mathematica et ACL2 est utile pour prouver des propriétés par induction ou par analyse de cas. Dans ce cas, la simulation symbolique est un outil de formalisation automatique pour le démonstrateur. Cependant, un problème technique se manifeste si la propriété n'est pas vérifiée car l'implémentation actuelle ne permet pas d'analyser la sortie du démonstrateur. C'est pourquoi, les raisons de l'échec du démonstrateur sont très difficiles à trouver. Dans ce cas, nous ne pouvons pas utiliser cette connexion pour vérifier des circuits industriels. L'amélioration de l'implémentation actuelle demande des connaissances poussées dans la conception interne d'ACL2.

Chapitre 6

6 Conclusion et perspectives

6.1 Contributions

Nous avons proposé une approche automatique de simulation symbolique de haut niveau pour les circuits décrits en VHDL. Les contributions de nos travaux de recherche sont les suivantes :

- Un modèle mathématique basé sur les équations récurrentes permettant de simuler symboliquement des circuits écrits en VHDL de haut niveau. Le sous-ensemble VHDL considéré permet d'écrire des descriptions abstraites non considérées jusqu'à aujourd'hui par les outils automatiques de simulation symbolique. Il est défini dans le rapport technique [Als04,].
- Un algorithme d'extraction du modèle SER à partir de la description VHDL. Un prototype est implémenté en utilisant Lex, Yacc et Mathematica.
- Un algorithme de simulation symbolique autour du modèle SER qui respecte le standard VHDL. L'implémentation s'est servie du moteur de substitution de Mathematica pour rendre efficace la simulation symbolique.
- Une méthodologie de simulation symbolique, qui distingue plusieurs modes d'exécution de tests symboliques (raisonnement, exécution et mixte) et de réduction du modèle. Cette méthodologie rend l'utilisation du simulateur symbolique plus efficace et mieux orientée. Nous obtenons une performance linéaire en mode exécution, pour les circuits qui ont des parties données de taille importante tels que les circuits de traitement du signal. Nous avons publié cette méthodologie dans [AST03].
- Une méthodologie de vérification sur les résultats du simulateur symbolique. Nous avons montré comment plusieurs paradigmes de vérification sont utilisés pour prouver les propriétés du circuit, sur deux applications significatives. La vérification du filtre numérique FIR est faite à l'aide de la reconnaissance de formes implémentée dans Mathematica. Nous avons vérifié par ailleurs une spécification de haut niveau d'une mémoire industrielle en collaboration avec STMicroelectronics, également par correspondance de formes. La preuve est indépendante de la taille de la mémoire. Nous avons publié la vérification de ce circuit dans [ABC05]. Dans les deux cas, il s'agit d'une vérification de conformité entre une mise en œuvre RTL et une spécification algorithmique de haut niveau, qui dépasse les capacités des outils de vérification automatique industriels.

- Une connexion automatique entre Mathematica et le démonstrateur de théorème ACL2 (en collaboration avec P. Ostier de l'équipe VDS), dans le but de prouver des propriétés sur les résultats de la simulation symbolique. La formalisation des propriétés dans le démonstrateur de théorème est automatique, ainsi que l'envoi et la réception des résultats des preuves. Nous avons publié l'idée de la combinaison de Mathematica et ACL2 dans [ATS03]. La connexion entre Mathematica et ACL2 est utilisée dans notre équipe pour vérifier un réseau sur puce [AST04].

6.2 Perspectives

Notre approche est conçue pour simuler symboliquement un nombre déterminé d'objets. La taille d'un objet peut être symbolique, par contre le nombre d'objets doit être numérique. Par exemple, nous ne pouvons pas simuler avec notre approche une architecture d'additionneur composée de n additionneurs élémentaires, où n est un symbole. De même, les bornes inférieures et supérieures des instructions de génération d'objets telles que for-loop et for-generate doivent être précisées numériquement avant la simulation. Pour pouvoir simuler un nombre symbolique d'objets, le modèle SER doit être étendu. La récurrence est par rapport au temps dans notre modèle actuel. Nous proposons d'ajouter un paramètre d'espace en nous inspirant des travaux de V. VAN DONGEN et P. QUINTON sur l'utilisation des équations récurrentes affines pour la synthèse de réseaux systoliques [VaQ88], et les travaux de K. Morin-Allory sur la vérification des propriétés de sûreté avec le modèle polyédrique [CaM03].

Le prototype de simulation symbolique TheoSim implémenté en Mathematica peut être amélioré. L'implémentation d'une interface graphique est la première de ces améliorations. Le prototype a servi pour valider les idées et vérifier des circuits étudiés pendant la thèse. Nous avons découvert pendant son utilisation quelques limites du système Mathematica. Par exemple, une équation récurrente ne doit pas contenir une expression symbolique de plus de 256 imbrications. L'implémentation que nous avons effectuée doit être refaite pour contourner ces limites.

Il est important aussi de consolider la méthodologie de simulation symbolique et de vérification par un plus grand nombre d'expérimentations sur des circuits industriels. Finalement nous avons étudié le langage VHDL, le travail peut être porté sur d'autres langages notamment System Verilog.

7 Annexes

7.1 Implémentation VHDL du filtre FIR

```
-----------------------------filtre.vhd---------------------------------------
library ieee ;
use ieee.std_logic_1164.all ;
library lib_filtre ;
-- use lib_filtre.all;
--library current;
--use current.all;

entity filter is
       port(  filter_in      : in std_logic_vector(7 downto 0) ;
              clk            : in std_logic ;
              reset          : in std_logic ;
              adc_busy       : in std_logic ;
              adc_convstb : out std_logic ;
              adc_rd_csb     : out std_logic ;
              filter_out     : out std_logic_vector(7 downto 0 ) ) ;
end filter;

architecture a of filter is

--component romd_64x8m4d4_r0_m4_ns
--component rom
--    port (
 --       q   : out std_logic_vector(7 downto 0);
 --       a   : in  std_logic_vector(4 downto 0);
 --       ck  : in  std_logic
 --       csn : in  std_logic;
 --       oen : in  std_logic
 --   );

--end component;


  component accu
       port(  accu_in        : in std_logic_vector(15 downto 0) ;
              accu_ctrl      : in std_logic ;
              clk            : in std_logic ;
              reset          : in std_logic ;
              accu_out       : out std_logic_vector(20 downto 0)) ;
end component;

component buff
       port(  buff_in        : in std_logic_vector(7 downto 0) ;
              buff_oe        : in std_logic ;
              clk            : in std_logic ;
              reset          : in std_logic ;
              buff_out       : out std_logic_vector(7 downto 0)) ;
end component;

component fsm
       port(  clk                    : in std_logic ;
              reset                  : in std_logic ;
              adc_busy               : in std_logic ;
              adc_convstb            : out std_logic ;
              adc_rd_csb             : out std_logic ;
              rom_address            : out std_logic_vector(4 downto 0) ;
              delay_line_address     : out std_logic_vector(4 downto 0) ;
              delay_line_sample_shift       : out std_logic ;
              accu_ctrl              : out std_logic ;
              buff_oe                : out std_logic ) ;
end component;

component mult
       port(  mult_in_a      : in std_logic_vector(7 downto 0) ;
```

```
              mult_in_b      : in std_logic_vector(7 downto 0) ;
              mult_out       : out std_logic_vector(15 downto 0)) ;
end component;

component delay_line
       port(  delay_line_in         : in std_logic_vector(7 downto 0) ;
              delay_line_address    : in std_logic_vector(4 downto 0) ;
              delay_line_sample_shift       : in std_logic ;
              reset                 : in std_logic ;
              clk                   : in std_logic ;
              delay_line_out        : out std_logic_vector(7 downto 0)) ;
end component;

component rom
port(  rom_address    : in std_logic_vector(4 downto 0) ;
              rom_out        : out std_logic_vector(7 downto 0 )) ;
end component;

signal delay_line_sample_shift               : std_logic ;
signal accu_ctrl, buff_oe            : std_logic ;
signal delay_line_out, rom_out          : std_logic_vector(7 downto 0) ;
signal mult_out                         : std_logic_vector(15 downto 0) ;
--signal accu_out                         : std_logic_vector(15 downto 0) ;
signal accu_out                         : std_logic_vector(20 downto 0) ;
signal rom_address                   : std_logic_vector(5 downto 0) ;
signal delay_line_address            : std_logic_vector(4 downto 0) ;
signal zero                          : std_logic ;

for u1: rom use entity work.rom(a);

for u2: delay_line use entity work.delay_line(a);

for u3: mult use entity work.mult(a);

for u4: accu use entity work.accu(a);

for u5: buff  use entity work.buff(a);

for u6: fsm  use entity work.fsm(a);


begin

--zero <= '0' ;
--rom_address(5) <= '0';

--u1:romd_64x8m4d4_r0_m4_ns port map (
u1:rom port map (
   rom_address => rom_address,--(4 downto 0),
   rom_out            => rom_out
);

u2:delay_line port map (
   delay_line_in             => filter_in,
   delay_line_address        => delay_line_address,
   delay_line_sample_shift   => delay_line_sample_shift,
   clk                       => clk,
   reset                     => reset,
   delay_line_out            => delay_line_out
);

u3:mult port map (
   mult_in_a   => delay_line_out,
   mult_in_b   => rom_out,
   mult_out    => mult_out
);

u4:accu port map (
   accu_in     => mult_out,
   accu_ctrl   => accu_ctrl,
   clk         => clk,
```

```
   reset       => reset,
   accu_out    => accu_out
);

u5:buff port map (
   buff_in     => accu_out,--(19 downto 12),
   buff_oe     => buff_oe,
   clk         => clk,
   reset       => reset,
   buff_out    => filter_out
);

u6:fsm port map (
   clk                        => clk,
   reset                      => reset,
   adc_busy                   => adc_busy,
   adc_convstb                => adc_convstb,
   adc_rd_csb                 => adc_rd_csb,
   rom_address                => rom_address,--(4 downto 0),
   delay_line_address         => delay_line_address,
   delay_line_sample_shift    => delay_line_sample_shift,
   accu_ctrl                  => accu_ctrl,
   buff_oe                    => buff_oe
);

end a;


------------------------------accu.vhd----------------------------------------
library ieee ;
use ieee.std_logic_1164.all ;
--library arithmetic;
use ieee.std_logic_arith.all;

entity accu is
       port(  accu_in        : in std_logic_vector(15 downto 0) ;
              accu_ctrl      : in std_logic ;
              clk            : in std_logic ;
              reset          : in std_logic ;
              accu_out       : out std_logic_vector(20 downto 0)) ;

end accu;

architecture a of accu is
signal accu :unsigned(20 downto 0);
begin

    p_accu : process (clk)
    begin
       if (clk'event and clk='1') then
          if reset = '1' then
                       -- theosim ok
              accu <=0; -- (others => '0' ) ;
          elsif accu_ctrl ='1' then
              -- lors du premier cycle, l'accumulateur effectue seulement une
              -- operation de memorisation du resultat de la multiplication
               accu <= accu_in;--"00000" & unsigned(accu_in);
          else
              -- pour les autres cycles, l'accumulateur effectue l'addition entre
              -- l'entree et le resultat de l'operation precedente
            -- vhdl off theosim ok
              accu <= accu+accu_in;--accu +("00000" & unsigned(accu_in));
          end if ;
       end if;
    end process p_accu;

    accu_out <= conv_std_logic_vector(accu,21);  -- la sortie est affectee du resultat

end a;
------------------------------buff.vhd----------------------------------------
library ieee ;
```

```
use ieee.std_logic_1164.all ;

entity buff is
       port(  buff_in : in std_logic_vector(7 downto 0) ;
              buff_oe        : in std_logic ;
              clk            : in std_logic ;
              reset          : in std_logic ;
              buff_out       : out std_logic_vector(7 downto 0)) ;
end buff;

architecture a of buff is
begin
-- le buffer memorise l'entree et la propage ‡ la sortie seulement quand
-- le signal oe est actif.
p_buff:process (clk)
begin
      if (clk'event and clk='1') then
       if reset = '1' then
           buff_out <=0;-- (others => '0') ;
       else
           if (buff_oe = '1') then
              buff_out <= buff_in ;
           end if ;
       end if ;
     end if;
end process p_buff;
end a;

-----------------------------regdec.vhd----------------------------------------

library ieee ;
use ieee.std_logic_1164.all ;
-- library arithmetic ;
use ieee.std_logic_arith.all ;
use ieee.std_logic_unsigned.all;


entity delay_line is
       port(  delay_line_in          : in std_logic_vector(7 downto 0) ;
              delay_line_address     : in std_logic_vector(4 downto 0) ;
              delay_line_sample_shift       : in std_logic ;
              reset                  : in std_logic ;
              clk                    : in std_logic ;
              delay_line_out         : out std_logic_vector(7 downto 0)) ;
end delay_line;

architecture a of delay_line is

 type delay_line is array (0 to 31) of std_logic_vector(7 downto 0);
signal x : delay_line ;
begin


-- a completer !
p_d1:process (clk,reset)
begin
     if clk'event and clk='1'then
     if reset='1' then
         x(0) <="00000000"; x(1) <="00000000"; x(2) <="00000000"; x(3) <="00000000";
         x(4) <="00000000"; x(5) <="00000000"; x(6) <="00000000"; x(7) <="00000000";
         x(8) <="00000000"; x(9) <="00000000"; x(10) <="00000000"; x(11) <="00000000";
         x(12) <="00000000"; x(13) <="00000000"; x(14) <="00000000"; x(15) <="00000000";
         x(16) <="00000000"; x(17) <="00000000"; x(18) <="00000000"; x(19) <="00000000";
         x(20) <="00000000"; x(21) <="00000000"; x(22) <="00000000"; x(23) <="00000000";
         x(24) <="00000000"; x(25) <="00000000"; x(26) <="00000000"; x(27) <="00000000";
         x(28) <="00000000"; x(29) <="00000000"; x(30) <="00000000"; x(31) <="00000000";

     elsif reset='0' and      delay_line_sample_shift='1' then
       x(31) <= x(30);   x(30) <= x(29);   x(29)<= x(28);   x(28) <= x(27);
       x(27) <= x(26);   x(26) <= x(25);   x(25)<= x(24);   x(24) <= x(23);
       x(23) <= x(22);   x(22) <= x(21);   x(21)<= x(20);   x(20) <= x(19);
```

```
       x(19) <= x(18);   x(18) <= x(17);   x(17)<= x(16);   x(16) <= x(15);
       x(15) <= x(14);   x(14) <= x(13);   x(13)<= x(12);   x(12) <= x(11);
       x(11) <= x(10);   x(10) <= x(9);   x(9)<= x(8);   x(8) <= x(7);
       x(7) <= x(6);   x(6) <= x(5);   x(5)<= x(4);   x(4) <= x(3);
       x(3) <= x(2);   x(2) <= x(1);   x(1)<= x(0);   x(0) <= delay_line_in;
     end if;

end if;
end process;



       delay_line_out <= x(conv_integer(delay_line_address));
end a;
------------------------------contr.vhd---------------------------------------

library ieee ;
use ieee.std_logic_1164.all ;
-- library arithmetic;
use ieee.std_logic_arith.all;


entity fsm is
       port(  clk                   : in std_logic ;
              reset                 : in std_logic ;
              adc_busy              : in std_logic ;
              adc_convstb           : out std_logic ;
              adc_rd_csb            : out std_logic ;
              rom_address           : out std_logic_vector(4 downto 0) ;
              delay_line_address    : out std_logic_vector(4 downto 0) ;
              delay_line_sample_shift       : out std_logic ;
              accu_ctrl             : out std_logic ;
              buff_oe               : out std_logic ) ;
end fsm;
-- machine ‡ Ètats, contrÙleur du filtre.

architecture a of fsm is
type state is (s0,s1,s2,s3);
signal current_state, next_state : state ;
signal tap_number, next_tap_number : integer ;

begin
       p_state:process(clk)
       -- mÈmorisation des variables d'etats
       begin
              if (clk='1' and clk'event ) then
       -- initialisation ‡ l'etat 0 sous contrÙle de "reset"
                 if (reset='1') then
                     current_state <= s0 ;
                     tap_number <= "-----" ;
                 else

                     tap_number <= next_tap_number ;
                            current_state <= next_state ;
                 end if ;
              end if;
       end process p_state;

       p_fsm:process(current_state, tap_number, adc_busy)
       begin
       case current_state is
               when s0 =>
                     delay_line_address <="-----";
                     rom_address <= "-----";
                     accu_ctrl <= '0';
                     buff_oe <= '0';
                     adc_convstb <= '0';
                     next_tap_number <= 1;
                     if (adc_busy='0') then
                            delay_line_sample_shift <= '1';
                            adc_rd_csb <= '0';
```

```
                                next_state <= s1;
                        else
                                delay_line_sample_shift <= '0';
                                adc_rd_csb <= '1';
                                next_state <= s0;
                        end if;

                when s1=>
                        delay_line_sample_shift <= '0';
                        delay_line_address <= "00000";
                        rom_address <= "00000";
                        accu_ctrl <= '1';
                        buff_oe <= '0';
                        adc_rd_csb <= '1';
                        adc_convstb <= '1';
                        next_tap_number <= 1;
                        next_state <= s2;


                when s2 =>
                        accu_ctrl <= '0';
                        buff_oe <= '0';
                        delay_line_sample_shift <= '0';
                        adc_rd_csb <= '1';
                        adc_convstb <= '0';

                        delay_line_address <= conv_std_logic_vector(tap_number,5) ;
                        rom_address <= conv_std_logic_vector(tap_number,5) ;
                        next_tap_number <= tap_number + 1;
                        if tap_number = 32 then
                                next_state <= s3;
                        else
                                next_state <= s2 ;
                        end if ;


                when s3 =>
                        delay_line_address <=  conv_std_logic_vector(tap_number,5);--
                          --"-----";
                        rom_address <= conv_std_logic_vector(tap_number,5);--"-----";
                        accu_ctrl <= '0';
                        buff_oe <= '1';
                        delay_line_sample_shift <= '0';
                        adc_rd_csb <= '1';
                        adc_convstb <= '0';
                        next_tap_number <= 0 ;
                        next_state <= s0;

        end case;
        end process p_fsm;

end a;


------------------------------mult.vhd----------------------------------------

library ieee ;
use ieee.std_logic_1164.all ;
-- library arithmetic;
use ieee.std_logic_arith.all;

entity mult is
       port(  mult_in_a      : in std_logic_vector(7 downto 0) ;
              mult_in_b      : in std_logic_vector(7 downto 0) ;
              mult_out       : out std_logic_vector(15 downto 0)) ;
end mult;


architecture a of mult is
begin
       p_mult:process(mult_in_a,mult_in_b)
```

```
       begin
-- signee
--        mult_out <= std_logic_vector(signed(mult_in_a) * signed(mult_in_b));

-- non signee
          mult_out <= conv_std_logic_vector((unsigned(mult_in_a) * unsigned(mult_in_b)),16);

       end process p_mult;
end a;

------------------------------rom.vhd------------------------------------

library ieee ;
use ieee.std_logic_1164.all ;
-- library arithmetic ;
use ieee.std_logic_arith.all ;
use ieee.std_logic_unsigned.all;

entity rom is
       port(  rom_address    : in std_logic_vector(4 downto 0) ;
              rom_out        : out std_logic_vector(7 downto 0 )) ;
end rom;

architecture a of rom is
type tab_rom is array (0 to 31) of std_logic_vector(7 downto 0);
constant filter_rom : tab_rom :=
       ( 0  => "00001101" , 1  => "00010101" , 2  => "00011111" , 3  => "00101100" ,
            --  0x0d               0x15               0x1f               0x2c
         4  => "00111100" , 5  => "01001101" , 6  => "01100001" , 7  => "01110101" ,
            --  0x3c               0x4d               0x61               0x75
         8  => "10001010" , 9  => "10011111" , 10 => "10110011" , 11 => "11000101" ,
            --  0x8a               0x9f               0xb3               0xc5
         12 => "11010100" , 13 => "11100001" , 14 => "11101001" , 15 => "11101110" ,
            --  0xd4               0xe1               0xe9               0xee
         16 => "11101110" , 17 => "11101001" , 18 => "11100001" , 19 => "11010100" ,
            --  0xee               0xe9               0xe1               0xd4
         20 => "11000101" , 21 => "10110011" , 22 => "10011111" , 23 => "10001010" ,
            --  0xc5               0xb3               0x9f               0x8a
         24 => "01110101" , 25 => "01100001" , 26 => "01001101" , 27 => "00111100" ,
            --  0x75               0x61               0x4d               0x3c
         28 => "00101100" , 29 => "00011111" , 30 => "00010101" , 31 => "00001101" ) ;
            --  0x2c               0x1f               0x15               0xd

begin

   rom_out <= filter_rom(conv_integer(rom_address)) ;

end a;
```

7.2 Fichiers M-code du filtre FIR

```
Clear[accu$a];

SetAttributes[accu$a, HoldAll];

accu$a[accu_, accu$0_, accu$1_, accu$ctrl_, accu$in_, accu$out_, accu$out$1_, clk_, clk$0_,
reset_]:=

Module[{},

Process[p$accu, If[Wait[clk],

{start[]
,

NextSig[accu$1,
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        0       ,
        Ife[equal[accu$ctrl, 1],
          accu$in          ,
          plus[accu, accu$in]          ]
        ]
      ,
      accu$1      ]
]

}]];

 (*fin de process  p$accu*)


Process[process$1329, If[Wait[accu],

{start[]
,

NextSig[accu$out$1,
      conv$std$logic$vector[accu, 21]]

}]];

 (*fin de process  process$1329*)

UpDate[{accu, accu$0, accu$1, accu$ctrl, accu$in, accu$out, accu$out$1, clk, clk$0, reset}];

];(* end of module *)



Clear[buff$a];

SetAttributes[buff$a, HoldAll];

buff$a[buff$in_, buff$oe_, buff$out_, buff$out$1_, clk_, clk$0_, reset_]:=

Module[{},

Process[p$buff, If[Wait[clk],

{start[]
,

NextSig[buff$out$1,
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
```

```
        0        ,
        Ife[equal[buff$oe, 1],
          buff$in          ,
          buff$out$1         ]
        ]
      ,
      buff$out$1      ]
]

}]];

 (*fin de process  p$buff*)

UpDate[{buff$in, buff$oe, buff$out, buff$out$1, clk, clk$0, reset}];

];(* end of module *)

Clear[delay$line$a];

SetAttributes[delay$line$a, HoldAll];

delay$line$a[clk_, clk$0_, delay$line$address_, delay$line$address$0_, delay$line$in_,
delay$line$out_, delay$line$out$1_, delay$line$sample$shift_, reset_, reset$0_, x_, x$0_, x$1_]:=

Module[{},

Process[p$d1, If[Wait[clk, reset],

{start[]
,

NextSig[indexed[x$1, 0],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          delay$line$in          ,
          indexed[x$1, 0]          ]
        ]
      ,
      indexed[x$1, 0]      ]
]

,

NextSig[indexed[x$1, 1],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 0]          ,
          indexed[x$1, 1]          ]
        ]
      ,
      indexed[x$1, 1]      ]
]

,

NextSig[indexed[x$1, 2],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 1]          ,
          indexed[x$1, 2]          ]
        ]
      ,
      indexed[x$1, 2]      ]
]
```

```
,

NextSig[indexed[x$1, 3],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 2]          ,
          indexed[x$1, 3]          ]
        ]
      ,
      indexed[x$1, 3]      ]
]

,

NextSig[indexed[x$1, 4],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 3]          ,
          indexed[x$1, 4]          ]
        ]
      ,
      indexed[x$1, 4]      ]
]

,

NextSig[indexed[x$1, 5],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 4]          ,
          indexed[x$1, 5]          ]
        ]
      ,
      indexed[x$1, 5]      ]
]

,

NextSig[indexed[x$1, 6],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 5]          ,
          indexed[x$1, 6]          ]
        ]
      ,
      indexed[x$1, 6]      ]
]

,

NextSig[indexed[x$1, 7],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 6]          ,
          indexed[x$1, 7]          ]
        ]
      ,
      indexed[x$1, 7]      ]
]

,
```

```
NextSig[indexed[x$1, 8],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 7]         ,
          indexed[x$1, 8]         ]
        ]
      ,
      indexed[x$1, 8]      ]
]

,

NextSig[indexed[x$1, 9],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 8]         ,
          indexed[x$1, 9]         ]
        ]
      ,
      indexed[x$1, 9]      ]
]

,

NextSig[indexed[x$1, 10],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 9]         ,
          indexed[x$1, 10]         ]
        ]
      ,
      indexed[x$1, 10]      ]
]

,

NextSig[indexed[x$1, 11],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 10]         ,
          indexed[x$1, 11]         ]
        ]
      ,
      indexed[x$1, 11]      ]
]

,

NextSig[indexed[x$1, 12],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 11]         ,
          indexed[x$1, 12]         ]
        ]
      ,
      indexed[x$1, 12]      ]
]

,
```

```
NextSig[indexed[x$1, 13],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 12]          ,
          indexed[x$1, 13]          ]
        ]
      ,
      indexed[x$1, 13]      ]
]

,

NextSig[indexed[x$1, 14],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 13]          ,
          indexed[x$1, 14]          ]
        ]
      ,
      indexed[x$1, 14]      ]
]

,

NextSig[indexed[x$1, 15],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 14]          ,
          indexed[x$1, 15]          ]
        ]
      ,
      indexed[x$1, 15]      ]
]

,

NextSig[indexed[x$1, 16],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 15]          ,
          indexed[x$1, 16]          ]
        ]
      ,
      indexed[x$1, 16]      ]
]

,

NextSig[indexed[x$1, 17],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 16]          ,
          indexed[x$1, 17]          ]
        ]
      ,
      indexed[x$1, 17]      ]
]

,

NextSig[indexed[x$1, 18],
```

```
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 17]          ,
          indexed[x$1, 18]          ]
        ]
      ,
      indexed[x$1, 18]      ]
]

,

NextSig[indexed[x$1, 19],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 18]          ,
          indexed[x$1, 19]          ]
        ]
      ,
      indexed[x$1, 19]      ]
]

,

NextSig[indexed[x$1, 20],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 19]          ,
          indexed[x$1, 20]          ]
        ]
      ,
      indexed[x$1, 20]      ]
]

,

NextSig[indexed[x$1, 21],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 20]          ,
          indexed[x$1, 21]          ]
        ]
      ,
      indexed[x$1, 21]      ]
]

,

NextSig[indexed[x$1, 22],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 21]          ,
          indexed[x$1, 22]          ]
        ]
      ,
      indexed[x$1, 22]      ]
]

,

NextSig[indexed[x$1, 23],
      Ife[and[event[clk], equal[clk, 1]],
```

```
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 22]          ,
          indexed[x$1, 23]          ]
        ]
      ,
      indexed[x$1, 23]      ]
]

,

NextSig[indexed[x$1, 24],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 23]          ,
          indexed[x$1, 24]          ]
        ]
      ,
      indexed[x$1, 24]      ]
]

,

NextSig[indexed[x$1, 25],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 24]          ,
          indexed[x$1, 25]          ]
        ]
      ,
      indexed[x$1, 25]      ]
]

,

NextSig[indexed[x$1, 26],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 25]          ,
          indexed[x$1, 26]          ]
        ]
      ,
      indexed[x$1, 26]      ]
]

,

NextSig[indexed[x$1, 27],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 26]          ,
          indexed[x$1, 27]          ]
        ]
      ,
      indexed[x$1, 27]      ]
]

,

NextSig[indexed[x$1, 28],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
```

```
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 27]          ,
          indexed[x$1, 28]          ]
        ]
      ,
      indexed[x$1, 28]      ]
]

,

NextSig[indexed[x$1, 29],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 28]          ,
          indexed[x$1, 29]          ]
        ]
      ,
      indexed[x$1, 29]      ]
]

,

NextSig[indexed[x$1, 30],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 29]          ,
          indexed[x$1, 30]          ]
        ]
      ,
      indexed[x$1, 30]      ]
]

,

NextSig[indexed[x$1, 31],
      Ife[and[event[clk], equal[clk, 1]],
      Ife[equal[reset, 1],
        {0, 0, 0, 0, 0, 0, 0, 0}        ,
        Ife[and[equal[reset, 0], equal[delay$line$sample$shift, 1]],
          indexed[x, 30]          ,
          indexed[x$1, 31]          ]
        ]
      ,
      indexed[x$1, 31]      ]
]

}]];

 (*fin de process  p$d1*)


Process[process$1355, If[Wait[delay$line$address, x],

{start[]
,

NextSig[delay$line$out$1,
      indexed[x, conv$integer[delay$line$address]]]

}]];

 (*fin de process  process$1355*)

UpDate[{clk, clk$0, delay$line$address, delay$line$address$0, delay$line$in, delay$line$out,
delay$line$out$1, delay$line$sample$shift, reset, reset$0, x, x$0, x$1}];
```

```
];(* end of module *)

Clear[mult$a];

SetAttributes[mult$a, HoldAll];

mult$a[mult$in$a_, mult$in$a$0_, mult$in$b_, mult$in$b$0_, mult$out_, mult$out$1_]:=

Module[{},

Process[p$mult, If[Wait[mult$in$a, mult$in$b],

{start[]
,

NextSig[mult$out$1,
      conv$std$logic$vector[times[unsigned[mult$in$a], unsigned[mult$in$b]], 16]]

}]];

 (*fin de process  p$mult*)

UpDate[{mult$in$a, mult$in$a$0, mult$in$b, mult$in$b$0, mult$out, mult$out$1}];

];(* end of module *)

Clear[rom$a];

SetAttributes[rom$a, HoldAll];

rom$a[filter$rom$0_, rom$address_, rom$address$0_, rom$out_, rom$out$1_]:=

Module[{},

Process[process$1319, If[Wait[filter$rom, rom$address],

{start[]
,

NextSig[rom$out$1,
      indexed[filter$rom, conv$integer[rom$address]]]

}]];

 (*fin de process  process$1319*)

UpDate[{filter$rom$0, rom$address, rom$address$0, rom$out, rom$out$1}];

];(* end of module *)




Clear[filter$a];

SetAttributes[filter$a, HoldAll];

filter$a[accu$ctrl_, accu$ctrl$1_, accu$out_, accu$out$1_, adc$busy_, adc$convstb_,
adc$convstb$1_, adc$rd$csb_, adc$rd$csb$1_, buff$oe_, buff$oe$1_, clk_, delay$line$address_,
delay$line$address$1_, delay$line$out_, delay$line$out$1_, delay$line$sample$shift_,
delay$line$sample$shift$1_, filter$in_, filter$out_, filter$out$1_, mult$out_, mult$out$1_,
reset_, rom$address_, rom$address$1_, rom$out_, rom$out$1_, zero_, zero$1_, u1$filter$rom$0_,
u2$x_, u2$x$0_, u2$x$1_, u4$accu_, u4$accu$0_, u4$accu$1_, u6$current$state_,
u6$current$state$0_, u6$current$state$1_, u6$next$state_, u6$next$state$1_, u6$next$tap$number_,
u6$next$tap$number$1_, u6$tap$number_, u6$tap$number$0_, u6$tap$number$1_]:=

Module[{},

Component[u1, rom$a[u1$filter$rom$0, rom$address, rom$address$0, rom$out, rom$out$1]];
```

```
Component[u2, delay$line$a[clk, clk$0, delay$line$address, delay$line$address$0, filter$in,
delay$line$out, delay$line$out$1, delay$line$sample$shift, reset, reset$0, u2$x, u2$x$0,
u2$x$1]];


Component[u3, mult$a[delay$line$out, delay$line$out$0, rom$out, rom$out$0, mult$out,
mult$out$1]];


Component[u4, accu$a[u4$accu, u4$accu$0, u4$accu$1, accu$ctrl, mult$out, accu$out, accu$out$1,
clk, clk$0, reset]];


Component[u5, buff$a[accu$out, buff$oe, filter$out, filter$out$1, clk, clk$0, reset]];


Component[u6, fsm$a[accu$ctrl, accu$ctrl$1, adc$busy, adc$busy$0, adc$convstb, adc$convstb$1,
adc$rd$csb, adc$rd$csb$1, buff$oe, buff$oe$1, clk, clk$0, u6$current$state, u6$current$state$0,
u6$current$state$1, delay$line$address, delay$line$address$1, delay$line$sample$shift,
delay$line$sample$shift$1, u6$next$state, u6$next$state$1, u6$next$tap$number,
u6$next$tap$number$1, reset, rom$address, rom$address$1, u6$tap$number, u6$tap$number$0,
u6$tap$number$1]];

UpDate[{accu$ctrl, accu$ctrl$1, accu$out, accu$out$1, adc$busy, adc$convstb, adc$convstb$1,
adc$rd$csb, adc$rd$csb$1, buff$oe, buff$oe$1, clk, delay$line$address, delay$line$address$1,
delay$line$out, delay$line$out$1, delay$line$sample$shift, delay$line$sample$shift$1, filter$in,
filter$out, filter$out$1, mult$out, mult$out$1, reset, rom$address, rom$address$1, rom$out,
rom$out$1, zero, zero$1, u1$filter$rom$0, u2$x, u2$x$0, u2$x$1, u4$accu, u4$accu$0, u4$accu$1,
u6$current$state, u6$current$state$0, u6$current$state$1, u6$next$state, u6$next$state$1,
u6$next$tap$number, u6$next$tap$number$1, u6$tap$number, u6$tap$number$0, u6$tap$number$1}];

];(* end of module *)
```

7.3 Le code VHDL du paquetage

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


package sha_function is

function ch(e,f,g : in std_logic_vector(31 downto 0))return std_logic_vector;
function maj(a,b,c : in std_logic_vector(31 downto 0))return std_logic_vector;
function add2(h,e,ch1,a,maj1,wi32,ki32 : in std_logic_vector(31 downto 0))return
std_logic_vector;
function shift(x : in std_logic_vector(31 downto 0))return std_logic_vector;
function ajout2(x,y : in std_logic_vector(31 downto 0))return std_logic_vector;
function adressage2(count: in std_logic_vector(3 downto 0);t:in integer)return std_logic_vector;

type operation is  (opodoo,kangoo,fastoo);
constant idle: std_logic_vector(2 downto 0):="000";
constant init: std_logic_vector(2 downto 0):="001";
constant sha_ini_one:std_logic_vector(2 downto 0):="010";
constant calculw_one: std_logic_vector(2 downto 0):="011";
constant calcul_abc_one: std_logic_vector(2 downto 0):="100";
constant result: std_logic_vector(2 downto 0):="101";
constant resultw:std_logic_vector(2 downto 0):="110";
end ;



package body sha_function is


--function ch
function ch(e,f,g : in std_logic_vector(31 downto 0))return std_logic_vector is
variable ch1:std_logic_vector(31 downto 0);
begin
ch1:=(e and f)xor(not(e)and   g) ;

return ch1;
end ch;


--function maj
function maj(a,b,c : in std_logic_vector(31 downto 0))return std_logic_vector is
variable maj1:std_logic_vector(31 downto 0);
begin
maj1:=(a and b) xor(a and c) xor (b and c);
return maj1;
end maj;



--function add2
function add2(h,e,ch1,a,maj1,wi32,ki32 : in std_logic_vector(31 downto 0))return std_logic_vector
is
variable s:std_logic_vector(31 downto 0);
variable sig_2:std_logic_vector(31 downto 0);
variable sig:std_logic_vector(31 downto 0);
begin
sig_2:=(e(5 downto 0)&e(31 downto 6) xor e(10 downto 0)&e(31 downto 11) xor e(24 downto 0)&e(31
downto 25));
```

```
sig:=(a(1 downto 0)&a(31 downto 2) xor a(12 downto 0)&a(31 downto 13) xor a(21 downto 0)&a(31
downto 22));
s:=h +sig_2  + ch1 + sig + maj1+ ki32+ wi32;
return s;
end add2;



--shift
function shift(x : in std_logic_vector(31 downto 0))return std_logic_vector is
variable y:std_logic_vector(31 downto 0);
begin
y:=x(30 downto 0) &x(31);
return y;
end shift;




--ajout2
function ajout2(x,y : in std_logic_vector(31 downto 0))return std_logic_vector is
variable z:std_logic_vector(31 downto 0);
begin
z:=x+y;
return z;
end ajout2;



--adressage2-------designe les w utilent au calcul du suivant---
function adressage2(count: in std_logic_vector(3 downto 0);t:in integer)return std_logic_vector
is
variable x : std_logic_vector(3 downto 0);
begin
x:=count;

 if  t=18 then
       x:=count;


 elsif t=19  then

       x:=x+2;


 elsif t=20 then

       x:=x+8;


 elsif t=21 then

       x:=x+13;

  end if;
 return x;
end adressage2;



end sha_function;
```

7.4 Le LIF du paquetage

```
(

 (library ieee)

 (use
  (
   (ieee std_logic_1164) all))

 (use
  (
   (ieee std_logic_arith) all))

 (use
  (
   (ieee std_logic_unsigned) all))

 (package sha_function

  (

   (function ch return std_logic_vector

    (parameter

     (

      (in
       (std_logic_vector
        (
         (downto 31 0)))
       (e f g))
)))

   (function maj return std_logic_vector

    (parameter

     (

      (in
       (std_logic_vector
        (
         (downto 31 0)))
       (a b c))
)))

   (function add2 return
std_logic_vector

    (parameter

     (

      (in
       (std_logic_vector
        (
         (downto 31 0)))
       (h e ch1 a maj1 wi32 ki32))
)))

   (function shift return
std_logic_vector

    (parameter

     (

      (in
       (std_logic_vector
        (
         (downto 31 0))) x)
)))

   (function ajout2 return
std_logic_vector

    (parameter

     (

      (in
       (std_logic_vector
        (
         (downto 31 0)))
       (x y))
)))

   (function adressage2 return
std_logic_vector

    (parameter

     (

      (in
       (std_logic_vector
        (
         (downto 3 0))) count)

      (in integer t)
)))

   (type operation
    (enum
     (opodoo kangoo fastoo)))

   (constant
    (std_logic_vector
     (
      (downto 2 0))) idle "000")

   (constant
    (std_logic_vector
     (
      (downto 2 0))) init "001")

   (constant
    (std_logic_vector
     (
      (downto 2 0))) sha_ini_one "010")

   (constant
    (std_logic_vector
     (
      (downto 2 0))) calculw_one "011")

   (constant
    (std_logic_vector
     (
      (downto 2 0))) calcul_abc_one
"100")
```

```
   (constant
    (std_logic_vector
     (
      (downto 2 0))) result "101")

   (constant
    (std_logic_vector
     (
      (downto 2 0))) resultw "110")
)
)

 (package sha_function

  (

   (

    (function ch return std_logic_vector

     (parameter

      (

       (in
        (std_logic_vector
         (
          (downto 31 0)))
        (e f g))
)))

    (declarative

     (

      (variable
       (std_logic_vector
        (
         (downto 31 0))) ch1)
)
)

    (sequential
     (

      (

       (changevar ch1
        (xor
         (
          (and e f))
         (
          (and
           (not
            (e)) g))))

       (return ch1)
)
)))

   (

    (function maj return
std_logic_vector

     (parameter

      (

       (in
        (std_logic_vector
         (
          (downto 31 0)))
        (a b c))
)))

    (declarative

     (

      (variable
       (std_logic_vector
        (
         (downto 31 0))) maj1)
)
)

    (sequential
     (

      (

       (changevar maj1
        (xor
         (xor
          (
           (and a b))
          (
           (and a c)))
         (
          (and b c))))

       (return maj1)
)
)))

   (

    (function add2 return
std_logic_vector

     (parameter

      (

       (in
        (std_logic_vector
         (
          (downto 31 0)))
        (h e ch1 a maj1 wi32 ki32))
)))

    (declarative

     (

      (variable
       (std_logic_vector
        (
         (downto 31 0))) s)

      (variable
       (std_logic_vector
        (
         (downto 31 0))) sig_2)

      (variable
       (std_logic_vector
        (
         (downto 31 0))) sig)
```

```
)
)

    (sequential
     (

      (

       (changevar sig_2
        (
         (xor
          (xor
           (&
            (indexed e
             (
              (downto 5 0)))
            (indexed e
             (
              (downto 31 6))))
           (&
            (indexed e
             (
              (downto 10 0)))
            (indexed e
             (
              (downto 31 11)))))
          (&
           (indexed e
            (
             (downto 24 0)))
           (indexed e
            (
             (downto 31 25)))))))

       (changevar sig
        (
         (xor
          (xor
           (&
            (indexed a
             (
              (downto 1 0)))
            (indexed a
             (
              (downto 31 2))))
           (&
            (indexed a
             (
              (downto 12 0)))
            (indexed a
             (
              (downto 31 13)))))
          (&
           (indexed a
            (
             (downto 21 0)))
           (indexed a
            (
             (downto 31 22)))))))

       (changevar s
        (+
         (+
          (+
           (+
            (+
             (+ h sig_2) ch1) sig) maj1)
ki32) wi32))

       (return s)
```

```
)
)))

    (

     (function shift return
std_logic_vector

      (parameter

       (

        (in
         (std_logic_vector
          (
           (downto 31 0))) x)
)))

     (declarative

      (

       (variable
        (std_logic_vector
         (
          (downto 31 0))) y)
)
)

     (sequential
      (

       (

        (changevar y
         (&
          (indexed x
           (
            (downto 30 0)))
          (indexed x
           (31))))

        (return y)
)
)))

    (

     (function ajout2 return
std_logic_vector

      (parameter

       (

        (in
         (std_logic_vector
          (
           (downto 31 0)))
         (x y))
)))

     (declarative

      (

       (variable
        (std_logic_vector
         (
          (downto 31 0))) z)
```

```
)
)

    (sequential
     (

      (

       (changevar z
        (+ x y))

       (return z)
)
)))

   (

    (function adressage2 return
std_logic_vector

     (parameter

      (

       (in
        (std_logic_vector
         (
          (downto 3 0))) count)

       (in integer t)
)))

    (declarative

     (

      (variable
       (std_logic_vector
        (
         (downto 3 0))) x)
)
)

    (sequential
     (
```

```
      (

       (changevar x count)

       (if
        (= t 18)

        (

         (changevar x count)
)

        (elsif
         (= t 19)

         (

          (changevar x
           (+ x 2))
)

         (elsif
          (= t 20)

          (

           (changevar x
            (+ x 8))
)

          (elsif
           (= t 21)

           (

            (changevar x
             (+ x 13))
)))))

       (return x)
)
)))
)
)
)
```

7.5 Modèle statique du paquetage

```
indexed[ch,{expr___}]:=ch@@FlattenExpression[expr];

FunctionQ[ch]:= True;

indexed[maj,{expr___}]:=maj@@FlattenExpression[expr];

FunctionQ[maj]:= True;

indexed[add2,{expr___}]:=add2@@FlattenExpression[expr];

FunctionQ[add2]:= True;

indexed[shift,{expr___}]:=shift@@FlattenExpression[expr];

FunctionQ[shift]:= True;

indexed[ajout2,{expr___}]:=ajout2@@FlattenExpression[expr];

FunctionQ[ajout2]:= True;

indexed[adressage2,{expr___}]:=adressage2@@FlattenExpression[expr];

FunctionQ[adressage2]:= True;

CodeType[opodoo, kangoo, fastoo];

ChangeVar[idle, {0, 0, 0}];

ChangeVar[init, {0, 0, 1}];

ChangeVar[sha$ini$one, {0, 1, 0}];

ChangeVar[calculw$one, {0, 1, 1}];

ChangeVar[calcul$abc$one, {1, 0, 0}];

ChangeVar[result, {1, 0, 1}];

ChangeVar[resultw, {1, 1, 0}];
```

7.6 Modèle dynamique du paquetage

```
Clear[ch];
SetAttributes[ch,HoldAll];
ch[e_, f_, g_]:=
  Module[{ch1},start[];

    ChangeVar[ch1, xor[and[e, f], and[not[e], g]]];

    ChangeVar[return, xor[and[e, f], and[not[e], g]]]
];

Clear[maj];
SetAttributes[maj,HoldAll];
maj[a_, b_, c_]:=
  Module[{maj1},start[];

    ChangeVar[maj1, xor[xor[and[a, b], and[a, c]], and[b, c]]];
```

```
    ChangeVar[return, xor[xor[and[a, b], and[a, c]], and[b, c]]]
];

Clear[add2];
SetAttributes[add2,HoldAll];
add2[a_, ch1_, e_, h_, ki32_, maj1_, wi32_]:=
  Module[{s, sig, sig$2},start[];

    ChangeVar[sig$2, xor[xor[append[indexed[e, downto[5, 0]], indexed[e, downto[31,
6]]], append[indexed[e, downto[10, 0]], indexed[e, downto[31, 11]]]], append[indexed[e,
downto[24, 0]], indexed[e, downto[31, 25]]]]];

    ChangeVar[sig, xor[xor[append[indexed[a, downto[1, 0]], indexed[a, downto[31, 2]]],
append[indexed[a, downto[12, 0]], indexed[a, downto[31, 13]]]], append[indexed[a,
downto[21, 0]], indexed[a, downto[31, 22]]]]];

    ChangeVar[s, ch1 + h + ki32 + maj1 + wi32 + xor[xor[append[indexed[a, downto[1, 0]],
indexed[a, downto[31, 2]]], append[indexed[a, downto[12, 0]], indexed[a, downto[31,
13]]]], append[indexed[a, downto[21, 0]], indexed[a, downto[31, 22]]]] +
xor[xor[append[indexed[e, downto[5, 0]], indexed[e, downto[31, 6]]], append[indexed[e,
downto[10, 0]], indexed[e, downto[31, 11]]]], append[indexed[e, downto[24, 0]],
indexed[e, downto[31, 25]]]]];

    ChangeVar[return, ch1 + h + ki32 + maj1 + wi32 + xor[xor[append[indexed[a, downto[1,
0]], indexed[a, downto[31, 2]]], append[indexed[a, downto[12, 0]], indexed[a, downto[31,
13]]]], append[indexed[a, downto[21, 0]], indexed[a, downto[31, 22]]]] +
xor[xor[append[indexed[e, downto[5, 0]], indexed[e, downto[31, 6]]], append[indexed[e,
downto[10, 0]], indexed[e, downto[31, 11]]]], append[indexed[e, downto[24, 0]],
indexed[e, downto[31, 25]]]]]
];

Clear[shift];
SetAttributes[shift,HoldAll];
shift[x_]:=
  Module[{y},start[];

    ChangeVar[y, append[indexed[x, downto[30, 0]], indexed[x, 31]]];

    ChangeVar[return, append[indexed[x, downto[30, 0]], indexed[x, 31]]]
];

Clear[ajout2];
SetAttributes[ajout2,HoldAll];
ajout2[x_, y_]:=
  Module[{z},start[];

    ChangeVar[z, x + y];

    ChangeVar[return, x + y]
];

Clear[adressage2];
SetAttributes[adressage2,HoldAll];
adressage2[count_, t_]:=
  Module[{x},start[];

    ChangeVar[x, Ife[equal[t, 18], count, Ife[equal[t, 19], 2 + count, Ife[equal[t, 20],
8 + count, Ife[equal[t, 21], 13 + count, count]]]]];

    ChangeVar[return, Ife[equal[t, 18], count, Ife[equal[t, 19], 2 + count, Ife[equal[t,
20], 8 + count, Ife[equal[t, 21], 13 + count, count]]]]]
];
```

8 Bibliographie

[ABC05] G. Al Sammane, D. Borrione and R. Chavallier, *Verification of behavioral descriptions by combining symbolic simulation and automatic reasoning*, Proceedings of the 15th ACM Great Lakes symposium on VLSI, Chicago, Illinois, USA, Pages: 260 – 263, 2005, ISBN:1-59593-057-4.

[ABO98] R. Airiau, J.M. Bergé, V. Olive, J. Rouillard, *VHDL, langage, modélisation, synthèse,* 2ème édition, Presses Polytechniques et Universitaires Romandes, 1998, Collection informatique, 2 ème éd. *ISBN : 2880743613*.

[Acc04a] Accellera. *Property Specification Language Reference Manual Version 1.1*, 2004.

[Acc04b] Accellera, *SystemVerilog 3.1a Language Reference Manual,* 2004.

[AlT03] G. Al Sammane, D. Toma, *Specification of VHDL List Intermediate Format,* Rapport technique TIMA, 2003.

[Als04] G. Al Sammane*, Specification of the VHDL subset supported by TheoSim,* Rapport technique TIMA, 2004.

[ARM] ARM. AMBA *Specification, rev 2.0 edition*. ARM IHI0011A.

[AST04] G. Al Sammane, J. Schmaltz, D. Toma, P. Ostier and D. Borrione, *TheoSim: Combining Symbolic Simulation and Theorem Proving for Hardware Verification*, SBCCI 2004, Porto de Galinhas, Pernambuco, Brazil.

[ATS03] G. AL SAMMANE, D. TOMA, J. SCMALTZ, P. OSTIER, and D. BORRIONE, *Constrained Symbolic simulation with Mathematica and ACL2* , CHARME 2003, L'Aquila, Italy, LNCS 2860, pp. 150-157, Sprenger-Verlag,2003.

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers*, Addison Wesley, January 1, 1986, ISBN: 0201100886.

[BBC87] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Shefier. COSMOS: *A compiled simulator for MOS circuits*. Proceedings of 24th Design Automation Conference, pages 9-16, June 1987.

[BBS91] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, *Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation*, Proceedings of 28th Design Automation Conference, June, 1991, pp. 397-402.

[BCC99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. *Symbolic model checking without BDDs*. In Proc. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 1579, pages 193–207. Springer, 1999.

[BCM92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, *Symbolic Model Checking: 10^{20} states and beyond, Information and Computation,* vol. 98, no. 2, June 1992, pp. 142-70.

[BeH99] J. Bergmann and M. *Horowitz. Improving coverage analysis and test generation for large designs*. Proceedings of the International Conference on Computer Aided Design, pages 580-583, November 1999.

[BeO02] V. Bertacco and K. Olukotun, *Efficient State Representation for Symbolic Simulation,* Proceedings of 39th Design Automation Conference, New Orleans, 2002, pp. 99-104.

[Ber03] J. Bergeron. Writing Testbenches: *Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd edition, 2003.

[BeW93] T. Becker, and V. Weispfenning, *Gröbner Bases: A Computational Approach to Commutative Algebra.* New York: Springer-Verlag, pp. 213-214, 1993.

[BGM01] D. Borrione, P. Georgelin et V. Moraes Rodrigues, *Symbolic Simulation and Verification of VHDL with ACL2*. In P.J. Ashenden, J.P. Mermet and R. Seepold, editors, System-on-chip Methodologies and Design Languages, pages 59-70. Kluwer Academic Publisher, 2001.

[BLM01] P. Bjesse, T. Leonard, and A. Mokkedem. *Finding bugs in an Alpha microprocessor using satisfiability solvers*. In Computer-Aided Verification, Lecture Notes in Computer Science 2102. Springer-Verlag, pages 454–464, 2001.

[BLO05] D. Borrione, M. Liu, P. Ostier, L. Fesquet, *PSL-based online monitoring of digital systems,* Accepted at Forum on specification & Design Languages (FDL'05), Lausanne, Switzerland, September 27-30, 2005.

[BLS02] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. *Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and*

uninterpreted functions. In Computer-Aided Verification (CAV'02), LNCS 2404, pages 78 - 92, 2002.

[BoM97] R. S. Boyer and J S. Moore. *A computational Logic Handbook, Second Edition*. Academic Press, New York, 1997.

[BRR87] Z. Barzilai, J. L. Carter, B. K. Rosen, and J. D. Rutledge. *HSS - a high speed simulator.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pages 601- 617, July 1987.

[Bry86] R. E. Bryant. *Graph-based algorithms for Boolean function manipulation*. IEEE Transactions on Computers, 35(8): pp. 677-691, August 1986.

[Bry92] R. E. Bryant. *Symbolic Boolean manipulation with ordered binary decision diagrams*. ACM Computing Surveys, 24(3): pages 293-318, September 1992.

[Buc76] B.Buchberger, "*Theoretical Basis for the Reduction of Polynomials to Canonical Forms*." SIGSAM Bull. 39, 19-24, Aug. 1976.

[Buc01] B. Buchberger, Theorema: *Theorem Proving for the Masses Using Mathematica,* The Mathematica Journal Volume 8, Issue 2, 2001.

[BuD94] J. Burch and D. Dill, *Automatic verification of pipelined microprocessors control*. In CAV'94: Computer Aided Verification. Springer, LNCS 818, 1994, pp 68-80.

[CaM03] D. CACHERA et K. MORIN-ALLORY : Verification of control properties in the polyhedral model. In Proceedings of Memocode 2003, IEEE, Mont-St-Michel, France, jun 2003.

[CBM90] O. Coudert, C. Berthet and J. C. Madre, *Verification of synchronous sequential machines based on symbolic execution*, Proceedings of the international workshop on Automatic verification methods for finite state systems, Grenoble, France, 1990, Springer-Verlag New York, Inc. ISBN 0-387-52148-8.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla, *Automatic Verification of Finite State Concurrent System Using Temporal Logic,* ACM Transactions on Programming Languages and Systems,vol. 8(2), 1986, pp. 244-263.

[CGP00] E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking*, The MIT Press (January 7, 2000), ISBN: 0262032708.

[ClZ93] E. M.Clarke and X. Zhao, *Analytica - A theorem prover for Mathematica*. The Mathematica Journal Volume 3, Issue 1, 1993, pages 56 – 71.

[Coh99] Ben Cohen, *VHDL Coding Styles and Methodologies*, Springer; 2 edition 1999, ISBN: 0792384741.

[CW81] C.A. Cole and S. Wolfram, *SMP: A Symbolic Manipulation Program* (1981), in SYMSAC '81*:* Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation (Snowbird, Utah, August 5-7, 1981) ed. P. Wang, 20-22.

[Dar79] J. Darringer. *The application of program verification techniques to hardware verification*. Proceedings of 16th Design Automation Conference, pages 375–381, 1979.

[Deh96] D. Deharbe. *Vérification formelle de propriétés temporelles : étude et application au langage VHDL.* Thèse de doctorat, Université Joseph Fourier, Grenoble I, 1996.

[Dev97] C. J. DeVane. *Efficient circuit partitioning to extend cycle simulation beyond synchronous circuits*. Proceedings of the International Conference on Computer Aided Design, pages 154 -161, nov 1997.

[DLL62] M. Davis, G. Longeman, and D. Loveland, *A Machine Program for Theorem Proving,* Communications of the ACM, vol. 5, pp. 394–397, 1962.

[Dum03] E. Dumitrescu, *Construction de modèles réduits et vérification symbolique de circuits industriels décrits au niveau RTL*, Thèse de Doctorat UJF, 2003, ISBN: 2-84813-021-0.

[DuO98] E. Dumitrescu and P. Ostier, *Identification of non-redundant memorizing elements in VHDL synchronous designs for formal verification tools*. In Proceeding of the Fifth International Workshop on Symbolic Methods and Applications in Circuit Design (SMACD), pages 233-243, 1998.

[Dus99] Julia Dushina. *Vérification Formelle des Résultats de la Synthèse de Haut Niveau*. PhD thesis, Université Joseph Fourier, Grenoble, 1999.

[EGK95] B. L. Evans, S. X. Gu, A. Kalavade, and E. A. Lee. *Symbolic Computation in System Simulation and Design*. In Proc. of SPIE Int. Sym. on Advanced Signal Processing Algorithms, Architectures, and Implememtations, July 1995. San Diego, CA, pp. 396-407. Downloadable at :

http://ptolemy.eecs.berkeley.edu/publications/papers/95/spie95symbcom p

[GBO02] P. Georgelin, D. Borrione, and P. Ostier. *A framework for VHDL combining theorem proving and symbolic simulation*. In Proc. 3rd Intl. Workshop on the ACL2 Theorem Prover and its Application, pages 1–15, April 2002.

[Geo01] Ph. Georgelin, *Vérification formelle de systèmes digitaux synchrones, basée sur la simulation symbolique,* Thèse de Doctorat UJF, Spécialité Informatique, 18 Oct 2001, ISBN: 2-913329

[Gra97] J. W. Gray, *Mastering Mathematica: Programming Methods and Applications*, *Second Edition*, 1997, Academic Press, ISBN: 0122961056.

[Gre98] D. Greve. *Symbolic simulation of the JEM1 microprocessor*. In G. Gopalakrishnan and P. Windley, editors, Formal Methods in Computer-Aided Design (FMCAD '98), LNCS 1522, pages 321–333, Palo Alto, CA, 1998. Springer-Verlag.

[GZA02] M. Ganai, L. Zhang, P. Ashar, and A. Gupta, *Combining Strengths of Circuit-based and CNF-based Algorithms for a High Performance SAT Solver,* in Proceedings of the 39th Design Automation Conference, 2002.

[Han88] C. Hansen. *Hardware logic simulation by compilation.* Proceedings of 25th Design Automation Conference, pages 712-716, June 1988.

[HKM01] F. I. Haque, K. A. Khan, and J. Michelson. *The Art of Verification with Vera,* Verification Central, 2001. ISBN: 0-9711994-0-X.

[HMN01] Y. Hollander, M. Morley, and A. Noy. *The e language: A fresh separation of concerns*. In Technology of Object-Oriented Languages and Systems, volume TOOLS-38, pages 41 - 50, March 2001.

[HSH00] P. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, Jerry Taylor, and Jiang Long. *Smart simulation using collaborative formal and simulation engines.* Proceedings of the International Conference on Computer Aided Design, pages 120-126, November 2000

[Hun89] W. A. Hunt. *Microprocessor design verification*. Journal of Automated Reasoning, 5(4): pages 429–460, 1989.

[IEEE99] IEEE. *1076.6-1999 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*, 1999.

[IEEE01] *IEEE Standard Verilog Hardware Description Language*, NY, IEEE Std 1364-2001

[IEEE04] IEEE. *1076.6-2004 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis,* 2004.

[Jer02] JERRAYA Ahmed-Amine*, Conception de haut niveau des systèmes monopuces (Traité EGEM, Série électronique et micro-électronique),* Hermes, 2002ISBN : 2-7462-0433-9

[JHM01] I. Janiszewski, B. Hoppe, and H. Meuth. *VHDL-based design and design methodology for reusable high performance direct digital frequency synthesizers*. In Proc. IEEE-ACM of the 38th Design Automation Conference, pages 573–578, 2001.

[KaN96] Michael Kantrowitz and Lisa M. Noack. *I'm done simulating; now what? verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor*. Proceedings of 33rd Design Automation Conference, pages 325 - 330, June 1996.

[KMM00a] M. Kaufmann, P. Manolios, and J. S. Moore. *ACL2 Computer Aided Reasoning: An Approach (Vol 1.) KluwerAcademic Press*, 2000, ISBN 0-7923-7744-3).

[KMM00b] M. Kaufmann, P. Manolios, and J. S. Moore. *ACL2 Computer Aided Reasoning: ACL2 Case Studies (Vol 2.).KluwerAcademic Press*, 2000, (ISBN 0-7923-7849-0).

[KMW67] R.M. KARP, R.E. MILLER et S. WINOGRAD : *The organization of computations for uniform recurrence equations*. Journal of the Association for Computing Machinery, 14(3): pages 563–590, juillet 1967.

[KnB70] D. E. Knuth and P. B. Bendix "*Simple Word Problems in Universal Algebra*." In Computational Problems in Abstract Algebra (Proc. Conf., Oxford, 1967). Pergamon Press, pp. 263-297, 1970.

[KND02] F. Karim, A. Nguyen and S. Dey *: An Interconnect Architecture For Networking Systems On Chip*. IEEE Micro (Sept-Oct 2002) pp. 36-45.

[KOW01] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai. *A framework for object oriented hardware specification, verification and synthesis*. Proceedings of 38th Design Automation Conference, pages 413 - 418, June 2001.

[LMB92] J. Levine, T. Mason, D. Brown, *Lex & yacc*, O'Reilly, *edition 2 (October 2, 1992) , ISBN: 1565920007*

[LMU02] O. Lachish, E. Marcus, S. Ur, and A. Ziv. *Hole analysis for functional coverage data*. Proceedings of 39th Design Automation Conference, pages 807 - 812, June 2002.

[LN03] D. LaBouve, D. Nicklas, *Complex memories: the art of mixing traditional simulation with innovative verification solutions,* EE Times,2003, electronic edition : http://www.eetimes.com/story/OEG20030428S0105.

[Mae96] R. Maeder, *The Mathematica Programmer II*, 1996, Academic Press, ISBN: 0124649920.

[Mae00] R. Maeder. *Computer Science with Mathematica*. Cambridge University Press, 2000. ISBN 0-521-63172-6.

[McC60] J. McCarthy. *Recursive functions of symbolic expressions and their computation by machine*, part 1. Comm. A.C.M., 3: pp. 184-195, 1960.

[McC63] J. McCarthy. *A Basis for a Mathematical Theory of Computation*. In P. Braffort and D. Hirschberg, editors, Computer Programming and Formal Systems, pages 33-70. North-Holland, Amsterdam, 1963.

[McM93] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, *1993. ISBN : 0792393805*

[McM03] K.L. McMillan, *Interpolation and SAT-based Model Checking*, Computer Aided Verification (CAV03), Springer, LNCS, Vol 2404, pages 27-39, July 2003.

[MMZ01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, *Chaff: Engineering an Efficient SAT Solver*, Proceedings of 38th Design Automation Conference, 2001. pp 530-535.

[Moo92] J S. Moore. *Introduction to the OBDD Algorithm for the ATP community*. Technical report, Computational Logic Inc., 1992.

[Moo98] J S. Moore, *Symbolic Simulation: An ACL2 Approach*, Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98), Springer-Verlag LNCS 1522, pp. 334-350, November, 1998.

[MSS99] J. P. Marques-Silva and K. A. Sakallah, GRASP: *A Search Algorithm for Propositional Satisfiability,* IEEE Transactions on Computers, vol. 48, pp. 506–521, 1999.

[OSR92] S. Owre, N. Shankar, and J. M. Rushby. PVS: *A prototype verification system.* In Proc. 11th International Conference on Automated Deduction (CADE), volume 607 of LNCS, pages 748-752. Springer, 1992.

[PaB97] M. Pandey, R. E. Bryant: *Exploiting Symmetry When Verifying Transitor-Level Circuits by Symbolic Trajectory Evaluation.* CAV 1997 LNCS vol. 1254 Springer-Verlag: 244-255

[PKZ98] E. Petru, K. Krzysztof, P. Zebo, *System Synthesis with VHDL*, Springer 1998, ISBN: 0-7923-8082-7.

[PZ00] J.C. van de Pol, H. Zantema, *Binary decision diagrams by shared rewriting*, Research Report, 2000, SEN-R0001, ISSN 1386-369X. http://db.cwi.nl/rapporten/.

[RSS96] H. Rueß, N. Shankar, M. Srivas, *Modular Verification of SRT Division*, Computer-Aided Verification, CAV'96, Springer-Verlag, LNCS, Vol :1102, pp 123-134.

[SaH02] J. Sawada, and W.A. Hunt, *Verification of FM9801: Out-of-Order Processor with Speculative Execution and Exceptions That May Execute Self-Modifying Code,* Journal on Formal Methods in System Design, Vol. 20, No. 2 (March 2002).

[SDH00] Kanna Shimizu, David L. Dill and Alan J. Hu. *Monitor-Based Formal Specification of PCI.* Proceedings of the Third International Conference of Formal Methods in Computer-Aided Design, Springer, LNCS vol: 1954, pages 335-353, November, 2000.

[Smu95] R. M. Smullyan, *First-Order Logic*, Dover Publications, 1995, ISBN: 0486683702.

[TBA04] D. TOMA , D. BORRIONE , G. AL SAMMANE, *Combining several paradigms for circuit validation and verification,* Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04), Marseille, France, LNCS vol: 3362, Springer 2005, ISBN 3-540-24287-2. pages 229-249.

[TPB04] D. TOMA , A. PEREZ , D. BORRIONE , E. BERGERET, *Design of a proven correct SHA circuit,* International Conference on Electrical, Electronic and Computer Engineering (ICEEC-04), Cairo, Egypt, September 5-7 , 2004. pages 31-34.

[Tra95] A. Tarski, *Introduction to Logic and to the Methodology of Deductive Sciences,* Dover Publications, 1995, ISBN: 048628462X

[VaQ88] V. VAN DONGEN et P. QUINTON, *Uniformization of linear reccurence equations: a step towards to the automatic synthesis of systolic arrays*. In International Conference on Systolic Arrays, San Diego (EU), Mai 1988.

[WHP87] L. Wang, N. E. Hoover, E. H. Porter, and J. J. Zasio. SSIM*: A software levelized compiled-code simulator*. Proceedings of 24th Design Automation Conference, pages 2-8, June 1987.

[WiD00] C. Wilson and D. L. Dill. Reliable verification using symbolic simulation with scalar values. Proceedings of 37th Design Automation Conference, pages 124-129, June 2000.

[Wol85] S. Wolfram, *Symbolic Mathematical Computation*, Communications of the ACM Vol: 28 (1985), pages 390-394.

[Wol88] S. Wolfram, Mathematica: *A System for Doing Mathematics by Computer* (Redwood City, CA: Addison-Wesley Publishing Company, Inc., 1988).

[Wol02] S. Wolfram, *A New Kind of Science*, 2002, Wolfram Media, ISBN: 1-57955-008-8

[Wol03] S. Wolfram, *The Mathematica Book, Fifth Edition*, Wolfram Media ISBN 1-57955-022-3.

[Zha97] H. Zhang, SATO: *An efficient propositional prover,* in Proceedings of International Conference on Automated Deduction, vol. 1249, LNAI, Springer, 1997, pp. 272–275.

Sites web

[UCLID] UCLID. *Rapport technique*, http://www.cs.cmu.edu/~uclid

[WWW01] http://www.alphaworks.ibm.com/tech/FoCs

[WWW02] http://www.cadence.com/

[WWW03] http://www.synopsys.com/

[Wei05] E. W. Weisstein, A Wolfram Web Resource:
Buchberger's Algorithm, From MathWorld --
http://mathworld.wolfram.com/BuchbergersAlgorithm.html,
Knuth-Bendix Completion Algorithm,
http://mathworld.wolfram.com/Knuth-BendixCompletionAlgorithm.html,
Gröbner Basis,
http://mathworld.wolfram.com/GroebnerBasis.html.

RESUMÉ

Ce travail de thèse présente une méthode originale pour la simulation symbolique des circuits décrits au niveau algorithmique. Tout d'abord, la description VHDL du circuit est modélisée sous le forme d'un ensemble d'équations récurrentes (SER) qui décrivent l'état du système à un instant donné en fonction des états précédents. Après une extraction automatique du SER du circuit, l'algorithme de simulation VHDL est exécuté pendant un nombre fixe de cycles déterminé par le concepteur. Pendant la simulation, un scénario de test et une simplification par règles de substitution sont appliqués pour obtenir les expressions symboliques ou numériques de chaque objet du circuit (registre, signal ou port de sortie). Trois modes de test (raisonnement, exécution et mixte) sont définis et expliqués en se basant sur la distinction entre la partie opérative et la partie contrôle de circuit. Le simulateur symbolique et le compilateur sont implémentés avec l'aide du système Mathematica.

Une méthodologie de vérification autour de la simulation symbolique avec SER est proposée. Plusieurs paradigmes de vérification (la correspondance de forme, la démonstration de théorèmes et SAT) sont employés sur les résultats de la simulation symbolique pour valider ou prouver les propriétés du circuit. La méthodologie est montrée sur deux circuits de taille réelle (un filtre numérique et une mémoire) et sur de nombreux cas académiques.

MOTS-CLÉS

Simulation symbolique, équations récurrentes, simulation de VHDL, vérification formelle, démonstration automatique de théorèmes, observateur VHDL.

TITLE

SYMBOLIC SIMULATION OF CIRCUITS DESCRIBED AT THE ALGORITHMIC LEVEL

ABSTRACT

This PhD thesis presents a new symbolic simulation method for circuits described at algorithmic level. First the VHDL description is modeled as a set of recurrence equations (SRE) that describe the state of the system at a given time as a function of previous states. After an automatic extraction of the model SRE, the VHDL simulation algorithm is applied for a fixed number of simulation cycles given by the designer. During the simulation, a test scenario and a simplification via substitution rules are applied to compute the symbolic or the numeric expression of each object in the design (register, signal or output port). Three test modes are defined and explained: tracking, reasoning and mixed. They are based on separation of the operative part from the control part of the circuit. The symbolic simulator and the VHDL to SRE compiler are implemented using Mathematica.

A verification methodology around the SRE symbolic simulation is proposed. Multiple verification paradigms (pattern matching, theorem proving and SAT) are applied to the symbolic simulation results to validate or to prove the properties of the circuit. The methodology is illustrated on two real size circuits (a RAM memory and digital filter) and on several academic examples.

KEYWRODS

Symbolic Simulation, recurrence equations, VHDL simulation, formal verification, theorem proving, VHDL monitors.

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN : 2-84813-069-5 (version brochée)

ISBNE : 2-84813-070-9 (version électronique)