Université de Montréal

Construction des machines à état abstrait à partir des modèles VHDL

par

Koty Dominique ANON

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures

en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)

en informatique

Novembre 1997

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé:

Construction des machines à état abstrait à partir des modèles VHDL

présenté par:

Koty Dominique ANON

a été évalué par un jury composé des personnes suivantes:

Mostapha Aboulhamid	président-rapporteur
Eduard Cerny	directeur de recherche
Michel Boyer	membre du jury

Mémoire accepté le 22.12.1997

Sommaire

La vérification formelle est un complément à la simulation, pour trouver les erreurs dans la conception des circuits numériques, avant l'étape de la fabrication, afin de minimiser les coûts de correction qui tendent à croître, avec la densité d'intégration de plus en plus élevée.

L'utilisation des graphes de décision multi-choix (Multiway Decision Graph - MDG) pour modéliser les circuits numériques a permis d'obtenir des représentations compactes qui ont amélioré les performances de la vérification formelle.
Le présent mémoire décrit la construction d'un traducteur, qui fait un pont entre le langage VHDL et un système de vérification basé sur MDG. Ce traducteur prend, en entrée les modèles VHDL de machines à état, et produit en sortie des machines à état abstrait basées sur MDG.

Les deux modèles de description de circuit numérique, supportent des modèles structuraux et comportementaux. VHDL supporte les descriptions hiérarchiques, et comporte un nombre important d'opérateurs qui n'ont pas d'équivalent dans le modèle MDG. Les modèles comportementaux de VHDL sont définis par des processus décrits essentiellement par des énoncés séquentiels de contrôle, d'affectation des signaux et variables, et de synchronisation. Chaque objet de VHDL (signal ou variable) est caractérisé par un type ou un sous-type, défini par un ensemble de propriétés, qui dénote les valeurs que cet objet peut prendre et les opérations qui lui sont applicables.

Les modèles abstraits de description de circuit basés sur MDG, utilisent une logique multi-sortée du premier ordre, qui d'une part introduit une relation d'ordre entre les symboles utilisés dans la description du circuit, et divisent les signaux du circuit en fonction de leur sort. Les signaux de données sont représentés par des variables abstraites (de sort abstrait), leurs valeurs sont non-interprétées, et les signaux de contrôle sont représentés par des variables concrètes (de sort concret) définies par leur ensemble de valeurs. Les opérations sur les données sont représentées par des symboles fonctionnels non-interprétées, qui sont soit des opérateurs croisés soit des symboles fonctionnels abstraits. Les descriptions comportementales en MDG sont définies par

fonctionnels abstraits. Les descriptions comportementales en MDG sont définies par des relations de transition et de sortie implantées par des composants représentés sous forme tabulaire.

Pour palier le manque de format spécifique de description de machine à état dans VHDL, nous avons défini un modèle de spécification, en fixant les modes d'activation des processus, et le mode de synchronisation de la machine à état, et en choisissant des formes spécifiques de certains énoncés. Cela conduit à la définition de différents styles de description de machines à état en VHDL. La spécification du sort d'un objet VHDL est, soit effectuée par un attribut utilisateur spécifié pour le type de cet objet, soit implicite pour les types d'énumération et les types entiers sur des ensembles finis.

Les syntaxes des descriptions comportementales dans les deux modèles étant totalement différentes, nous utilisons une sémantique opérationnelle, pour décrire le comportement des modèles VHDL de machine à état. Cette méthode de définition sémantique associe à une spécification VHDL de machine à état, une machine théorique (un automate) dont l'exécution passe par différents états dont certains (les états de synchronisation) correspondent aux états de la machine VHDL. Cette sémantique base la description du comportement sur la sémantique de chaque énoncé et la syntaxe du langage VHDL.

Les tâches de construction d'un traducteur étant composées globalement de l'analyse, la synthèse et de la génération de code, nous utilisons un environnement de compilation (LEDA VHDL SYSTEM) pour réaliser les tâches de l'analyse. Cette analyse produit un arbre de syntaxe qui peut être étendu en utilisant l'outil GRAPHGEN de LEDA pour produire un flot de contrôle et de données. Ce graphe décrivant l'exécution du code de la machine à état VHDL, est une implantation de la machine théorique issue de la sémantique opérationnelle.

La synthèse du code cible consiste essentiellement à aplatir éventuellement les descriptions structurales hiérarchiques, et à partir du graphe de flot de contrôle et de données de chaque processus, produire les relations de transition et de sortie de la machine à état abstrait, interpréter les expressions et opérateurs concrets, créer des symboles fonctionnels pour les opérateurs croisés et abstraits, et définir les spécifications algébriques et les déclarations de signaux dans les modèles MDG en fonction des

déclarations de signaux et variables, des fonctions, des types et les spécifications de sorts. La dernière étape de la synthèse réalise un ordonnancement des signaux et symboles fonctionnels de la structure aplatie (sans hiérarchie).

La génération de code, produit la spécification algébrique et le code de chaque élément constitutif de la description de la machine à état abstrait, conformément à la syntaxe du langage cible.

Remerciements

Je voudrais remercier sincèrement Professeur Eduard Cerny, pour m'avoir accepté son laboratoire, et pour l'encadrement minutieux dont j'ai fait l'objet pendant le déroulement du projet, et la rédaction de ce mémoire.

Mes remerciements vont également à l'endroit de Professeur Xiayu Song, pour sa grande disponibilité et son attention particulière.

Je saisis l'occasion pour remercier très profondément l'ensemble de tous les professeurs du DIRO, qui ont participé activement à ma formation.

Ma reconnaissance va à l'endroit des autres pionniers des MDGs, Dr. Francisco Corella et Dr. Michel Langevin, dont l'ingéniosité a conduit la création du projet sur lequel j'ai travaillé.

Je voudrais remercier Professeur Sofiene Tahar, Dr. Zijan Zhou et Dr. Otmane Ait Mohamed pour leur encadrement et leur support attentif.

Je voudrais exprimer toute ma gratitude aux autres membres du groupe MDG-VERIF, à savoir Ying Xu, Nancy Boulerice, Dan Voicu et Boubkar Elmalki pour les discussions fructueuses.

Je remercie Jindrich Zejda et Samir Boubezari pour leur assistance et leur disponibilité, respectivement sur le système informatique du DIRO et le logiciel LEDA.

Je voudrais enfin remercier Denise St-Michel pour son support administratif, et le GRIAO (Groupe Interuniversitaire en Architecture des Ordinateurs et VLSI) pour tout le soutien matériel.

Table des Matières

Table des Matières ..... **v**

Liste des figures ..... **xi**

Liste des tables ..... **xiv**

Chapitre I **1**

Introduction **1**

Chapitre II **4**

Les méthodes de vérification formelle de matériels **4**

2.1 La vérification formelle dans le processus de la synthèse 4

2.1.1 La pertinence de la vérification formelle 4

2.1.2 La synthèse des circuits numériques 5

2.1.3 La vérification dans le flot de synthèse du design 7

2.1.3.1 La validation du design 7

2.1.3.2 La vérification des résultats d'une synthèse comportementale 7

2.1.3.3 La vérification logique 8

2.1.3.4 La vérification de plan de masque lithographique 8

2.2 Les méthodes formelles de vérification 8

2.2.1 Les Prouveurs de théorèmes 9

2.2.2 Vérification de l'équivalence de deux machines à état 11

2.2.3 La vérification symbolique de modèle 12

2.2.4 Les tendances: intégration de différentes techniques de preuves 14

2.3 Le système de vérification formelle MDG 15

2.3.1 Les symboles de MDG 16

2.3.2 Les graphes de décisions multi-choix 16

2.3.3 Formule Directe 17

2.3.4 Machine à état abstrait 18

2.3.5 Procédure de vérification avec MDG 18

Chapitre III 20

Modèle de spécification VHDL de machines à état 20

3.1 Les modèles de description des circuits en VHDL 20

3.1.1 Entité de design 20

3.1.2 Le modèle comportemental 21

3.1.2.1 Les processus VHDL 21

3.1.3 Le modèle structural 22

3.2 Le sous-ensemble VHDL 24

3.2.1 L'activation et la suspension des processus 24

3.2.1.1 Activation et suspension des processus combinatoires 25

3.2.1.2 Activation et suspension des processus séquentiels 26

3.2.2 La synchronisation de la machine avec l'horloge 27

3.2.3 La remise à l'état initial 28

3.2.4 Le délai d'assignation 29

3.2.5 La spécification des sort des données 30

3.2.6 Spécification de boîtes noires 30

3.3 Exemples de styles de description des machines à état 31

3.3.1 Modélisation de machines à état implicite par un seul processus ... 32

3.3.2 Modélisation de machine à état explicite 33

3.3.3 Modélisation de machines à état par un ensemble de processus 35

Chapitre IV 39

Le langage de description de matériels MDG-HDL 39

4.1 Fondement théorique du langage 39
4.1.1 Ordonnancement des symboles dans les MDGs 40
4.1.2 Partitionnement des relations de transition et de sortie 41
4.1.3 Interprétation des symboles des machines à état abstrait 42
4.2 Description abstraite de machines à état 43
4.2.1 Spécification informelle et algorithme de calcul du GCD 43
4.2.2 Machine à état fini de GCD 44
4.2.3 Description abstraite de la machine de GCD 45
4.2.3.1 Interprétation des symboles abstrait de GCD 45
4.2.3.2 Description abstraite des ensembles 46
4.2.3.3 L'ensemble des états initiaux 47
4.2.3.4 La relation de transition 47
4.2.3.5 La relation de sortie 49
4.3 Description de circuits numériques 50
4.3.1 Exemple: spécification MDG-HDL de la machine à état GCD 51
4.3.2 Section de la spécification algébrique de la machine à état GCD ... 51
4.3.3 Spécification de la machine à état abstrait 51
4.3.4 Liste des symboles ordonnés 54
Chapitre V 55
Sémantique opérationnelle des machines à état VHDL 55
5.1 Introduction 55
5.2 Modélisation du langage source 56
5.2.1 Évaluation d'expressions concrètes 56
5.2.2 Exécution des énoncés 57
5.2.3 Représentation de l'exécution des énoncés 58

5.2.4 Exécution d'un modèle 59
5.3 Automates pour énoncés séquentiels 60
5.3.1 Enoncé wait 60
5.3.2 Énoncé d'assignation de signal 62
5.3.3 Enoncés conditionnels 63
5.3.4 Séquence d'énoncés séquentiels 64
5.4 Automate pour un processus unique 66
5.4.1 Construction de l'automate 66
5.4.2 Exemple d'automate d'un processus 68
5.5 Automate pour un modèle comportemental VHDL 71
5.6 Dérivation du comportement d'une machine à état à partir de l'automate d'exécution d'un modèle VHDL 73
Chapitre VI 76
Environnement de la traduction 76
6.1 Les phases de la traduction 76
6.1.1 L'analyse 77
6.1.2 La synthèse du langage cible 78
6.2 Les outils de la traduction 78
6.3 Le système VHDL de LEDA 80
6.3.1 Le format VIF 80
6.3.2 L'interface procédurale de LEDA 81
6.3.3 Le générateur de graphes de flot de contrôle et de données GRAPHGEN 81
6.3.3.1 Les noeuds du graphe de flot de contrôle 82
6.3.3.2 Les noeuds du graphe de flot de données 85

6.3.3.3 Partitionnement des graphes de flot de contrôle 86
6.4 Contraintes de traduction imposées par MDG-HDL 88
6.4.1 Les sorts et leur spécification dans le code VHDL 88
6.4.2 Transformations induites par la partition des types de données 88
6.4.3 Représentation plate de MDG-HDL 89
6.4.4 Opérateur implicite de MDG-HDL 89
6.5 Caractéristiques spécifiques de VHDL 89
6.5.1 Machines à état décrites sous forme comportementale 89
6.5.2 Correspondance entre états des machines et énoncés wait 89
6.5.3 Traitement des séquences de commandes 90
Chapitre VII 91
Traduction des formalismes de VHDL 91
7.1 La traduction des déclarations 91
7.1.1 Les déclarations de type 91
7.1.1.1 Les Types d'énumération 92
7.1.1.2 La déclaration de sous-type 92
7.1.1.3 Les autres types 92
7.1.2 Les déclaration des constantes 93
7.1.3 Les déclarations des signaux et variables 93
7.1.4 Les déclaration des fonctions 94
7.2 La traduction des expressions 94
7.3 La traduction des modèles comportementaux 95
7.3.1 Les étapes de traduction des modèles comportementaux 95
7.3.2 La traduction de l'état initial de la machine 96
7.3.2.1 Les variables d'état concrètes 96

7.3.2.2 Les variables d'état abstraites 97

7.3.3 La traduction des relations de transition et de sortie 97

7.4 La traduction des modèles structuraux 103

7.5 Les autres aspects de la traduction 112

7.5.1 L'ordonnancement des symboles 112

7.5.2 Le contrôle sémantique 113

7.6 Résultats expérimentaux 114

7.6.1 Implantation 114

7.6.2 Les exemples 115

Chapitre VIII 116

Conclusions et perspectives 116

8.1 Conclusions 116

8.2 Perspectives 117

Bibliographie 119

Appendice A 122

Le sous-ensemble VHDL 122

Appendice B 140

Le manuel d'utilisateur du compilateur VHDL-MDG 140

Liste des figures

figure 1.1 Schéma typique d'une vérification de machine à état VHDL avec le système de vérification basé sur les MDGs 3

figure 2.1 Flot de synthèse de design.....6

figure 2.1 Structure de la preuve de matériels avec HOL.....10

figure 2.2 Procédure de vérification avec un vérificateur de modèle.....14

figure 2.3 Procédure de vérification d'invariants avec MDG.....19

figure 3.1 Description structurale d'un demi-additionneur.....23

figure 3.2 Exemple de processus décrivant la logique combinatoire.....25

figure 3.3 Formes de l'énoncé wait pour le codage des processus séquentiels..26

figure 3.4 Exemples de processus séquentiels utilisant les deux formes de wait retenues.....27

figure 3.5 Processus séquentiels avec condition de remise à l'état initial.....29

figure 3.6 La syntaxe de la spécification de la configuration des composants...31

figure 3.7 Une bascule D avec mise à 0 et 1 modélisée comme une machine à état implicite.....33

figure 3.8 Un exemple de modélisation de machine à état explicite par un seul processus.....35

figure 3.9 Machine à état implantée par deux processus, la fonction de transition est réalisée par le processus séquentiel.....37

figure 3.10 Machine à état implantée par deux processus, la fonction de transition est réalisée par le processus combinatoire.....38

figure 4.1 Algorithme de calcul du plus grand commun diviseur.....44

figure 4.2 Représentation graphique de la machine à état fini de GCD.....45

figure 4.3 Représentation MDG de l'alphabet d'entrée de la machine à état abstrait GCD....46

figure 4.4 Représentation MDG d'un état initial de la machine à état GCD....47

figure 4.5 Représentation MDG de la relation de transition de la machine à état GCD....49

figure 4.6 Représentation MDG de la relation de sortie de machine à état GCD50

figure 4.7 Section algébrique de la spécification de la machine à état GCD....51

figure 4.8 Spécification MDG-HDL de la machine à état GCD....53

figure 4.9 Liste des symboles ordonnés sur mesure de la description abstraite de la machine à état GCD....54

figure 5.1 Processus de simulation....60

figure 5.2 Automate de l'énoncé séquentiel "wait on S"....61

figure 5.3 Automate d'énoncé séquentiel....62

figure 5.4 Automate....63

figure 5.5 Automate d'énoncé séquentiel d'une séquence de deux énoncés....65

figure 5.6 Automate d'un énoncé processus....67

figure 5.7 Spécification d'une machine à état fini VHDL de type MEALY....69

figure 5.8 Automate d'énoncé de processus détaillé du processus sm de la figure 5.7....70

figure 5.9 Composition du processus noyau et des processus utilisateurs....72

figure 5.10 Représentation graphique de la machine à état dérivée à partir de l'automate de l'énoncé de processus de la figure 5.8....74

figure 6.1 Décomposition des tâches de la traduction....76

figure 6.2 Décomposition des tâches de l'analyse....77

figure 6.3 Organisation logique d'un compilateur....78

figure 6.4 Les phases de réalisation de notre compilateur.....79

figure 6.5 CFG du processus de la machine à état VHDL de la figure 5.7.....84

figure 6.6 Graphe de flot de données de l'affectation d'une constante.....85

figure 6.7 Partitions en blocs de base du graphe de flot de contrôle la figure 6.5.87

figure 7.1 Code VHDL d'une machine à état de type Mealy.....98

figure 7.2 Algorithme de construction de l'ensemble des environnements d'affection d'une variable v.....100

figure 7.3 Ensemble des environnements d'affectation de la variable z de l'exemple de la figure 7.1.....100

figure 7.4 La section algébrique de spécification de la machine à état abstrait correspondant à la description VHDL de figure 7.1.....101

figure 7.5 Machine à état abstrait du modèle VHDL de la figure 7.1.....102

figure 7.6 Algorithme de traduction d'une description structurale.....104

figure 7.7 Structure hiérarchique d'un additionneur.....104

figure 7.8 Description structurale d'un additionneur.....105

figure 7.9 Description structurale d'un demi-additionneur.....106

figure 7.10 Une description comportementale de porte ou.....107

figure 7.11 Une description comportementale de porte et.....108

figure 7.12 Exemple d'application de l'algorithme de traduction d'une description structurale à un additionneur.....109

figure 7.13 Section algébrique de la spécification machine à état abstrait correspondant à la description structurale de l'additionneur.....110

figure 7.14 Machine à état abstrait de la description structurale de l'additionneur.....112

Liste des tables

Table 6.1 Attributs du noeud architecture_body 81
Table 7.1 Chemins d'exécution de la machine à état de la figure 7.1 99
Table 7.2 Représentation tabulaire de la relation de sortie de z dérivée de la figure 7.3 101
Table 7.3 Exemples de traductions réalisées 115

Chapitre I

Introduction

Le terme traducteur désigne un processeur de langage, qui accepte une description dans un langage source comme entrée, et produit une description fonctionnellement équivalente dans un langage cible en sortie.

Le langage source de notre traducteur, VHDL, est un standard pour la description des circuits numériques. Il possède un grand nombre de formalismes permettant de décrire les circuits numériques à différents niveaux d'abstraction, et avec aussi bien des modèles comportementaux que des modèles structuraux. Les descriptions concernées dans le cadre du projet, sont effectuées au niveau transfert-registre (Register Transfer Level - RTL) où le séquencement des opérations est défini à chaque cycle d'horloge.

Le langage de description de matériel MDG-HDL, notre langage cible, supporte le modèle de description de matériels du système de vérification MDG. Une description de circuit de MDG-HDL est une interconnexion de composants dans une structure plate (sans hiérarchie). Les composants sont, soit prédéfinis dans la bibliothèque, soit des boîtes noires définies par leur interface avec l'environnement extérieur et des symboles fonctionnels non-interprétés, soit des composants dont le comportement est défini par une machine à état abstrait.

Les deux langages de description de matériels ont en commun la possibilité de décrire des modèles structuraux et comportementaux. Le trait caractéristique de VHDL est la description de modèles hiérarchiques. Quant à MDG-HDL, il divise les symboles en deux classes, les sorts concrets et les sorts abstraits, et réalise une description abstraite des circuits numériques, dont le comportement est défini à l'aide de relations de transition et de sortie sous forme de tables, et des symboles de fonction non-interprétés.

Notre traducteur extrait le comportement de la machine à état fini VHDL, et construit une machine à état abstrait en tenant compte du caractère abstrait ou concret des objets (signaux ou variables). L'allocation des ressources MDG-HDL est basée sur une hypothèse de ressources illimitées, et sur le caractère abstrait ou concret des opérateurs et des objets. L'extraction du comportement de la machine à état n'effectue aucune optimisation, de l'encodage des états des machines, et de la logique combinatoire des circuits séquentiels.

Une utilisation typique de notre traducteur pour vérifier des descriptions VHDL avec le système de vérification MDG, est représentée dans la figure 1.1. La première opération consiste à compiler le code source VHDL en utilisant le compilateur LEDA-VHDL. Cette compilation produit une représentation abstraite du langage source, sous la forme d'un arbre de syntaxe abstraite (Abstract Syntax Tree - AST). Puis les outils APEX et GRAPHGEN sont utilisés pour étendre le schéma de description. APEX regroupe par ajout de liens et noeuds supplémentaires, des informations difficiles à retrouver dans l'arbre de syntaxe initial. GRAPHGEN construit un graphe de flot de contrôle et de données pour chaque processus, qui définit tous les chemins d'exécution des processus de la spécification des machines à état VHDL. La troisième étape consiste à utiliser notre traducteur pour construire des machines à état abstrait (Abstract State Machine - ASM), à partir des graphes de flot de contrôle et de données issus du modèle VHDL. Au cours de l'étape finale, les vérifications basées sur MDG peuvent être exécutées.

L'organisation du mémoire est présentée comme suit: le chapitre 2 justifie la pertinence de la vérification formelle dans le contexte de la synthèse des circuits numériques, et présente quelques méthodes de vérification formelle. Le chapitre 3 introduit le langage de description de matériels (Hardware Description Language - HDL) VHDL, donne les caractéristiques du niveau d'abstraction utilisé, définit un modèle de spécification des machines à état en VHDL. Le chapitre 4 décrit le modèle de description de circuit supporté par le système de vérification MDG. Le chapitre 5 présente une méthode opérationnelle de définition sémantique des machines à état VHDL, qui définit leur comportement par l'exécution d'une machine théorique. Le chapitre 6 présente les outils utilisés, et certaines propriétés des langages source et cible, et leur impact sur

la réalisation du traducteur. Le chapitre 7 décrit les mécanismes de la traduction des formalismes de VHDL en MDG-HDL. Le chapitre 8 conclut le mémoire et fait des suggestions sur de futurs travaux. L'annexe A décrit en détail le sous-ensemble VHDL utilisé. L'annexe B décrit un manuel d'utilisation du traducteur.

Mes contributions originales dans le projet sont:

- La définition du sous-ensemble VHDL compilable à MDG-HDL,
- La définition d'annotations VHDL supplémentaires pour spécifier les sorts des types de signaux et de variables,
- Le choix et l'adaptation de la sémantique qui décrit le comportement des machines à état VHDL,
- L'algorithme et l'implantation du traducteur,
- Les tests de nombreux modèles VHDL.

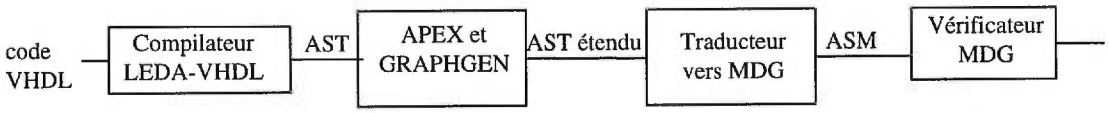


figure 1.1 Schéma typique d'une vérification de machine à état VHDL avec le système de vérification basé sur les MDGs

Nous avons présenté dans ce chapitre, les conditions générales de la réalisation du projet et l'organisation de ce mémoire. Le chapitre suivant justifie le bien fondé de la vérification formelle, présente les méthodes de vérification formelle, et quelques bases pour la description abstraite des circuits numériques.

Chapitre II

Les méthodes de vérification formelle de matériels

Le travail réalisé dans le cadre du projet a consisté à construire un traducteur, de VHDL vers le modèle supporté par le système de vérification formelle MDG. Nous justifions dans ce chapitre le bien-fondé de la vérification formelle, puis situons son utilisation dans le cadre de la synthèse des circuits numériques, et présentons les méthodes usuelles de vérification formelle de matériels.

2.1 La vérification formelle dans le processus de la synthèse

2.1.1 La pertinence de la vérification formelle

Le monde est de plus en plus dépendant des systèmes électroniques, dont la taille croît sans cesse. Cette croissance fulgurante amplifie le coût de correction des erreurs. En conséquence une demande croissante est en faveur des méthodes de conception qui peuvent produire un design correct en une seule étape de fabrication.

Avant l'avènement des systèmes de vérification, les simulateurs étaient les seuls outils pour trouver les erreurs avant la fabrication. Les outils de simulation sont utilisés, pour tester les réponses du design pour des entrées particulières. La simulation a un problème inhérent de couverture: des entrées et états particuliers sont simulés, de telle sorte que pour les designs de taille ordinaire, il y a une combinaison d'entrées et d'états que la simulation ne couvre pas [15].

Les problèmes de la simulation sont devenus critiques avec les densités d'intégration élevées, qui nécessitent des niveaux d'abstraction plus élevés. La solution est un moyen précis et compréhensible pour spécifier un comportement correct, et

une méthode exhaustive pour déterminer que le modèle du système satisfait la spécification pour toutes les entrées. La vérification formelle est une solution au problème de la couverture puisqu'elle prouve les propriétés du design sans faire référence à des vecteurs d'entrée spécifiques.

Les outils de vérification sont habituellement classés en deux catégories: ceux qui valident les spécifications, et ceux qui vérifient les implantations [15]. Un outil de validation formelle de design vérifie la présence ou l'absence de la satisfaction de propriétés spécifiques. Un outil de vérification formelle d'implantation vérifie la consistance entre une spécification, et un modèle extrait de la réalisation physique du circuit. Les outils de vérification formelle peuvent être utilisés à n'importe quelle étape du processus de la synthèse des circuits numériques.

2.1.2 La synthèse des circuits numériques

La synthèse des circuits intégrés est une suite de transformations de modèles de circuit, partant du moins détaillé vers le plus détaillé. Elle passe par différents niveaux d'abstraction et nécessite plusieurs vues des modèles de circuit.

Au niveau d'abstraction le plus élevé, le concepteur décrit le système, en utilisant des formalismes de haut niveau d'abstraction comme des équations mathématiques [14]. Aucun détail n'est spécifié sur le mode de résolution des équations, et des contraintes de temps.

Le niveau suivant est le niveau comportemental. Modéliser la fonctionnalité d'un circuit au niveau comportemental, signifie la modéliser correctement sans se soucier, ni du comportement exact à chaque cycle d'horloge, ni des unités matérielles. La description au niveau comportemental commence par un algorithme qui peut être représenté par un graphe de flot de données et de contrôle. Les arcs dans le graphe de flot de données, représentent les dépendances de donnés entre les opérations, et les noeuds représentent les opérations réalisées sur les données. Le graphe de flot de contrôle définit les différentes séquences possibles de l'exécution des opérations. Les arcs du graphe de flot de contrôle représentent l'ordre d'exécution des opérations, et les noeuds représentent l'exécution des opérations.

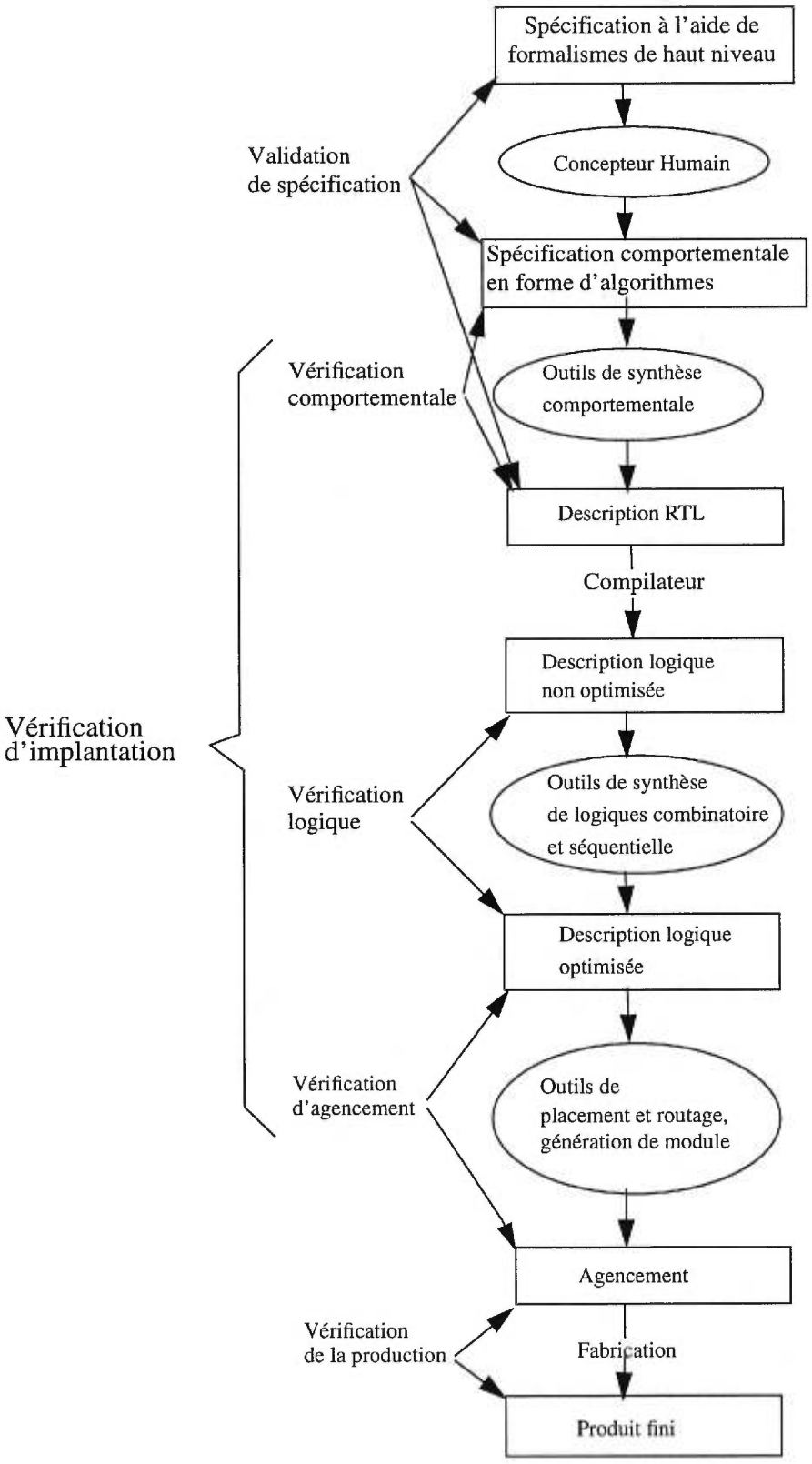


figure 2.1 Flot de synthèse de design

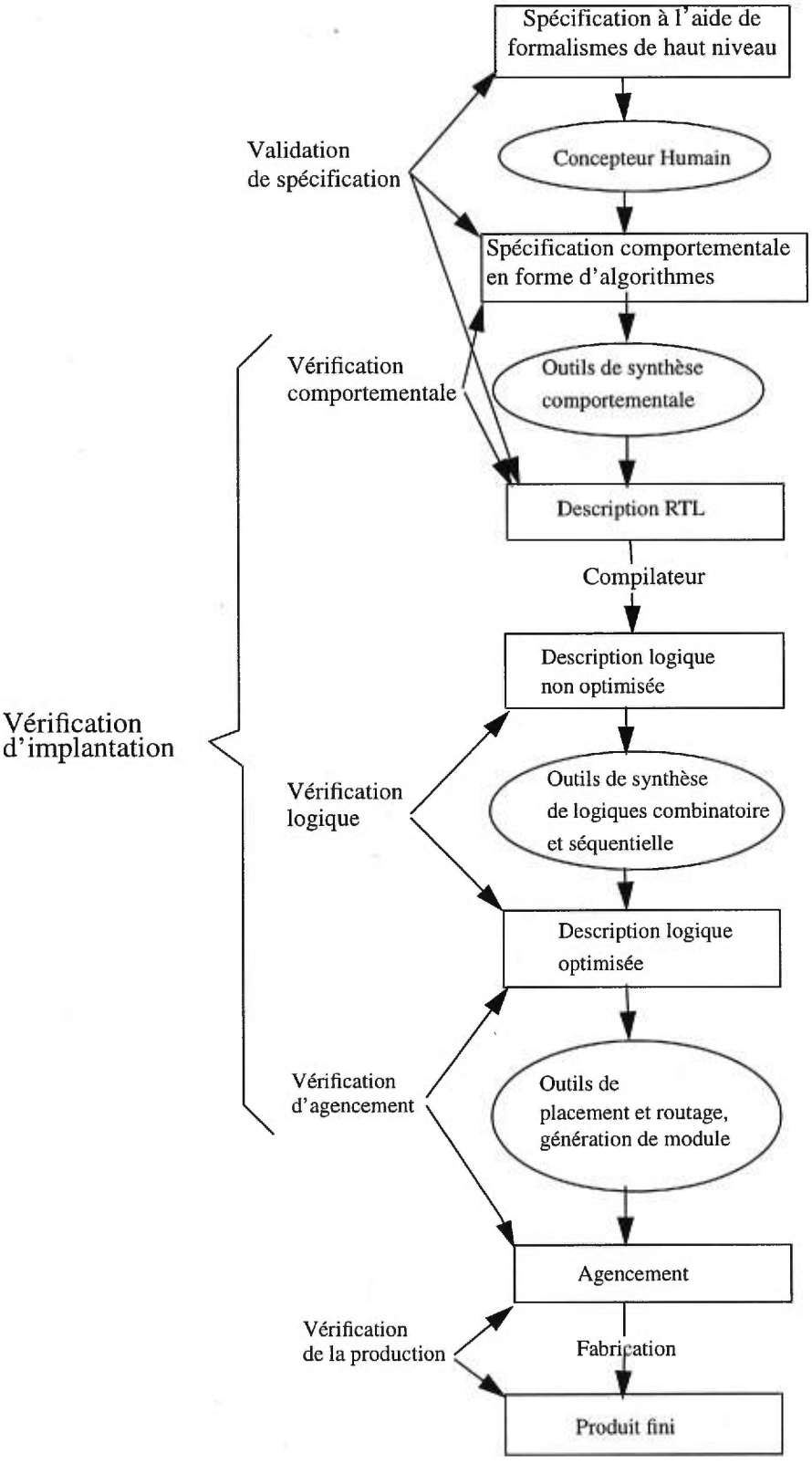


figure 2.1 Flot de synthèse de design

Le troisième niveau, le niveau transfert de registre (register transfer level - RTL), représente l'organisation du circuit. À ce niveau, les opérations d'un circuit séquentiel sont décrites comme des transferts synchrones entre les unités fonctionnelles. Ces transferts sont réalisés sous la responsabilité d'un contrôleur et sont synchronisés par une horloge. L'ordonnancement étant déterminé à ce niveau, l'ordre exact selon lequel les opérations sont exécutées sur les différentes unités fonctionnelles est typiquement contrôlé par des machines a état fini.

La synthèse logique consiste à traduire les descriptions RTL décrites dans un langage de description de matériel (hardware description language - HDL), dans une description optimale de logique combinatoire et de registres. Le but de l'optimisation logique est de minimiser la surface tout en satisfaisant des contraintes de performance.

La prochaine étape produit une description au niveau de masques lithographiques, dans une technologie donnée. Les modules ou les portes sont agencés, pour produire la description finale du circuit au niveau de masques

2.1.3 La vérification dans le flot de synthèse du design

On peut diviser le processus de la synthèse en deux parties, en fonction du type de vérification utilisé entre les niveaux d'abstraction.

2.1.3.1 La validation du design

La validation du design vérifie les propriétés de la spécification, réalisée dans un langage programmation ou dans un langage de description de matériel. Elle permettra de répondre à la question de savoir si la spécification HDL correspond aux intentions du concepteur.

2.1.3.2 La vérification des résultats d'une synthèse comportementale

Un concepteur utilisant la synthèse comportementale est concerné par le fait que le modèle RTL résultant de la synthèse comportementale soit consistant avec les

intentions du modèle original. La vérification comportementale vérifie la consistance entre, l'ordonnancement des opérations et l'allocation des unités fonctionnelles, et le modèle comportemental original.

2.1.3.3 La vérification logique

La vérification logique comporte deux éléments: la vérification séquentielle et la vérification combinatoire. En voyant un circuit séquentiel comme une machine à état fini, le problème d'équivalence de circuits séquentiels revient à montrer l'équivalence de machines à état fini. Les optimisations séquentielles peuvent concerner, par exemple le ré-encodage des états de la machine à état ou le *retiming* des registres du circuit séquentiel [10].

Une grande variété de techniques d'optimisation existe. Sur un circuit combinatoire, de nombreuses transformations booléennes peuvent être appliquées. La vérification de l'équivalence booléenne peut être appliquée pour résoudre le problème de vérification de la justesse de ces transformations.

2.1.3.4 La vérification de plan de masque lithographique

Après l'optimisation logique, est produit un plan des masques lithographiques. La vérification du plan consiste à établir l'équivalence entre le circuit extrait du plan des masques après le placement des cellules et le routage des interconnexions, et le circuit logique [14].

2.2 Les méthodes formelles de vérification

Un système de vérification formelle possède trois éléments de base: un modèle mathématique du système à vérifier, un langage formel pour formuler le problème de justesse (*correctness*), et une méthodologie pour prouver la justesse [23].

La méthodologie de preuve permet de distinguer deux tendances: la preuve par vérification de modèle (*model-checking*) et la preuve de théorème (*theorem pro-*

ving).

2.2.1 Les démonstrateurs de théorèmes

La preuve de théorème est un processus déductif. La plupart des démonstrateurs de théorèmes disponibles sont semi-automatiques au mieux, en cela qu'ils requièrent une forme d'intervention humaine pour guider le processus de développement des preuves.

Les démonstrateurs de théorème sont basés, soit sur la logique du premier ordre (Boyer-Moore), soit sur la logique d'ordre supérieur (HOL, PVS, etc.). Dans la logique du premier ordre, seules les variables du domaine peuvent être quantifiées. Si une quantification est autorisée sur des sous-ensembles de ces variables, c'est-à-dire sur des prédicats, on obtient une logique du deuxième ordre. Dans le même ordre d'idée, la logique d'ordre supérieur (*Higher-Order Logic*) autorise la quantification sur tous les prédicats et toutes les fonctions. La possibilité de quantifier ces symboles conduit à un grand pouvoir d'expressivité dans les logiques d'ordre supérieur, par rapport à la logique du premier ordre.

L'inconvénient des logiques d'ordre supérieur est l'incomplétude d'un système de preuve valide. Cela rend le raisonnement plus difficile que dans le cas de la logique du premier ordre, et des règles d'inférence et heuristiques ingénieuses sont requises. Ainsi, des inconsistances peuvent facilement survenir dans les systèmes de vérification de la logique d'ordre supérieur, si la sémantique n'est pas définie avec attention [11].

Un système de vérification par preuve de théorème intègre un générateur de preuves interactif, qui permet à l'utilisateur de construire une preuve en dirigeant le système pour appliquer des étapes spécifiques d'inférence. L'utilisateur fournit les étapes et le système les applique au but de la preuve, en les divisant progressivement en sous-buts plus simples. De façon purement syntaxique, le vérificateur de preuve pourra décider si chaque sous-but simplifié est un axiome ou provient des théorèmes déjà prouvés par une règle d'inférence.

Une vue simplifiée de la structure de preuves matériels dans HOL est repré-

sentée dans la figure 2.1. Étant données la spécification formelle "S" et l'implantation "I", le but de la vérification est de montrer soit l'équivalence de la spécification et de l'implantation "$I \leftrightarrow S$", soit que l'implantation implique la spécification "$I \rightarrow S$" [16].

L'étape 1 fixe le but de la vérification. La spécification et l'implantation dépendent habituellement de prédicats prédéfinis et de modules du langage de spécification de HOL, pour avoir des descriptions hiérarchiques, modulaires et compréhensibles. Pour exécuter les tâches de la preuve, ces définitions doivent être éclatées. Ceci peut être fait automatiquement à l'étape 2. À l'étape 3, l'utilisateur doit utiliser ses connaissances pour décomposer le but en sous-buts, utiliser des stratégies de preuves comme l'induction et les lemmes requis. L'étape 4 conduit à des simplifications logiques de sous-buts, réalisées automatiquement. L'étape 5 prouve les sous-buts et enfin l'étape 6 sauve le but comme un théorème.

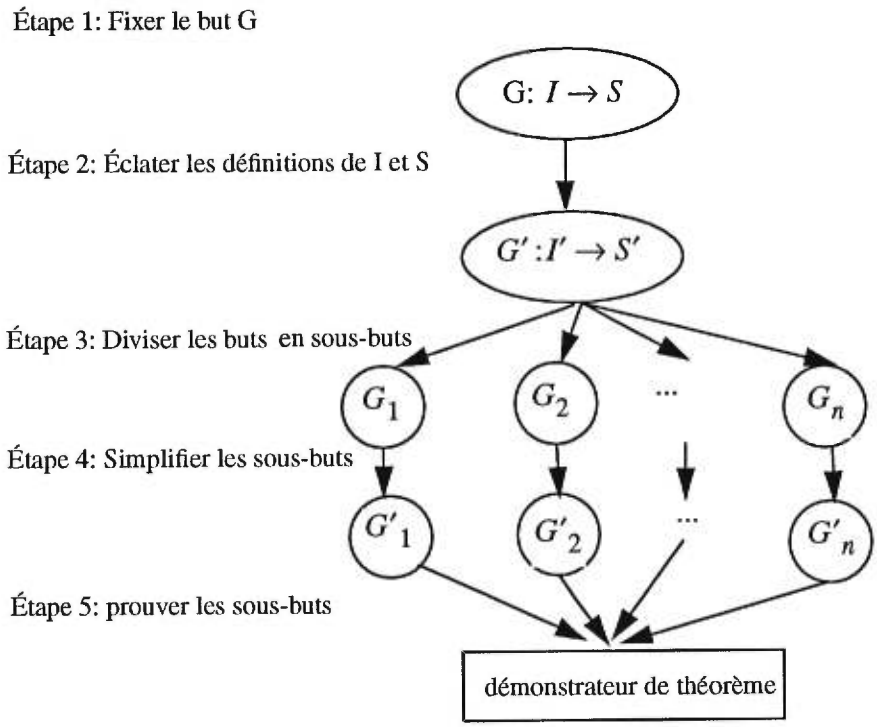


figure 2.1 Structure de la preuve de matériels avec HOL

Certains théorèmes peuvent être prouvés en utilisant des procédures de déci-

sion. L'intégration judicieuse des procédures de décision dans PVS, le distingue de HOL. Les procédures de décision peuvent être invoquées automatiquement pour améliorer les performances du système.

L'approche basée sur la preuve de théorème, établit les résultats de justesse pour n'importe quelle taille de mots. Les multiplieurs, par exemple, notoires pour rendre la vie difficile pour les vérifications basées sur les vecteurs de bits se vérifient aisément avec les démonstrateurs de théorème [22].

Les recherches en cours visent à automatiser davantage la construction des preuves.

2.2.2 Vérification de l'équivalence de deux machines à état

Les machines à état fini sont la pierre angulaire des designs de circuits synchrones. De plus, les récentes recherches ont montré le succès de la vérification formelle de circuits séquentiels, basée sur le modèle de machines à état fini.

Une machine à état fini déterministe peut être définie [6] par le sextuplet $M = (\Sigma, O, S, s_I, \delta, \lambda)$ où:

- Σ est l'alphabet d'entrée,
- O est l'alphabet de sortie,
- S est l'ensemble des états,
- $s_I \in S$ est l'état initial,
- $\delta : \Sigma \times S \rightarrow S$ est la fonction de transition.
- $\lambda : \Sigma \times S \rightarrow O$ est la fonction de sortie.

Σ, O et S sont caractérisés par $\Sigma \cap O = \varnothing$, $\Sigma \cap S = \varnothing$ et $S \cap O = \varnothing$, où $\varnothing$ représente l'ensemble vide.

La machine à état est complètement spécifiée, si les conditions suivantes sont remplies:

- $\forall(\sigma, s) \in \Sigma \times S$, $\exists s' \in S$ tel que $\delta(\sigma, s) = s'$. La fonction de transition est partout définie.

- $\forall(\sigma, s) \in \Sigma \times S$, $\exists o \in O$ tel que $\lambda(\sigma, s) = o$. La fonction de sortie est partout définie.
- $\forall o \in O$, $\exists s \in S$, $\exists \sigma \in \Sigma$ tel que $\lambda(\sigma, s) = o$. Toutes les sorties sont couvertes. La fonction de sortie est surjective.

La méthode usuelle de la vérification de l'équivalence entre deux machines [6] $M_1 = (\Sigma, O, S_1, s_{01}, \delta_1, \lambda_1)$ et $M_2 = (\Sigma, O, S_2, s_{02}, \delta_2, \lambda_2)$ consiste à construire la machine produit $M = (\Sigma, \{Vrai, Faux\}, S, s_0, \delta, \lambda)$ où nous avons $S = S_1 \times S_2$, $s_0 = (s_{01}, s_{02})$, la fonction de sortie $\lambda((s_1,s_2),x) = (\lambda_1(s_1,x) \equiv \lambda_2(s_2,x))$ et la fonction de transition $\delta((s_1,s_2),x) = (\delta_1(s_1,x), \delta_2(s_2,x))$. Alors les machines M_1 et M_2 sont équivalentes si et seulement si pour chaque état de transition de M qui peut être atteint à partir de l'état produit (s_{01},s_{02}), la machine M produit la sortie *Vrai*.

2.2.3 La vérification symbolique de modèle

La vérification symbolique de modèle (*Symbolic Model Checking*) est une méthode de vérification des propriétés d'un système, basée sur l'exploration de l'espace d'états d'une machine à état fini extraite du modèle en HDL. L'approche est symbolique parce qu'elle évite une énumération explicite des états du modèle [5].

Étant données la structure d'un domaine et les interprétations des variables individuelles et des relations de transition d'une machine à état, la vérification de modèle exécute la tâche de vérification, qui consiste à déterminer si une formule F en logique temporelle est vraie dans cette structure, en d'autres termes, la structure est-elle un modèle de F [9].

De nombreux outils de la première génération pour la vérification formelle automatique étaient basés sur deux approches théoriques [4]: la première est la vérification de modèle de la logique temporelle, où les propriétés à vérifier sont exprimées par des formules d'une logique temporelle, et le système est décrit par une machine à état fini. Elle utilise des BDDs pour représenter des fonctions booléennes qui encodent

les relations de transition et les ensembles d'états. Les représentations BDD de certains circuits comme les multiplicateurs et diviseurs croissent exponentiellement avec la taille du mot. En conséquence, il n'est pas possible d'appliquer directement les méthodes basées sur BDD pour prouver la justesse de plusieurs circuits arithmétiques intéressants avec une grande taille de mots.

En particulier, la vérification de modèle de la logique de l'arbre de calcul, une logique temporelle arborescente (Computational Tree Logic - CTL) est une technique dont les pionniers sont Clark et Emerson [4], pour vérifier si une machine à état satisfait les formules de la logique CTL. SMV [23] est un système développé à Carnegie Mellon University qui appartient à cette classe d'outils.

La seconde approche, l'inclusion de langage, requiert que le système et des propriétés soient représentés par des automates ω. On vérifie la justesse en montrant que le langage du système est contenu dans le langage de la propriété [4].

Étant données une machine à état M et une formule F, l'algorithme de vérification de modèle retourne la fonction caractéristique de l'ensemble des états accessibles, où la formule F est valide [7]. Soit S_i l'ensemble des états accessibles en i (entier positif > 0) étapes, à partir de l'étape S_0. En général, tous les états accessibles à partir de l'état initial S_0 sont visités en k étapes de telle sorte que $S_k = S_l$, pour tout l, tel que $l > k$. Le problème de calcul des états accessibles revient à un calcul de point de fixe [5], [23]. La procédure de vérification avec un vérificateur de modèle se retrouve dans la figure 2.2.

La vérification de modèle présente les avantages d'être totalement automatique, exprime facilement le comportement séquentiel, et produit un contre-exemple en cas d'échec de la vérification. Contrairement, il est confronté à l'explosion des états. Les recherches en cours ont pour objectif de résoudre le problème de l'explosion d'états.

sion. L'intégration judicieuse des procédures de décision dans PVS, le distingue de HOL. Les procédures de décision peuvent être invoquées automatiquement pour améliorer les performances du système.

L'approche basée sur la preuve de théorème, établit les résultats de justesse pour n'importe quelle taille de mots. Les multiplieurs, par exemple, notoires pour rendre la vie difficile pour les vérifications basées sur les vecteurs de bits se vérifient aisément avec les démonstrateurs de théorème [22].

Les recherches en cours visent à automatiser davantage la construction des preuves.

2.2.2 Vérification de l'équivalence de deux machines à état

Les machines à état fini sont la pierre angulaire des designs de circuits synchrones. De plus, les récentes recherches ont montré le succès de la vérification formelle de circuits séquentiels, basée sur le modèle de machines à état fini.

Une machine à état fini déterministe peut être définie [6] par le sextuplet $M = (\Sigma, O, S, s_I, \delta, \lambda)$ où:

- Σ est l'alphabet d'entrée,
- O est l'alphabet de sortie,
- S est l'ensemble des états,
- $s_I \in S$ est l'état initial,
- $\delta : \Sigma \times S \rightarrow S$ est la fonction de transition.
- $\lambda : \Sigma \times S \rightarrow O$ est la fonction de sortie.

Σ, O et S sont caractérisés par $\Sigma \cap O = \varnothing$, $\Sigma \cap S = \varnothing$ et $S \cap O = \varnothing$, où $\varnothing$ représente l'ensemble vide.

La machine à état est complètement spécifiée, si les deux conditions suivantes sont remplies:

- $\forall(\sigma, s) \in \Sigma \times S$, $\exists s' \in S$ tel que $\delta(\sigma, s) = s'$. Les transitions d'état sont complètement définies.

- $\forall s \in S$, $\forall o \in O$, $\exists \sigma \in \Sigma$ tel que $\lambda(\sigma, s) = o$. Dans chaque état, la fonction de sortie est définie pour tous les symboles de l'alphabet de sortie.

La méthode usuelle de la vérification de l'équivalence entre deux machines [6] $M_1 = (\Sigma, O, S_1, s_{01}, \delta_1, \lambda_1)$ et $M_2 = (\Sigma, O, S_2, s_{02}, \delta_2, \lambda_2)$ consiste à construire la machine produit $M = (\Sigma, \{Vrai, Faux\}, S, s_0, \delta, \lambda)$ où nous avons $S = S_1 \times S_2$, $s_0 = (s_{01}, s_{02})$, la fonction de sortie $\lambda((s_1,s_2),x) = (\lambda_1(s_1,x) \equiv \lambda_2(s_2,x))$ et la fonction de transition $\delta((s_1,s_2),x) = (\delta_1(s_1,x), \delta_2(s_2,x))$. Alors les machines M_1 et M_2 sont équivalentes si et seulement si pour chaque état de transition de M qui peut être atteint à partir de l'état produit (s_{01},s_{02}), la machine M produit la sortie *Vrai*.

2.2.3 La vérification symbolique de modèle

La vérification symbolique de modèle (*Symbolic Model Checking*) est une méthode de vérification des propriétés d'un système, basée sur l'exploration de l'espace d'états d'une machine à état fini extraite du modèle en HDL. L'approche est symbolique parce qu'elle évite une énumération explicite des états du modèle [5].

Étant données la structure d'un domaine et les interprétations des variables individuelles et des relations de transition d'une machine à état, la vérification de modèle exécute la tâche de vérification, qui consiste à déterminer si une formule F en logique temporelle est vraie dans cette structure, en d'autres termes, la structure est-elle un modèle de F [9].

De nombreux outils de la première génération pour la vérification formelle automatique étaient basés sur deux approches théoriques [4]: la première est la vérification de modèle de la logique temporelle, où les propriétés à vérifier sont exprimées par des formules d'une logique temporelle, et le système est décrit par une machine à état fini. Elle utilise des BDDs pour représenter des fonctions booléennes qui encodent les relations de transition et les ensembles d'états. Les représentations BDD de cer-

tains circuits comme les multiplicateurs et diviseurs croissent exponentiellement avec la taille du mot. En conséquence, il n'est pas possible d'appliquer directement les méthodes basées sur BDD pour prouver la justesse de plusieurs circuits arithmétiques intéressants avec une grande taille de mots.

En particulier, la vérification de modèle de la logique de l'arbre de calcul, une logique temporelle arborescente (Computational Tree Logic - CTL) est une technique dont les pionniers sont Clark et Emerson [4], pour vérifier si une machine à état satisfait les formules de la logique CTL. SMV [23] est un système développé à Carnegie Mellon University qui appartient à cette classe d'outils.

La seconde approche, l'inclusion de langage, requiert que le système et des propriétés soient représentés par des automates ω. On vérifie la justesse en montrant que le langage du système est contenu dans le langage de la propriété [4].

Étant données une machine à état M et une formule F, l'algorithme de vérification de modèle retourne la fonction caractéristique de l'ensemble des états accessibles, où la formule F est valide [7]. Soit S_i l'ensemble des états accessibles en i (entier positif > 0) étapes, à partir de l'étape S_0. En général, tous les états accessibles à partir de l'état initial S_0 sont visités en k étapes de telle sorte que $S_k = S_l$, pour tout l, tel que $l > k$. Le problème de calcul des états accessibles revient à un calcul de point de fixe [5], [23]. La procédure de vérification avec un vérificateur de modèle se retrouve dans la figure 2.2.

La vérification de modèle présente les avantages d'être totalement automatique, exprime facilement le comportement séquentiel, et produit un contre-exemple en cas d'échec de la vérification. Contrairement, il est confronté à l'explosion des états. Les recherches en cours ont pour objectif de résoudre le problème de l'explosion d'états.

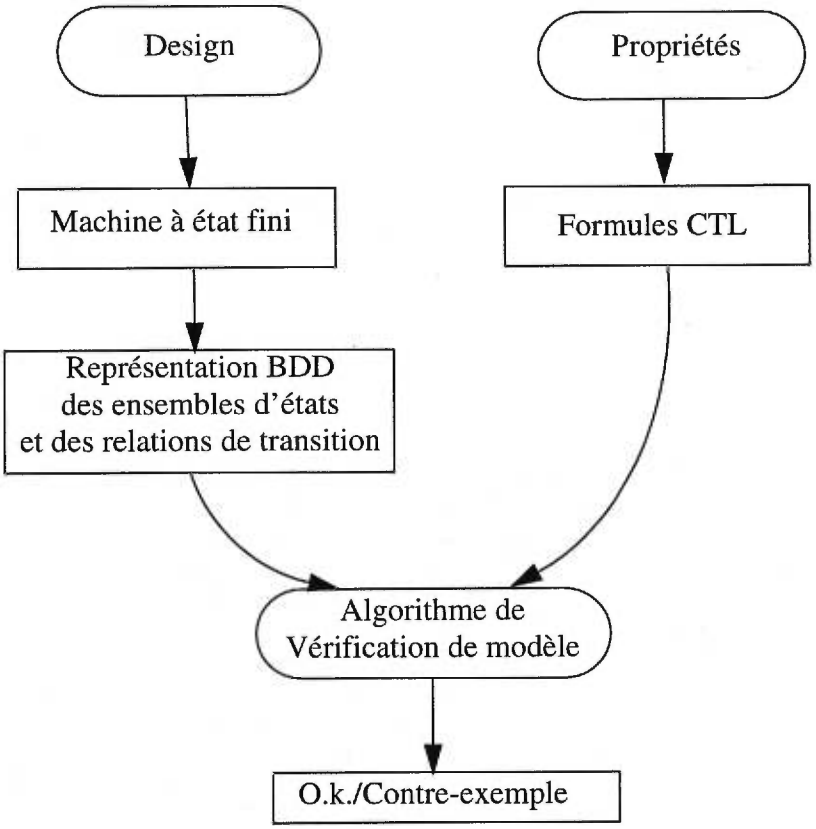


figure 2.2 Procédure de vérification avec un vérificateur de modèle

2.2.4 Les tendances: intégration de différentes techniques de preuves

L'introduction de la vérification de modèle comme procédure de décision dans PVS est décrite dans [26]. Dans l'application de l'approche preuve de théorème aux problème de vérification, l'on vérifie une propriété P d'un programme M en prouvant que $M \supset P$. L'approche vérification de modèle vérifie le même programme en montrant que la machine à états de M est un modèle satisfaisant de P, à savoir $M \models P$. Ces deux points de vue sont unifiables. Cette unification permet d'incorporer un vérificateur de modèle comme procédure de décision pour un sous-ensemble bien défini de PVS [26].

C@S est une nouvelle approche qui fournit un cadre général [26], pour explorer différentes techniques de preuves, en vue d'automatiser totalement le processus de

construction de la preuve. Le système a pour objectif de fournir différentes procédures de décision, qui sont invoquées aussitôt que des buts de preuves automatisables sont détectés. Les autres preuves de vérification sont divisées interactivement en sous-buts jusqu'à ce qu'elles puissent être prouvées par des procédures de décision. Comme résultat C@S enrichit HOL, de nombreuses procédures de décision et méthodes de preuves (preuve par théorème de la logique temporelle et vérification de modèle, vérification de modèle CTL comme implanté dans SMV, automates ω, etc.) [25].

2.3 Le système de vérification formelle MDG

Le système de vérification formelle MDG est basé sur une nouvelle classe de graphes de décision appelée Graphes de Décision Multi-choix ou *Multiway Decision Graphs* (MDG). Les MDGs sont une représentation efficace d'une classe de formules de la logique du premier ordre, en distinguant les sorts abstraits (non-interprétés) des sorts concrets (définis par leur ensemble de valeurs). Dans un MDG, un signal de données est représenté par une variable de sort abstrait (au lieu d'un vecteur de variables booléennes) et une opération sur les données est représentée par un symbole fonctionnel non-interprété [30].

Cette représentation est plus compacte que les ROBDD, utilisés dans la représentation des formules logiques dans les vérificateurs de modèle. Le langage de description de matériels MDG-HDL qui supporte le modèle MDG est décrit dans le Chapitre V.

La compacité des MDGs réduit le problème d'explosion d'états au niveau booléen, rencontré dans la vérification de modèle en représentant les signaux de données indépendamment de leur largeur. Le système de vérification MDG présente aussi l'avantage d'être automatique et peut fournir des contre-exemples, comme les vérificateurs de modèles. Il permet par ailleurs de réaliser la vérification à des niveaux d'abstraction plus élevés que le niveau booléen.

2.3.1 Les symboles de MDG

Soit f un symbole fonctionnel n-aire (n>0) de type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$ où $\alpha_1 \ldots \alpha_{n+1}$ sont des sorts. La distinction entre les sorts conduit à distinguer trois types de symboles de fonction. Si α_{n+1} est de sort abstrait, alors f est un symbole fonctionnel de sort abstrait. Si α_{n+1} est de sort concret, et qu'au moins un symbole $\alpha_1 \ldots \alpha_n$ est abstrait, alors f est un opérateur croisé. Si tous les symboles $\alpha_1 \ldots \alpha_{n+1}$ sont concrets, alors f est un symbole fonctionnel concret. Les symboles de fonction abstraits et les opérateurs croisés sont non-interprétés.

Les constantes apparaissant dans une énumération sont appelées des constantes individuelles, et les autres constantes sont des constantes génériques. Il existe deux ensembles disjoints de termes: les termes concrètement réduits et les termes croisés. Les termes concrètement réduits sont définis inductivement comme suit: (i) les constantes individuelles; (ii) les constantes abstraites génériques; (iii) les variables abstraites: (iv) et les termes de la forme "$f(A_1,\ldots, A_n)$" où f est symbole fonctionnel abstrait et $A_1, \ldots, A_n$ sont des termes concrètement réduits. Les termes croisés ont la forme "$f(A_1,\ldots, A_n)$",où f est un opérateur croisé et $A_1, \ldots, A_n$ sont des termes concrètement réduits.

Les opérateurs croisés sont utilisés pour modéliser les rétroactions, en provenance des chemins de données des circuits de contrôle. Les symboles fonctionnels concrets doivent avoir une définition explicite, ils doivent par conséquent être éliminés et n'apparaissent pas dans les MDGs. MDG-HDL utilise des variables abstraites pour représenter les signaux de données, et des symboles de fonctions non-interprétées pour représenter les opérations sur les données.

2.3.2 Les graphes de décisions multi-choix

Un graphe de décisions multi-choix (MDG) est un graphe fini orienté acyclique G où les noeuds feuilles sont étiquetés par des formules, les noeuds internes sont étiquetés par des termes, et les arcs provenant d'un noeud interne N sont étiquetés par des termes de même sort que l'étiquette de N. Un tel graphe représente une formule définie

inductivement comme suit: (i) si G est un noeud feuille unique étiqueté par une formule P, alors G représente P; (ii) si G a un noeud racine étiqueté A avec des arcs $B_1 \ldots B_n$ conduisant à des sous-graphes $G'_1 \ldots G'_n$, et si chaque G'_i représente une formule P_i, alors G représente la formule $\bigcup_{1 \le i \le n} ((A = B_i) \wedge P_i)$.

Un noeud interne doit être étiqueté, soit par une variable de sort abstrait avec des arcs étiquetés par des termes concrètement réduits du même sort, soit par une variable de sort concret avec des arcs étiquetés par des constantes individuelles dans l'énumération du sort, soit par des termes croisés avec des arcs étiquetés par des constantes individuelles dans l'énumération du sort du terme croisé. Une feuille doit être étiquetée par T, à l'exception du cas où le graphe a un seul noeud étiqueté par F.

2.3.3 Formule Directe

Etant donnés deux ensembles disjoints de variable U et V, une *formule directe* (*Directed Formula* - DF) de type $U \rightarrow V$ est une formule en forme normale disjonctive (*Disjunctive Normal Form* - DNF) telle:

1. Chaque opérande de la disjonction est une conjonction d'équations de la forme
   - $A = a$, où A est un terme croisé de sort concret α ne contenant aucune variable autres que les éléments de U, et a est une constante individuelle dans la numération de α, ou,
   - $u = a$ où $u \in U$ est une variable de sort concret α et a est une constante individuelle dans la numération de α, ou,
   - $v = a$ où $v \in V$ est une variable concrète de sort α, et a est une constante individuelle dans l'énumération de α, ou,
   - $v = A$ où $v \in V$ est une variable de sort abstrait α et A est un terme de sort α ne contenant aucune autre variable que les éléments de U;
2. Dans chaque opérande de la disjonction, les membres de gauche des équations sont différents; et
3. Chaque variable $v \in V$ apparaît dans le membre de gauche d'une équation $v = A$ dans chacun des opérandes de la disjonction.

2.3.4 Machine à état abstrait

Le comportement d'un circuit numérique peut être modélisé par une machine à état abstrait, qui est ensuite représenté pas des MDGs.

Une machine à état abstrait est définie par le tuple $D = (X, Y, Z, Y', \eta, F_I, F_T, F_O)$ avec:

1. X, Y, Z sont des ensembles finis deux à deux disjoints, de variables d'entrée, d'état et de sortie.
2. Y' est ensemble fini des variables d'état suivant disjoint de $X \cup Y \cup Z$.
3. η est une fonction qui fait correspondre à chaque variable d'état une variable d'état suivant.
4. F_I est une formule Directe (DF) du type $U_0 \rightarrow Y$, où U_0 est un ensemble de variables abstraites disjoint de $Y \cup Y \cup Y' \cup Z$. F_I est une description abstraite des états initiaux.
5. F_T est une formule directe du type $X \cup Y \rightarrow Y'$. F_T est une description abstraite de la relation de transition.
6. F_O est une formule directe du type $X \cup Y \rightarrow Z$. F_O est une description abstraite de la relation de sortie.

2.3.5 Procédure de vérification avec MDG

Les formules exprimant les propriétés sont également représentées par des MDGs. L'analyse d'accessibilité des MDGs de la machine à état abstrait et des MDGs des propriétés, permet de déterminer si la propriété est un invariant du design. En cas d'échec de la vérification, un contre-exemple est produit. La procédure de vérification des propriétés de designs est représentée dans la figure 2.3.

L'utilisation de variables abstraites et de symboles fonctionnels non-interprétés est responsable dans certains cas de la non-terminaison de la procédure d'analyse d'accessibilité.

Les recherches en cours visent, au moins à minimiser la classe de circuits responsables de non-terminaison, sinon les supprimer, développer des algorithmes pour la vérification de modèle en utilisant une logique temporelle du premier ordre, supportant les représentations abstraites des données, et intégrer MDG à un démonstrateur de

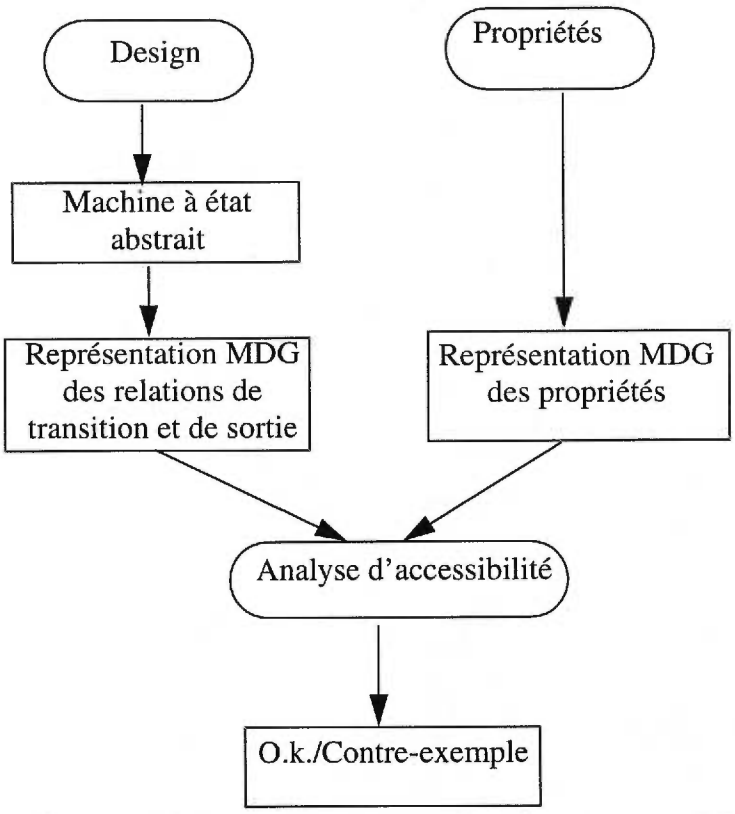


figure 2.3 Procédure de vérification d'invariants avec MDG

théorème comme une procédure de décision.

Ce chapitre a présenté la vérification formelle dans le processus de la synthèse, les méthodes de la vérification formelle, et a introduit le système de vérification formelle basé sur MDG.

Les circuits numériques que nous voulons vérifier sont décrits initialement en VHDL d'où un modèle de machines á état est extrait. Dans le chapitre suivant nous décrivons les modèles de circuits numériques en VHDL, puis nous présentons les formalismes particuliers du langage VHDL que nous acceptons. Enfin nous présentons des styles de description des machines à état qui en découlent.

Chapitre III

Modèle de spécification VHDL de machines à état

VHDL est un langage de description de matériels permettant de décrire les circuits numériques à plusieurs niveaux d'abstraction. Il ne fait d'hypothèse, ni sur la technologie des circuits, ni sur la méthodologie de conception des circuits. Nous allons définir dans ce chapitre, quelques propriétés du sous-ensemble de VHDL, utilisé pour décrire le comportement des circuits séquentiels, modélisables au niveau transfert-registre (Register Transfer Level - RTL) par une machine à état fini déterministe.

3.1 Les modèles de description des circuits en VHDL

On distingue deux modèles de description de circuits: le modèle comportemental et le modèle structurel. L'unité de base du modèle comportemental est le processus et celle du modèle structural est le composant.

3.1.1 Entité de design

Une entité de design est l'abstraction primaire des descriptions de matériels en VHDL. Contenant le modèle des circuits, elle se compose de deux unités de bibliothèque (library units) compilables séparément: une déclaration d'entité définissant l'interface qui comporte les ports d'entrée et de sortie, et une architecture définissant le modèle. L'entité de design peut décrire des circuits numériques à n'importe quel niveau d'abstraction compris entre le niveau comportemental et le niveau logique.

Une entité de design peut être décrite comme une hiérarchie de blocs d'énon-

cés concurrents ou comme une interconnexion de composants. Chaque composant peut être décrit comme une hiérarchie de composants. L'entité de design de plus bas niveau dans la décomposition hiérarchique d'un composant décrit un comportement.

Le paquetage est une autre unité de bibliothèque comportant une partie déclarative et un corps. Un corps de paquetage peut encapsuler des types de donnés, des déclarations de signaux, des déclarations de composants, et des sous-programmes.

3.1.2 Le modèle comportemental

Le comportement d'un système peut être défini par un algorithme. En général, le comportement peut être décomposé en un certain nombre d'activités concurrentes. Chaque activité peut être décrite par un processus qui peut communiquer avec d'autres processus par des signaux.

En VHDL, la description d'un circuit numérique peut être à deux niveaux: séquentiel et concurrent [21]. Le niveau séquentiel nécessite de décrire le comportement de chaque processus par un programme séquentiel, alors que le niveau concurrent nécessite de définir comment interagissent plusieurs processus, qui s'exécutent en parallèle et communiquent par des signaux.

3.1.2.1 Les processus VHDL

Un processus est toujours actif et s'exécute (il est actif) en tout temps tant qu'il n'est pas suspendu. Le mécanisme pour suspendre et activer conditionnellement un processus utilise, soit une liste de sensitivité de processus (*process sensitivity list*), soit un énoncé *wait* comportant également une liste de sensitivité (*wait sensitivity list*).

La liste de sensitivité d'un processus est une liste de signaux spécifiés entre des parenthèses après le mot clé *process*. Le processus est activé quand un évènement survient sur n'importe quel signal de la liste. Le processus devient suspendu après l'exécution du dernier énoncé, jusqu'à ce qu'un autre évènement survienne sur un signal auquel il est sensitif. Sans considérer les évènements des signaux de la liste de sensitivité, les processus sont exécutés une fois jusqu'à la suspension, au début de la

simulation.

L'énoncé séquentiel *wait* est un formalisme comportemental pour suspendre et activer l'exécution des processus. Cet énoncé peut apparaître sous quatre formes [21]. Nous utilisons les deux formes suivantes:

WAIT ON *liste_de_sensitivité_de_wait*;

WAIT UNTIL *condition_de_wait*;

La suspension causée par "WAIT ON" est rompue lorsqu'un évènement survient sur un signal de la liste de sensitivité (*liste_de_sensitivité_de_wait*). Quand un processus est suspendu par un "WAIT UNTIL", l'exécution reprend quand la condition booléenne qui lui est associée (*condition_de_wait*) passe de FAUX à VRAI.

3.1.3 Le modèle structural

La partition d'un modèle en composantes fonctionnelles facilite la description des circuits numériques complexes. Chaque composante fonctionnelle peut-être implantée par un ou plusieurs composants (physiques). Unités de base des descriptions structurales, les composants sont des modèles de sous-systèmes connectés par des signaux. Ils sont considérés comme des boîtes noires par rapport aux opérations qu'ils réalisent, ce qui permet d'ignorer l'implantation effective de ces opérations.

La description structurale d'un circuit définit les interfaces de ses composants, et comment ils sont connectés les uns aux autres. Une déclaration de composant définit un composant formel qui ne peut pas exister dans une bibliothèque de design. Elle doit apparaître, soit dans un paquetage, soit dans la partie déclarative d'une architecture. Une opération de configuration établit les associations entre une déclaration de composant, et une entité de design qui existe dans une bibliothèque de design. La configuration peut permettre d'avoir différentes vues d'un même design, en le configurant différemment sans avoir recours à une analyse du design.

Une instantiation de composant crée une instance d'un composant formel (défini dans la déclaration) et identifie chacun des signaux de l'instance à un signal de son environnement. Seule la vue externe de l'instance est visible. Une instantiation de composant ne peut exister que, soit dans un paquetage soit dans un corps d'architec-

ture.

Dans la figure 3.1 se retrouve la description structurale d'un demi-additionneur. Les descriptions comportementales des composants *and2* et *or2* présentées dans la section 7.4, ont été omises dans cette section, pour permettre de se focaliser sur la structure du demi-additionneur. Cette description comporte deux déclarations de composants *and2* et *or2,* configurés respectivement avec les entités *and_gate* et *or_gate* et les architectures *beh_and_gate* et *beh_or_gate,* et dont les instances sont étiquetées par *i1* et *i2*.

```
-- déclarations d'entité
entity half_adder is
port(
    x: in bit:='0';
    y: in bit:='0';
    sum: out bit:='0';
    carry: out bit:='0');
end half_adder;
```

```
-- description d'architecture
architecture str_half_adder of half_adder is
    component and2
        port(and_i1: in bit; and_i2: in bit; and_o: out bit);
    end component;

    component or2
        port(or_i1: in bit; or_i2: in bit; or_o: out bit);
    end component;

    use work.all;
    for all: and2 use entity and_gate(beh_and_gate);
    for all: or2  use entity or_gate(beh_or_gate);

    begin
        l1: or2
        port map(or_i1 => x, or_i2 => y, or_o => sum);

        l2: and2
        port map(and_i1 => x, and_i2 => y, and_o => carry)

end str_half_adder;
```

Déclarations de deux composants

Configuration de composants

Instantiation de composants

figure 3.1 Description structurale d'un demi-additionneur

3.2 Le sous-ensemble VHDL

Le langage VHDL ne comportant pas de format spécifique pour la description des machines à état fini, nous avons adopté des choix d'implantation des mécanismes d'activation et de suspension des processus, et de synchronisation de la machine avec l'horloge, pour faciliter la reconnaissance de la machine à état par le traducteur. En outre, des annotations sont réalisées pour attribuer des sorts aux signaux et variables.

Toutes les opérations de la logique séquentielle synchrone dépendent d'un signal de l'environnement externe, en particulier pour déterminer quand les transitions ont lieu. Ce signal est le signal d'horloge.

Les descriptions RTL spécifient les opérations qui surviennent à chaque cycle d'horloge. Tout modèle de machine à état fini déterministe en VHDL, comporte les opérations suivantes:

1. La synchronisation de la machine avec l'horloge.
2. La spécification des transitions d'état.
3. Les affectations aux sorties.
4. La fixation des opérations exécutées dans chaque état.
5. Une condition de remise synchrone optionnelle à l'état initial.

Les machines à état que nous spécifions doivent remplir les conditions suivantes:

1. Les machines sont déterministes.
2. Les machines sont complètement spécifiées.

3.2.1 L'activation et la suspension des processus

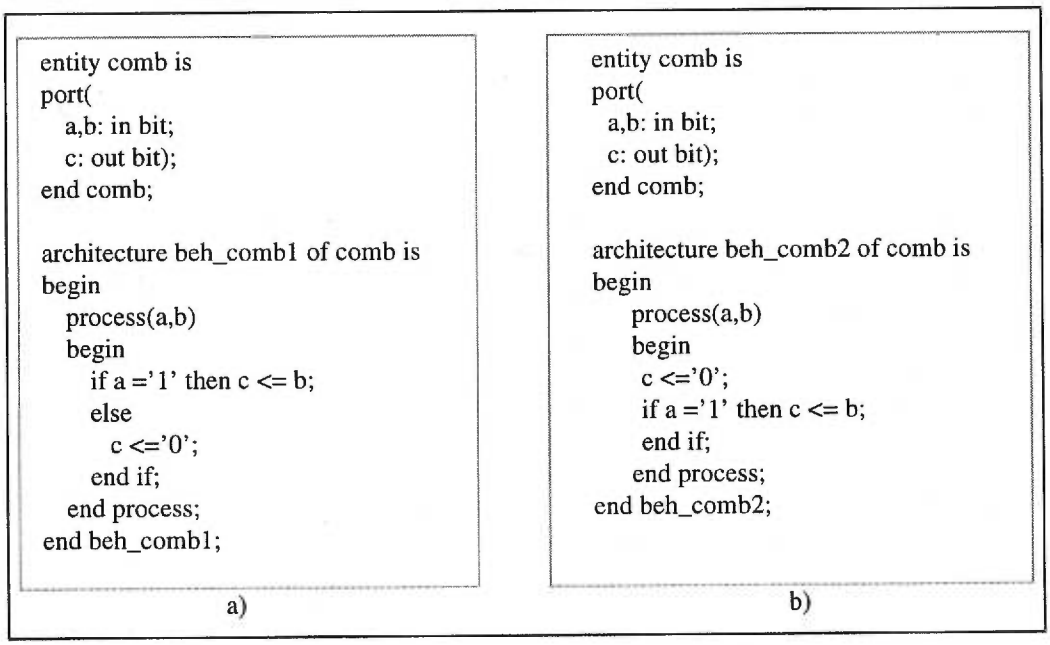
La description d'un modèle comportemental au niveau transfert-registre est constituée d'un processus séquentiel, et d'un certain nombre de processus combinatoires, dont les mécanismes d'activation et de suspension sont décrits ci-dessous. Des exemples de description sont données dans la section 3.3.

3.2.1.1 Activation et suspension des processus combinatoires

Nous avons choisi de ne pas utiliser d'énoncé *wait* dans les processus combinatoires, pour que le mécanisme de suspension et d'activation des processus combinatoires n'utilise qu'une liste de sensitivité de processus, qui contient tous les signaux d'entrée du circuit (ou de la portion de circuit) que le processus modélise.

Tous les signaux de sortie des processus combinatoires doivent, soit faire l'objet d'une affectation dans chacune des alternatives des énoncés de contrôle (un exemple se retrouve dans la figure 3.2.a), soit faire l'objet d'une affectation par défaut avant les énoncés de contrôle (un exemple se retrouve dans la figure 3.2.b). Autrement le signal de sortie est sensé conserver son ancienne valeur dans l'alternative où sa nouvelle valeur n'est pas définie, et le processus n'est plus combinatoire.

Les processus combinatoires de la figure 3.2 sont activés par des évènements survenant sur les signaux d'entrée *a* ou *b*, puis ils affectent une valeur à la variable de sortie *c* en fonction de la valeur de *a*, puis sont suspendus, en attente du prochain évènement sur *a* ou *b*.



```
entity comb is
port(
   a,b: in bit;
   c: out bit);
end comb;

architecture beh_comb1 of comb is
begin
   process(a,b)
   begin
      if a ='1' then c <= b;
      else
        c <='0';
      end if;
   end process;
end beh_comb1;
```

a)

```
entity comb is
port(
  a,b: in bit;
  c: out bit);
end comb;

architecture beh_comb2 of comb is
begin
     process(a,b)
     begin
      c <='0';
      if a ='1' then c <= b;
      end if;
     end process;
end beh_comb2;
```

b)

figure 3.2 Exemple de processus décrivant la logique combinatoire

3.2.1.2 Activation et suspension des processus séquentiels

Contrairement aux processus combinatoires, nous avons choisi de ne pas avoir de signaux dans la liste de sensitivité de processus et d'utiliser un énoncé *wait* pour la suspension et l'activation de chaque processus. L'énoncé *wait* est le premier énoncé du processus qui peut être sous l'une des formes indiquées dans la figure 3.3.

wait on *clock_signal*;
if *clock_signal*'event and *clock_signal_condition*;

ou

wait until *clock_signal*'event and *clock_signal_condition*;

Où *clock_signal* est le nom du signal d'horloge et *clock_signal_condition* est une condition booléenne portant sur *clock_signal* qui peut être, soit *clock_signal* = *'1'* (front montant) soit *clock_signal* = *'0'* (front descendant).

figure 3.3 Formes de l'énoncé *wait* pour le codage des processus séquentiels

La figure 3.4 représente deux entités de design dont les architectures contiennent chacune un processus séquentiel. Chaque processus utilise une des formes du *wait* décrite ci-dessus. Les deux processus additionnent les deux signaux d'entrée *a* et *b*, et l'affectent au signal *c*, à chaque cycle de l'horloge *clk*. L'utilisation du type *std_logic* et de l'addition de signaux de type *std_logic* nécessite d'inclure respectivement les paquetages *std_logic_1164* et *std_logic_arith* de la bibliothèque *IEEE*. Ceci est exprimé par les trois premiers énoncés de la description.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity seq is
port(
   clk: in std_logic;
   a,b: in std_logic;
   c: out std_logic);
end seq;

architecture beh_seq1 of seq is
begin
   process
   begin
     wait until clk'event and clk ='1';
     c<= a + b;
   end process;
end beh_seq1;
```

a)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity seq is
port(
   clk: in std_logic;
   a,b: in std_logic;
   c: out std_logic);
end seq;

architecture beh_seq2 of seq is
begin
    process
    begin
      wait on clk;
      if clk'event and clk ='1' then
         c<= a + b;
      end if;
   end process;
end beh_seq2;
```

b)

figure 3.4 Exemples de processus séquentiels utilisant les deux formes de *wait* retenues

3.2.2 La synchronisation de la machine avec l'horloge

Cette section, à l'instar de la précédente, décrit le choix d'implantation d'une opération de la machine à état. Des styles de description de machines à état, basés sur ces choix sont présentés dans la section 3.3.

Un signal de synchronisation est un signal dont le changement de valeur détermine si une partie du circuit va être activée ou non. Tous les signaux apparaissant dans la liste de sensitivité d'un énoncé *wait* ou d'un processus sont définis dans le manuel de référence de VHDL [12] comme des signaux synchronisant le processus.

L'exemple le plus courant des signaux de synchronisation est le signal d'horloge. Chaque processus séquentiel doit comporter un signal d'horloge qui cadence les

opérations de la machine à état. Nous implantons le signal d'horloge comme le seul signal de la liste de sensitivité de *wait*. Associé à l'attribut prédéfini *event*, et à la condition *clock_signal ='0'* ou *clock_signal ='1'*, le signal d'horloge permet d'activer le processus séquentiel à chaque front montant ou descendant. Dans l'exemple de la figure 3.4, le signal d'horloge *clk* est activé à chaque front montant.

3.2.3 La remise à l'état initial

La remise à l'état initial est synchrone quand son effet dépend du signal d'horloge. Dans ce cas, la condition de remise à l'état initial et l'affectation des valeurs initiales vient après l'énoncé *wait* dans l'ordre séquentiel des énoncés du processus (la figure 3.5.b illustre cette situation). Autrement, la condition de remise à l'état initial et l'affectation des valeurs initiales sont les premiers énoncés du processus, et la remise à l'état initial est asynchrone (la figure 3.5.a illustre cette situation). Nous permettons la remise à l'état initial synchrone dans nos descriptions.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity seq is
port(
   clk: in std_logic;
   rst: in std_logic;
   a,c: in std_logic;
   c: out std_logic);
end seq;

architecture beh_seq1 of seq is
begin
   process
   begin -- remise asynchrone
      if(rst ='0') then a <='0';
      else
         wait until clk'event and clk ='1';
         a <= b + c;
      end if;
   end process;
end beh_seq2;
```

a)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity seq is
port(
   clk: in std_logic;
   rst: in std_logic;
   a,c: in std_logic;
   c: out std_logic);
end seq;

architecture beh_seq2 of seq is
begin
   process
   begin -- remise synchrone
      wait until clk'event and clk ='1';
      if(rst ='0') then a <='0';
      else
         a <= b + c;
      end if;
   end process;
end beh_seq2
```

b)

figure 3.5 Processus séquentiels avec condition de remise à l'état initial

3.2.4 Le délai d'assignation δ

Dans sa forme la plus simple, une affectation à un signal cible *a* de l'expression *expr* se présente comme suit: *a* <= *expr*. Un délai δ s'écoulera avant que le signal *a* ne prenne la valeur de l'expression *expr*. Le délai δ est le temps qui sépare deux cycles successifs d'exécution de processus. C'est cette forme simplifiée que nous avons retenue.

Le délai δ est une restriction à délai zero dans les affectations aux signaux. La forme générale d'une affectation à un signal contient une clause *after* qui spécifie un délai fini additionnel.

Contrairement aux signaux, la mise à jour des variables cibles se fait sans

délais suite aux affectations.

3.2.5 La spécification des sort des données

Chaque signal de MDG est, soit de sort concret soit de sort abstrait. Nous avons prédéfini les sorts des données de certains types prédéfinis de VHDL puisqu'il n'est pas possible de définir des attributs pour les types prédéfinis, par exemple les données de type *integer* sont de sort abstrait, et les données de type *std_logic*, *boolean* et *bit* sont de sort concret. Pour les types de données définis par l'utilisateur, soit il s'agit d'un type d'énumération et le sort est concret, soit il s'agit d'un sous-type numérique défini sur un segment fini (de bornes inférieures et supérieures définies) et le sort est concret, soit il s'agit de tableau dont les éléments sont de type *bit* (*bit_vector*) ou *std_logic* (*std_logic_vector*) dont les index sont définis sur un segment fini et le sort est concret, soit l'utilisateur doit définir explicitement le sort du type de données.

La spécification explicite du sort d'un type de données VHDL, nécessite:

- au préalable de déclarer l'attribut *sort* comme suit:

```
type sort_type is (concrete, abstract);
attribute sort: sort_type;
```

- puis de spécifier l'attribut *sort*. Par exemple nous spécifions le sort abstrait des signaux de type *positive*, comme suit:

```
attribute sort of positive: type is abstract;
```

où *positive* est le type de données dont le sort est abstrait.

3.2.6 Spécification de boîtes noires

Le modèle de description de matériels du langage cible de notre traducteur, autorise des composants abstraits dont on ignore tout comportement. Nous appelons des boîtes noires, ces composants qui ne sont définis que par leur interface avec l'environnement extérieur.

L'entité de design de plus bas niveau dans une description en VHDL décrit un comportement. Le concept de boîte noire n'existe donc pas en VHDL. Pour simuler l'existence de ce type de composant, nous allons nous servir de la syntaxe de la spécification de la configuration des composants. La forme que nous avons retenue est présentée dans la figure 3.6 (voir l'annexe A pour de plus amples informations).

```
configuration_specification ::=
    for component_specification use binding_indication

component_specification ::=
      instantiation_list : component_name

instantiation_list ::=
      instantiation_label

binding_indication ::=
      entity entity_name[(architecture_identifier)]
```

figure 3.6 La syntaxe de la spécification de la configuration des composants

Les règles de production de la figure 3.6 dérivées des règles de production dans [12], utilise la forme de Backus-Naur, pour définir la syntaxe de la spécification de la configuration des composants que nous acceptons. Les deux crochets droits ([]) dans la description de l'indication du lien (*binding indication*) qui définit le nom de l'entité et de l'architecture, indique que le nom de l'architecture est optionnelle. Comme le comportement de l'entité de design est défini dans l'architecture, lorsque le nom de l'architecture est absent, le composant instancié est considéré comme une boîte noire. Dans la description structurale de l'additionneur de la figure 3.1, les instances de composants *or2* et *and2* ne sont pas des boîtes noires puisque les noms des architectures sont indiqués.

3.3 Exemples de styles de description des machines à état

Une même machine à état peut être décrite avec différents styles. Les différents styles de description des machines à état sont caractérisés par le choix, du nom-

bre de processus, et de la représentation de l'état de la machine. On peut distinguer parmi ces processus, un processus séquentiel qui est le seul à décrire la synchronisation de la machine avec l'horloge et la remise à l'état initial. Toutes les autres opérations peuvent être décrites dans n'importe quel processus. De plus on peut décrire le comportement de chaque variable de sortie, et de chaque variable d'état suivant, par un processus combinatoire.

Lorsque l'état de la machine est représenté explicitement par une variable d'état ou un signal, la machine à état est explicite. Dans le cas contraire, l'état de la machine est représenté par un ensemble de variables et de signaux et la machine à état est implicite.

Tous les styles de description doivent être conformes aux restrictions de la section 3.2. Nous présentons dans les sections suivantes quelques exemples de styles de description. Les deux premiers exemples utilisent un seul processus avec des représentations implicites et explicites de l'état de la machine. Les deux exemples suivants utilisent deux processus, avec une représentation explicite de l'état de la machine.

3.3.1 Modélisation de machines à état implicite par un seul processus

Dans une description de machine à état implicite, nous ne décrivons que les activités qui surviennent à chaque cycle d'horloge. L'état de la machine n'est pas décrite explicitement: la machine ne comporte pas de registre d'état explicite.

Le contrôle de la machine est décrit en utilisant un ensemble d'énoncés "if-then-else" imbriqués. Une alternative des "if-then-else" modélise un état de la machine, et est représentée par tous les signaux et les variables utilisés dans les expressions conditionnelles de cette alternative. Par exemple, le processus de la figure 3.7, comporte trois alternatives de "if-then-else". Dans la première alternative, l'état de la machine est défini par la condition *set = '0'* avec le signal *set*. Dans la seconde alternative l'état est défini par la condition *set /= '0'* (le contraire de la condition précédente) et *clr = 0*, avec les signaux *set* et *clr*. Enfin dans la dernière alternative l'état de la machine est définie par les conditions *set /= '0'* et *clr /= '0'*.

Si un processus rencontre les critères suivants, il implante une machine à états

implicite:

- Le processus contient au moins un énoncé wait. Nous avons retenu de ne considérer qu'un seul énoncé *wait*, pour des raisons de simplicité.
- L'utilisateur n'utilise pas une variable ou signal comme variable (ou signal) d'état explicite.

L'exemple dans la figure 3.7. décrit une machine à état implicite par un processus séquentiel.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity bascule is
 port(
     data: in std_logic;
      clk, set, clr: in std_logic;
      outp: out std_logic);
end bascule;
```

```
architecture imp_fsm_bascule of bascule is
begin
   process
   begin
     wait until clk'event and clk ='1';
     if (set ='0') then
          outp <='1';
     elsif(clr ='0') then
          outp <='0';
     else
          outp <= data;
     end if;
   end process;
end imp_fsm_bascule;
```

Synchronisation de la machine avec l'horloge "clk"

figure 3.7 Une bascule D avec mise à 0 et 1 modélisée comme une machine à état implicite

3.3.2 Modélisation de machine à état explicite

Dans une description de machine à état explicite, nous devons utiliser une variable ou un signal qui représente le registre d'état pour contenir explicitement l'état courant, et contrôler la séquence des états en faisant des affectations explicites à ce registre. Le contrôle de la machine est décrit en utilisant un énoncé *case*: le registre

d'état est utilisé comme variable de sélection, et chaque état du système est modélisé comme une des alternatives de l'énoncé *case*.

Un processus doit rencontrer les critères suivants pour implanter une machine à état explicite:

- Il contient un seul signal d'horloge, et le temps d'exécution d'un processus est inférieur ou égale à une période d'horloge.
- Nous utilisons une variable ou un signal du processus pour mémoriser l'état de la machine.

L'exemple dans la figure 3.4 montre le codage d'une machine à état explicite décrit par un processus. L'état de la machine est représenté par la variable d'état *state* qui peut prendre deux valeurs s0 et s1.

```
entity state_machine is
port(
    clk: in bit; -- horloge
    x: in bit; -- entrée
    rst: in bit; -- remise à l'état initial
    z: out bit); -- sortie
end state_machine;
```

```
architecture arch_exp_state_machine of state_machine is

 type states is (s0, s1);
 signal state: states:= s0;

-- process pour definir l'état suivant et les sortie
sm: process
begin
      wait on clk;
      if (clk'event and clk ='1') then
            if (rst ='0') then
                  state <= s0;
                  z <='0';
            elsif (rst ='1') then
                  case state is
                        when s0 => if (x ='1') then
                                          state <= s1;
                                          z <='1';
                                    else
                                          z<='0';
                                    end if;
                        when s1 => if (x ='1') then
                                          state <= s1;
                                          z <='1';
                                    else
                                          z<='0';
                                    end if;
                  end case;
            end if;
      end if;
end process sm;
end arch_exp_state_machine;
```

Synchronisation de la machine avec l'horloge "clk"

variable d'état explicite: state
Deux états de la machine: s0, s1

figure 3.8 Un exemple de modélisation de machine à état explicite par un seul processus

3.3.3 Modélisation de machines à état par un ensemble de processus

Ce modèle de description reflète la séparation des circuits séquentiels en circuit combinatoire et des registres. Il est décrit par un processus séquentiel et un certain

nombre de processus combinatoires. Le contrôle de la machine à état et les registres sont décrits par le processus séquentiel, et la logique combinatoire est décrite par un ou plusieurs processus combinatoires. Le processus séquentiel réalise la synchronisation de la machine avec l'horloge.

Les processus combinatoires sont activés chaque fois qu'un évènement survient sur un des signaux d'entrée ou d'état. Comme conséquence, les sorties et les signaux qui représentent l'état suivant sont calculés. La liste de sensitivité doit être composée de tous les signaux d'entrée du circuit et des signaux d'état.

Deux exemples de description utilisant deux processus se retrouvent dans la figure 3.9 et dans la figure 3.10. Dans l'exemple de la figure 3.9, le processus séquentiel réalise la synchronisation de la machine avec l'horloge *clk,* et décrit la fonction de transition. Le processus combinatoire décrit la fonction de sortie.

Lorsque les calculs d'état suivant (définition de la fonction de transition) ne sont pas spécifiés dans le processus séquentiel, la variable d'état courant doit être mémorisée par l'affectation d'une variable d'état suivant. Ce cas est illustré par la figure 3.10, où le processus séquentiel réalise la synchronisation de la machine avec l'horloge *clk* et affecte la variable d'état *state* avec la variable d'état suivant *next_state*. Le processus combinatoire décrit la fonction de sortie et la fonction d'état suivant.

Ce chapitre a d'abord présenté les modèles de description de circuit en VHDL, puis quelques aspects du sous-ensemble VHDL accepté, et enfin des styles de description de machine à état fini qui en découlent. Le chapitre suivant complète la description du modèle MDG amorcée dans la section 2.3, du langage cible.

```
entity state_machine is
port(
    clk: in bit;
    x: in bit;
    rst: in bit;
    z: out bit);
end state_machine;
```

Déclaration d'entité

```
architecture arch_multp_state_machine of state_machine is

type states is (s0, s1);
signal state: states:= s0;

-- process pour définir l'état suivant et la sortie
sm: process
begin
    wait on clk;
    if (clk'event and clk ='1') then
        if (rst ='0') then
            state <= s0;
            z <='0';
        elsif (rst ='1') then
            case sate is
                when s0 => if(x ='1') then
                                state <= s1;
                           end if;
                when s0 => if(x ='1') then
                                state <= s1;
                           end if;
            end case;
        end if;
    end if;
end process sm;
```

Processus séquentiel: réalise la synchronisation et affecte la variable d'état

```
-- Processus combinatoire. Il décrit les fonctions de transition et de sortie.
output: process (state, x)
begin
    case sate is
        when s0 => if(x ='1') then
                       z <='1';
                   else
                       z<='0';
                   end if;
        when s0 => if(x ='1') then
                       z <='1';
                   else
                       z<='0';
                   end if;
    end case;
end process output;
end arch_multp_state_machine;
```

Processus combinatoire affectant la variable de sortie

figure 3.9 Machine à état implantée par deux processus, la fonction de transition est réalisée par le processus séquentiel

```
entity state_machine is
port(
    clk: in bit;
    x: in bit;
    rst: in bit;
    z: out bit);
end state_machine;
```

Déclaration d'entité

```
architecture arch_multp_state_machine of state_machine is

type states is (s0, s1);
signal state, next_state: states:= s0;
begin
-- Processus séquentiel. Il définit la synchronisation de la
-- machine avec l'horloge et la remise à l'état initial
   clkd: process
   begin
     wait on clk;
     if (clk'event and clk ='1') then
          if (rst ='0') then
               state <= s0;
               z <='0';
          elsif (rst ='1') then
              state <= next_state;
      end if;
   end process clkd;
```

Processus séquentiel: réalise la synchronisation et affecte la variable d'état

```
-- Processus combinatoire. Il définit les fonctions de tran. et de sortie.
output_n_trans: process (state, x)
begin
      case sate is
           when s0 => if (x ='1') then
                              z <='1';
                              next_state <= s1;
                           else
                              z<='0';
                           end if;
           when s0 => if (x ='1') then
                              z <='1';
                              next_state <= s0;
                           else
                              z<='0';
                           end if;
      end case;
end process output;

end arch_multp_state_machine;
```

Processus combinatoire affectant la variable de sortie

figure 3.10 Machine à état implantée par deux processus, la fonction de transition est réalisée par le processus combinatoire

Chapitre IV

Le langage de description de matériels MDG-HDL

Le langage de description de matériels MDG-HDL supporte les descriptions structurales et les descriptions comportementales des machines à état abstrait. Une description structurale est une interconnexion de composants prédéfinis de la bibliothèque ou définis par l'utilisateur. Parmi les composants définis par l'utilisateur on peut distinguer, ceux dont le comportement est décrit par les relations de transition et de sortie de la machine à état abstrait associée, et ceux qui sont des boîtes noires représentées par des blocs de fonctions non-interprétés. Le contenu de ce chapitre est issu principalement de [30], [31], [32] et [33].

4.1 Fondement théorique du langage

Le fondement théorique du langage de MDG-HDL est une représentation efficace d'une classe d'expressions de la logique multi-sortée du premier ordre, par des graphes de décision multi-choix (*Multiway Decision Graphs* - MDG), une nouvelle forme de graphes de décision.

Après une définition des MDGs et des machines à état abstrait dans la section 2.3, nous présentons dans cette section l'ordonnancement des symboles, le partitionnement des relations de transition et de sortie, et l'interprétation des symboles des machines à état abstrait.

4.1.1 Ordonnancement des symboles dans les MDGs

L'ordonnancement des symboles est une des conditions de la structure MDG bien formée (*well-formedness condition*) [30]. Les étiquettes des arcs issus d'un noeud donné doivent apparaître dans un ordre de terme standard (*standard symbol order*) sans répétition. Le long de chaque chemin, les variables et les opérateurs croisés doivent apparaître dans un ordre sur mesure (*custom symbol order*), et les termes croisés ayant le même opérateur croisé doivent apparaître dans le même ordre standard.

L'ordre de symbole sur mesure est une généralisation (pour inclure les termes croisés) de l'ordre des variables utilisés dans les BDDs, et joue le même rôle. Il nécessite seulement de lister les constantes génériques, les variables, et les symboles fonctionnels qui composent les opérateurs croisés et les variables qui doivent apparaître comme des étiquettes de noeud. Il est choisi avec précaution pour chaque application particulière, de façon à obtenir des MDGs de taille convenable pour l'espace mémoire et le temps de calcul.

L'ordre de terme standard, compatible avec l'ordre de symbole sur mesure, est choisi une fois pour toute. De ces deux ordres, est défini un ordre des étiquettes de noeud parmi les variables et termes croisés comme suit: A vient avant B si et seulement si le symbole de plus haut niveau de A vient avant B dans l'ordre sur mesure, ou A et B sont des termes croisés de même opérateur croisé et A vient avant B dans l'ordre standard de terme. Les étiquettes de noeuds doivent apparaître dans l'ordre d'étiquette de noeud le long de chaque chemin.

Si un noeud N est étiqueté par une variable abstraite y, et une variable abstraite x apparaît dans un terme A qui étiquette un des arcs issus de N, alors x doit venir avant y dans l'ordre de symbole sur mesure. De façon similaire, si N est étiqueté par un terme croisé A avec un opérateur croisé f, et x est une variable abstraite qui apparaît dans A, alors x doit venir avant f dans l'ordre de symbole sur mesure.

4.1.2 Partitionnement des relations de transition et de sortie

Soit $D = (X, Y, Z, Y', \eta, F_I, F_T, F_O)$ une machine à état abstrait comme décrite dans la section 2.3.4. Conceptuellement, il est possible de représenter la relation de transition F_T par un seul MDG N de type $X \cup Y \rightarrow Y'$, appelée relation de transition monolithique. De façon similaire, un seul MDG O de type $X \cup Y \rightarrow Z$ peut être utilisé pour représenter la relation de sortie. F_I peut aussi être représenté par un MDG unique S_I. Toutefois en pratique, une relation de transition monolithique, soit est trop grande pour tenir en mémoire, soit prend trop de temps à construire. En conséquence, on utilise une représentation alternative connue comme relation de transition partitionnée conjonctive (*Conjunctive Partitioned Transition Relations* - CPTR), constituée d'une conjonction des partitions de la relation de transition monolithique initiale, qui lui est équivalente.

Pour chaque variable $y'_i \in Y' (1 \leq i \leq n)$, il existe une fonction de transition individuelle δ_i définissant la relation de transition individuelle:

$$y'_i = \delta_i(X,Y)$$

Cette relation de transition individuelle est reliée à la relation de transition totale F_T comme suit:

$$(x, y, y'_i) \in F_T \Leftrightarrow y'_i = \delta_i(X,Y)$$

où $x \in X$ et $y \in Y$. La relation de transition (totale) F_T peut-être représentée comme la conjonction de toutes les relations de transition individuelles.

$$F_T: \ ((y'_1 = \delta_1(X,Y)) \wedge \ldots \wedge (y'_n = \delta_n(X,Y)))$$

Ceci conduit à la représentation des CPTR où chaque bloc de partition contient une relation de transition individuelle ou plus.

La même méthode de partitionnement est applicable à la relation de sortie.

4.1.3 Interprétation des symboles des machines à état abstrait

Traditionnellement les machines à état fini modélisent les circuit séquentiels synchrones. La modélisation des circuit séquentiels synchrones par des machines à état abstrait dans l'approche MDG, a permis d'accroître les performances de la vérification formelle. Le lien entre ces deux types de machines à état est réalisé par une interprétation définie ci-dessous.

Une interprétation ψ affecte une dénotation (*denotation*) à chaque symbole de sort, constante et fonction de la machine à état abstrait, et satisfait les conditions suivantes:

1. La dénotation $\psi(\alpha)$ d'un sort abstrait α est un ensemble non vide.
2. Si α est un sort concret avec l'énumération $\{a_1, \ldots, a_n\}$ alors
   $\psi(\alpha) = \{\psi(a_1), \ldots, \psi(a_n)\}$ et $\psi(a_i) \neq \psi(a_j)$ pour $1 \leq i < j \leq n$.
3. Si f est un symbol de fonction de type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$ alors $\psi(f)$ est une fonction du produit cartésien $\psi(\alpha_1) \times \ldots \times \psi(\alpha_n)$ vers $\psi(\alpha_{n+1})$. En particulier, si $n = 0$ c'est-à-dire f est une constante générique de sort α_1, $\psi(f) \in \psi(\alpha_1)$.

Si X est un ensemble de variables, une affectation de variable avec le domaine X compatible avec une interprétation ψ est une fonction ϕ qui fait correspondre à chaque variable $x \in X$ de sort α à un élément $\phi(x)$ de $\psi(\alpha)$. Nous notons Φ_X^ψ l'ensemble des affectations compatibles à ψ aux variables de X. Nous noterons $\psi, \phi \models P$, la validité d'une formule P sous une interprétation ψ et une affectation ϕ de variable compatible avec ψ à des variables libres de P, et $\models P$ si $\psi, \phi \models P$ est satisfaite pour tout ϕ et ψ.

Pour voir comment une DF peut représenter un ensemble, remarquons que l'affectation ϕ à une variable, avec le domaine V compatible avec une interprétation donnée ψ, peut être vue comme un vecteur de valeurs, indexé par les variables dans V. Soit P une DF de type $U \rightarrow V$, où U contient seulement des variables abstraites. P représente alors l'ensemble des vecteurs

$$Set_V^\psi(P) = \{\phi \in \Phi_V^\psi \mid \psi, \phi \models (\exists U)P\}.$$

Pour une machine à état abstrait définie par $D = (X, Y, Z, Y', \eta, F_I, F_T, F_O)$ décrite dans la section 2.3.4, il existe exactement une machine à état unique $M = (\Phi_X^\Psi, \Phi_Y^\Psi, \Phi_Z^\Psi, S_I, R_T, R_O)$ pour chaque interprétation ψ, avec:

- Φ_X^Ψ est l'ensemble de toutes les affectations compatibles avec ψ de toutes les variables dans X, l'alphabet d'entrée.
- Φ_Y^Ψ est l'ensemble des états.
- Φ_Z^Ψ est l'alphabet de sortie.
- $S_I = Set_Y^\Psi(F_I)$ est l'ensemble des états initiaux.
- $R_T = \{(\phi, \phi', \phi'') \in \Phi_X^\Psi \times \Phi_Y^\Psi \times \Phi_Z^\Psi \mid \psi, \phi \cup \phi' \cup (\phi'' \circ \eta^{-1}) \models F_T\}$ est la relation de transition de la machine à état.
- $R_O = \{(\phi, \phi', \phi'') \in \Phi_X^\Psi \times \Phi_Y^\Psi \times \Phi_Z^\Psi \mid \psi, \phi \cup \phi' \cup \phi'' \models F_O\}$ est la relation de sortie de la machine à état.

4.2 Description abstraite de machines à état

Cette section va présenter, une machine à état fini d'une version traditionnelle de calcul du plus grand commun diviseur (Greatest Common Divider - GCD) de deux nombres positifs p_1 et p_2, et une machine à état abstrait correspondante.

4.2.1 Spécification informelle et algorithme de calcul du GCD

Une version traditionnelle de calcul du plus grand commun diviseur (Greatest Common Divisor - GCD) de deux nombres positifs p_1 et p_2, est donnée par l'algorithme d'Euclide dans la figure 4.1. Elle procède par soustractions répétitives. Deux variables y_1 et y_2 sont initialisées avec les valeurs p_1 et p_2, puis répétitivement on affecte à la variable de plus grande valeur (parmi y_1 et y_2) la valeur absolue de leur différence, jusqu'a ce que les deux variables y_1 et y_2 aient la même valeur. Cette valeur est le plus grand commun diviseur.

```
fonction gcd_euclide(entier_positif y1 , y2 ) retourne entier_positif
  tantque ( y1 <> y2 ) faire
     Si y1 < y2 alors y2 := y2 - y1
     sinon y1 := y1 - y2

  retourner( y2 )
```

figure 4.1 Algorithme de calcul du plus grand commun diviseur

À partir de cette description au niveau comportemental, on peut déduire une description au niveau RTL. La section suivante présente une description RTL de la machine à état GCD.

4.2.2 Machine à état fini de GCD

La machine à état étant définie au transfert registre, les relations de transition et de sortie définissent respectivement les états suivants et les sorties à chaque cycle d'horloge.

La machine à état de GCD est définie par:

- L'alphabet d'entrée $X = \{x_0, x_1, x_2\}$ où x_1 et x_2 contiennent les valeurs des deux nombres positifs dont if faut calculer le plus grand diviseur et $x_0 \in \{0, 1\}$, lorsqu'elle a la valeur 0, indique que la machine est prête à recevoir de nouvelles valeurs.
- L'ensemble des variables d'état $Y = \{y_0, y_1, y_2\}$ où $y_0 \in \{0, 1\}$ sert à identifier deux états de la machine. Lorsque $y_0 = 1$, le processus de calcul d'un GCD est en cours, $y_0 = 0$ dans le cas contraire. y_1 et y_2 sont initialisées avec x_1 et x_2 , et contiennent les valeurs courantes des nombres intermédiaires pendant le processus de calcul.
- L'ensemble des variables d'état suivant $Y' = \{y'_0, y'_1, y'_2\}$,
- L'alphabet de sortie $Z = \{z_0\}$, où z_0 contient le GCD calculé.
- L'état initial de la machine est défini par $y_0 = 0$, $y_1 = x_1$ et $y_2 = x_2$
- La relation de transition est définie comme suit:
  - si $(x_0 = 1$ et $y_0 = 0)$ alors $\{y'_0 \leftarrow 1, y'_1 \leftarrow x_1, y'_2 \leftarrow x_2\}$
  - si $(y_0 = 1$ et $y_2 < y_1)$ alors $\{y'_0 \leftarrow 1, y'_1 \leftarrow y_1 - y_2, y'_2 \leftarrow y_2\}$
  - si $(y_0 = 1$ et $y_1 < y_2)$ alors $\{y'_0 \leftarrow \ , y'_1 \leftarrow y_1, y'_2 \leftarrow y_2 - y_1\}$

- si ($y_0 = 1$ et $y_1 = y_2$) alors $\{y'_0 \leftarrow 0, y'_1 \leftarrow x_1, y'_2 \leftarrow x_2\}$
- si ($x_0 = 0$ et $y_0 = 0$) alors $\{y_0 \leftarrow 0, y'_1 \leftarrow y_1, y'_2 \leftarrow y_2\}$

- La relation de sortie est définie par: si ($y_0 = 1$ et $y_1 = y_2$) alors $z_0 = y_1$. Dans tous les autres cas, la valeur du signal de sortie est quelconque.

La machine à état correspondante peut être représentée graphiquement comme montrée dans la figure dans la figure 4.2.

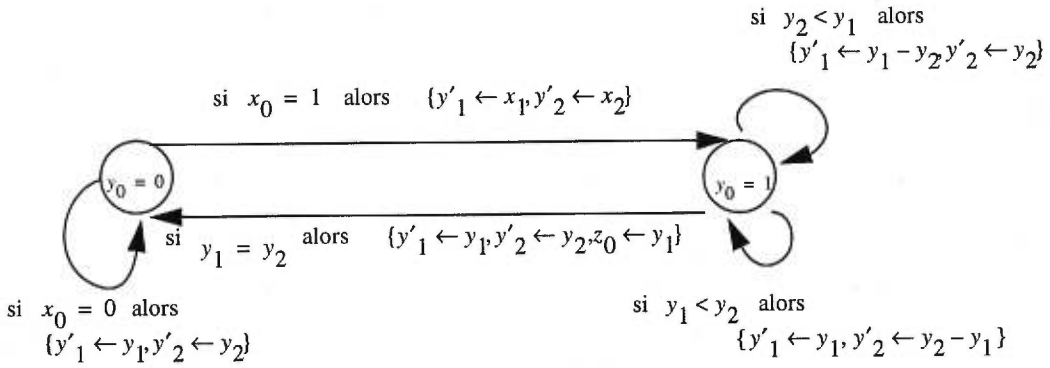


figure 4.2 Représentation graphique de la machine à état fini de GCD

4.2.3 Description abstraite de la machine de GCD

Les MDGs sont une représentation canonique et compacte des formules directes définies dans la section 2.3.4. Chaque élément participant à la définition de la machine à état abstrait (les ensembles de variables d'entrée, d'état, d'état suivant, de sortie des états initiaux, la fonction de correspondance entre les variables d'état et les variables d'état suivant, et les relations de transition et de sortie) est représenté par une formule directe d'où l'on déduit le MDG correspondant.

4.2.3.1 Interprétation des symboles abstrait de GCD

Cette section définit l'interprétation des symboles de la machine à état abstrait de GCD.

- Le sort concret *bool* est l'ensemble des booléens défini par l'énumération de ses valeurs, $\{0, 1\}$,
- Le sort abstrait *NUM* est l'ensemble des entiers positifs,
- La fonction abstraite *sub* de type $NUM \times NUM \rightarrow NUM$ représente la soustraction,
- L'opérateur croisé *eq* de type $NUM \times NUM \rightarrow BOOL$ représente l'opérateur d'égalité,
- L'opérateur croisé *lt* de type $NUM \times NUM \rightarrow BOOL$ représente l'opérateur inférieur.

4.2.3.2 Description abstraite des ensembles

Les ensembles de variables d'entrée, d'état, d'état suivant et de sortie de la machine à état GCD sont respectivement:

- $X = \{x_0, x_1, x_2\}$, avec x_0 concret de sort *bool*, x_1 et x_2 de sort abstrait NUM,
- $Y = \{y_0, y_1, y_2\}$, avec y_0 concret de sort *bool*, y_1 et y_2 de sort abstrait NUM,
- $Y' = \{y'_0, y_1, y_2\}$, avec y'_0 concret de sort *bool*, y'_1 et y'_2 de sort abstrait NUM,
- $Z = \{z_0\}$, avec z_0 abstrait de sort NUM.

Chaque ensemble de variables est représenté par un MDG. Par exemple, pour représenter l'ensemble des vecteurs d'entrée, nous utilisons la formule

$$(\exists w_3)(\exists w_4)((x_1 = w_3) \wedge (x_2 = w_4))$$

où w_3 et w_4 sont des variables auxiliaires de sort abstrait NUM. Le MDG correspondant est représenté comme suit:

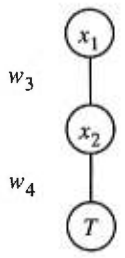


figure 4.3 Représentation MDG de l'alphabet d'entrée de la machine à état abstrait GCD

4.2.3.3 L'ensemble des états initiaux

Il peut avoir soit un seul état initial où toutes les variables d'état prennent la valeur 0, ceci peut être décrit par la formule $(y_0 = 0) \wedge (y_1 = zero) \wedge (y_2 = zero)$, où zéro est une constante abstraite générique. La figure 4.4 représente le MDG de cet état initial, de la machine à état GCD. Soit on peut prendre comme état initial n'importe quel état parmi les états décrits par la formule $(\exists w_1)(\exists w_2)((y_0 = 0) \wedge (y_1 = w_1) \wedge (y_2 = w_2))$, où w_1 et w_2 sont des variables auxiliaires de sort abstrait *NUM*. Cette formule est logiquement équivalente à $y_0 = 0$, mais les valeurs symboliques de y_1 et y_2 sont requises pour l'exploration d'état.

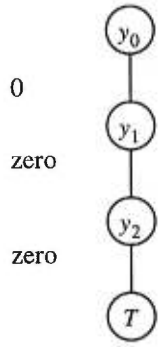


figure 4.4 Représentation MDG d'un état initial de la machine à état GCD

4.2.3.4 La relation de transition

La fonction de transition (totale) de la machine à état GCD (dans la figure 4.2) est représentée par la forme normale disjonctive ci-dessous,

$$
\begin{aligned}
&((y_0 = 0) \wedge (x_0 = 0) \wedge (y'_0 = 0) \wedge (y'_1 = y_1) \wedge (y'_2 = y_2)) \vee \\
&((y_0 = 0) \wedge (x_0 = 1) \wedge (y'_0 = 1) \wedge (y'_1 = x_1) \wedge (y'_2 = x_2)) \vee \\
&((y_0 = 1) \wedge (eq(y_1, y_2) = 0) \wedge (lt(y_1, y_2)) = 0) \wedge \\
&(y'_0 = 1) \wedge (y'_1 = sub(y_1, y_2))) \wedge (y'_2 = y_2)) \vee \\
&((y_0 = 1) \wedge (eq(y_1, y_2) = 0) \wedge (lt(y_1, y_2) = 1) \wedge \\
&(y'_0 = 1) \wedge (y'_1 = y_1) \wedge (y'_2 = sub(y_1, y_2))) \vee \\
&((y_0 = 1) \wedge (eq(y_1, y_2) = 1) \wedge (y'_0 = 0) \wedge (y'_1 = y_1) \wedge (y'_2 = y_2))
\end{aligned}
$$

où chaque opérande de la disjonction représente une transition. Par exemple le premier opérande de la disjonction,

$$((y_0 = 0) \wedge (x_0 = 0) \wedge (y'_0 = 0) \wedge (y'_1 = y_1) \wedge (y'_2 = y_2))$$

représente la boucle sur l'état repérée par $y_0 = 0$ dans la représentation de la machine à état de la figure 4.2.

De cette relation de transition totale, on peut déduire trois relations de transition individuelles correspondant à chacune des variables d'état suivant. Par exemple, la relation de transition individuelle de la variable d'état y'_0 est définie comme suit:

$$
\begin{aligned}
&((y_0 = 0) \wedge (x_0 = 0) \wedge (y'_0 = 0)) \vee \\
&((y_0 = 0) \wedge (x_0 = 1) \wedge (y'_0 = 1)) \vee \\
&((y_0 = 1) \wedge (eq(y_1, y_2) = 0) \wedge (y'_0 = 1)) \vee \\
&((y_0 = 1) \wedge (eq(y_1, y_2) = 1) \wedge (y'_0 = 0))
\end{aligned}
$$

La figure 4.5 représente les trois MDGs δ_{y_0}, δ_{y_1} et δ_{y_2} des relations de transition individuelles des variables respectives de y'_0, y'_1 et y'_2. Chaque terme de la disjonction d'une relation de transition individuelle, représente un chemin entre la racine et la feuille du MDG correspondant. Par exemple

$$((y_0 = 0) \wedge (x_0 = 0) \wedge (y'_0 = 0))$$

correspond à un chemin dans δ_{y_0} dont les arcs sont représentés en pointillé.

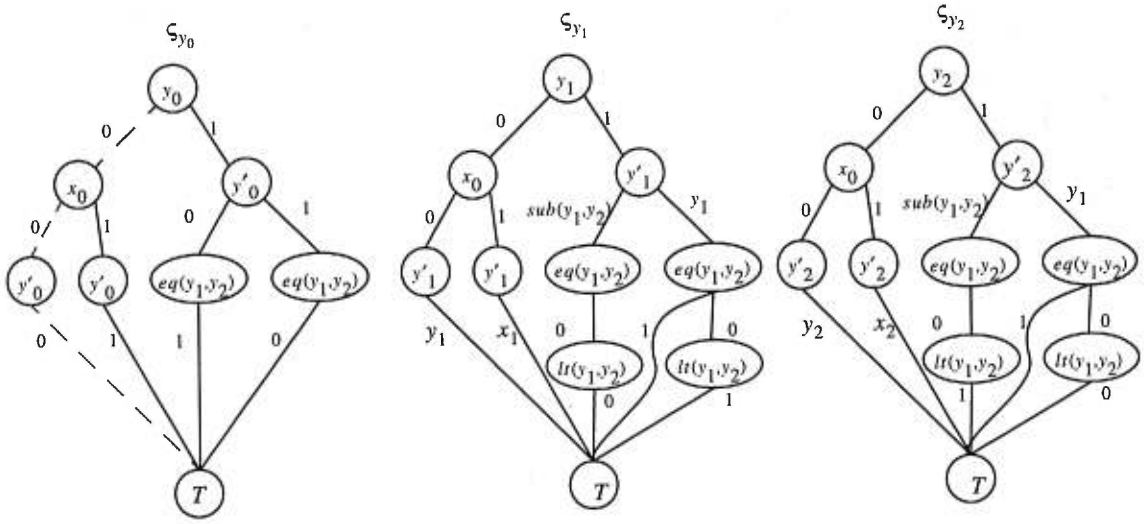


figure 4.5 Représentation MDG de la relation de transition de la machine à état GCD

4.2.3.5 La relation de sortie

Considérons la seule sortie de la machine à état la variable z_0, la relation de sortie lie la variable de sortie z_0 et les variables d'entrée et d'état. La relation des sortie peut-être décrite par la formule ci-dessous:

$$((y_0 = 0) \wedge (z_0 = y_1)) \vee$$
$$((y_0 = 1) \wedge (eq(y_1,y_2) = 0) \wedge (z_0 = zero)) \vee$$
$$((y_0 = 1) \wedge (eq(y_1,y_2) = 1) \wedge (z_0 = y_1))$$

Cette relation de sortie ne comporte que la variable de sortie z_0, et est par conséquent elle-même une relation de sortie individuelle. La représentation MDG de la relation de sortie individuelle de z_0 est donnée dans la figure 4.6. S'il y avait plusieurs variables de sortie, le principe de dérivation d'une relation individuelle de sortie à partir d'une relation totale de sortie serait le même que dans le cas d'une relation de transition.

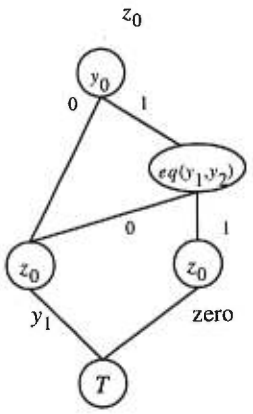


figure 4.6 Représentation MDG de la relation de sortie de machine à état GCD

4.3 Description de circuits numériques

Nous décrivons dans cette section le modèle MDG-HDL du circuit numérique modélisé par la machine à état abstrait de GCD. Ces descriptions sont une description textuelle des représentations MDG.

Le langage MDG-HDL permet de décrire aussi bien la spécification que l'implantation des circuits numériques. On distingue trois parties dans la description des circuits en MDG-HDL.

1. La section algébrique de la spécification. Elle comporte la déclaration des sorts, des fonctions et des constantes génériques.
2. La description du circuit représentant la machine à état, comporte les sections suivantes: de déclaration des signaux de la machine, des relations de transition individuelles, des relations de sortie individuelles, de déclaration de l'état initial de la machine, de définition de la fonction η, de déclaration des variables de sortie, du partitionnement de la relation des transition, et du partitionnement de la relation de sortie.
3. La liste ordonnée des symboles selon l'ordre sur mesure.

4.3.1 Exemple: spécification MDG-HDL de la machine à état GCD

Nous avons omis volontairement d'inclure dans la spécification algébrique et dans la description les en-têtes des fichiers décrits dans [33], pour améliorer la lisibilité de la présentation.

4.3.2 Section de la spécification algébrique de la machine à état GCD

La section algébrique de la spécification est illustrée dans la figure 4.7. Cette spécification ne mentionne pas la déclaration du sort *bool* qui est prédéfinie dans le fichier standard de la section algébrique de spécification.

```
%=============================================================
% Section de la spécification algébrique de la machine à état abstrait de GCD
%=============================================================
%
% Introduire ici l'en-tête du fichier de spécification algébrique
%

function(lt, [NUM, NUM], bool).

abs_sort(NUM).
function(sub, [NUM, NUM], NUM).
function(eq, [NUM, NUM], NUM).
gen_const(zero,NUM).
```

figure 4.7 Section algébrique de la spécification de la machine à état GCD

4.3.3 Spécification de la machine à état abstrait

La figure 4.8 présente une partie de la description de la spécification de la machine à état abstrait. Certaines parties de la description ont été omises pour simplifier la présentation et faire ressortir son organisation générale.

Chaque relation de transition (ou de sortie) individuelle est représentée sous forme tabulaire. Nous décrivons ci-dessous, comment obtenir la représentation tabulaire d'une relation de transition (ou de sortie) individuelle à partir d'une formule directe. Pour illustrer le principe de la dérivation, considérons la formule directe représentant la relation de sortie individuelle correspondant à la variable z_0 ci dessous représentée.

$$((y_0 = 0) \wedge (z_0 = y_1)) \vee$$
$$((y_0 = 1) \wedge (eq(y_1,y_2) = 0) \wedge (z_0 = zero)) \vee$$
$$((y_0 = 1) \wedge (eq(y_1,y_2) = 1) \wedge (z_0 = y_1))$$

Les éléments de l'en-tête du tableau sont constitués par l'union des membres de gauche des équations contenues dans la formule de la relation de sortie individuelle. Le membre de gauche correspondant à la variable d'état ou à la variable de sortie est placée dans la dernière colonne. L'en-tête du tableau de la relation de sortie correspondant à z_0 est par conséquent $\{y_0, eq(y_1,y_2), z_0\}$.

Chaque ligne du corps du tableau est définie par un des termes liés par l'opérateurs de disjonction de la relation de sortie (ou de transition) individuelle. Trois termes sont liés par l'opérateur de disjonction, dans la formule directe de la relation de sortie de z_0. Ce sont:

$$((y_0 = 0) \wedge (z_0 = y_1)),$$

$$((y_0 = 1) \wedge (eq(y_1,y_2) = 0) \wedge (z_0 = zero)) \text{ , et}$$

$$((y_0 = 1) \wedge (eq(y_1,y_2) = 1) \wedge (z_0 = y_1)).$$

Les éléments composant une ligne du corps du tableau sont définis pour chaque colonne de la façon suivante. Si l'élément de l'en-tête de la colonne est le membre de gauche d'une équation du terme, on inscrit dans la colonne, le membre de droite de l'équation, sinon on inscrit *. Par exemple, considérons la représentation d'une ligne de tableau de la variable z_0 définie par le terme $((y_0 = 0) \wedge (z_0 = y_1))$. On marque 0 dans la colonne correspondant à l'entête y_0, * dans la colonne correspondant à l'entête $eq(y_1, y_2)$, et y_1 dans la colone correspondant à l'entête z_0. On remplit de

manière identique les deux lignes suivantes et on obtient le deuxième tableau dans la figure 4.8.

```
% ============================================================
% Spécification de la machine à état GCD
%============================================================
% Inclure ici l'en-tête de fichier de description de circuit

%--- déclaration des signaux --
% La déclaration des signaux x1 et x2 à inclure ici (omise pour simplifier la présentation)
signal(x0, bool).
signal(y2, NUM).
signal(z0, NUM).

% la descript. des composants y0 et y2 à inclure ici (omise pour simplifier la présentation)
% -- n_y1 est la variable d'état suivant de la variable y1
component(y1_comp, table([[y0, x0, eq(y1,y2), lt(y1,y2), n_y1],
                          [0, 0, *, *, y1],
                          [0, 1, *, *, x1],
                          [1, *, 0, 0, sub(y1,y2)],
                          [1, *, 0, 1, y1],
                          [1, *, 1, *, y1]])).

component(z0_comp, table([[y0, eq(y1,y2), z0],
                          [0, *, y1],
                          [1, 0, zero],
                          [1, 1, y1]])).

%--- état initial de la machine à état ---
init_var(init_y1,NUM).
init_var(init_y2,NUM).

init_val(y0,0).
init_val(y1, init_y1).
init_val(y2, init_y2).

% Sortie
outputs([z0]).

%---Partitions de la relation de sortie ---
output_partition([[[z0]]]).

% -- Partition de la relation de transition
next_state_partition([[[n_y0]], [[n_y1]], [[n_y2]]]).

%Correspondance entre la var. d'état et la var. d'état suivant
st_nxst(y0, n_y0).
st_nxst(y1, n_y1).
st_nxst(y2, n_y2).
```

Relation individuelle de transition de la variable y1

Relation individuelle de sortie de la variable z0

Description de l'état initial de la machine

Description de la fonction de correspondance entre état et état suivant

figure 4.8 Spécification MDG-HDL de la machine à état GCD

4.3.4 Liste des symboles ordonnés

La figure ci-dessous fournit une liste ordonnée des symboles de la description MDG-HDL de la machine à état. L'ordre est conforme à l'ordre de symbole sur mesure. Le principe de détermination de l'ordre des symboles est donné dans la section 7.5.1.

```
%=========================================================
% Liste des symboles ordonnés de la spécification de la machine à état GCD
%=========================================================
order_main([
x0,
x1,
x2,
signal_zero,
y0,
y1,
y2,
zero,
n_y0,
n_y1,
n_y2,
z0,
eq,
lt,
sub
]).
```

figure 4.9 Liste des symboles ordonnés sur mesure de la description abstraite de la machine à état GCD

Nous avons présenté dans ce chapitre le fondement théorique du langage de description MDG-HDL, puis nous avons décrit la machine à état fini du GCD et la machine à état abstrait correspondante, enfin nous avons terminé en donnant la description du circuit numérique modélisé par la machine à état abstrait.

Bien que les langages cible et source décrivent des machines à état, les structures des langages sont totalement différentes. Nous établissons un lien entre ces deux types de machines à état, via une sémantique opérationnelle du langage VHDL, décrite dans le chapitre suivant.

Chapitre V

Sémantique opérationnelle des machines à état VHDL

5.1 Introduction

Les descriptions comportementales dans notre langage source sont basées, sur des processus définis essentiellement par des énoncés de flot de contrôle et d'assignation. Les descriptions comportementales dans le langage cible sont réalisées, à l'aide de relations de transitions et de sorties individuelles implantées, par des composants représentés sous une forme tabulaire.

La grande différence de la structure des langages cible et source, nous amène à baser la traduction sur la sémantique des descriptions comportementales plutôt que sur leur syntaxe. La sémantique opérationnelle qui décrit la signification des langages de programmation, en spécifiant comment il exécute une machine théorique nous paraît la mieux adaptée, pour décrire le comportement des machines à états VHDL.

Ce chapitre fait le lien entre le texte source statique d'une machine à état, et les actions qui doivent être associées à l'exécution du code source, pour implanter cette machine. La mise en oeuvre du comportement implanté par la machine à état dépend de la sémantique du langage VHDL.

La sémantique opérationnelle associe à un code dans un langage, une machine théorique (que nous appelons automate), dont l'exécution implante le comportement décrit par ce code. Nous représentons dans ce chapitre l'exécution, des énoncés séquentiels, des séquences de ces énoncés, des énoncés de processus d'une machine à état, par des automates. Ce chapitre fournit une définition et une méthode de construction de ces automates finis dans lesquels, les transitions sont étiquetées par, les conditions de transition portant sur les signaux d'entrée et les variables et signaux

internes, et les énoncés d'affectation.

La génération des automates finis pour un modèle comportemental de VHDL se fait par composition. Nous attachons un sous-modèle aux énoncés séquentiels des processus, puis composons ces sous-modèles en sous-modèle pour chaque processus. Le modèle de comportement de chaque machine à état est constitué des sous-modèles de comportement de chaque processus. Le modèle exige qu'une description VHDL hiérarchique soit aplatie, avant de construire un automate pour chaque processus.

5.2 Modélisation du langage source

Nous nous basons sur la modélisation des langages impératifs de [29], pour modéliser la description comportementale dans le langage VHDL. Nous avons élargi ce modèle en modifiant la définition de l'état de l'automate, en vue d'inclure l'énoncé *null* nécessaire pour la modélisation de certains énoncés, par exemple *wait*. Le langage source VHDL est modélisé par un tuple dont les éléments sont les ensembles suivants:

- l'ensemble des entiers naturels N,
- l'ensemble des valeurs de vérité $T = \{vrai, faux\}$,
- un ensemble d'objets composés de signaux et variables Obj,
- un ensemble des expressions arithmétiques $Aexp$,
- un ensemble des expressions booléennes $Bexp$,
- un ensemble d'énoncés de VHDL E.

5.2.1 Évaluation d'expressions concrètes

Le résultat de l'évaluation d'une expression influence l'exécution des énoncés de contrôle de flot. L'état de l'automate à un instant donné peut être représenté par tous les couples d'objets et leur contenu et le numéro d'ordre séquentiel du dernier énoncé exécuté seq, soit $((x_1,\sigma(x_1)),\ldots,(x_n,\sigma(x_n)), seq)$ avec $\sigma : Obj \rightarrow N$ où $\sigma(x_i)$ est la valeur ou le contenu de l'objet x_i dans l'état σ, $seq = 0$ à l'état initial, $seq \neq 0$ pour tout autre état.

À la différence de la définition de l'état dans [29], l'état que nous avons défini

contient un numéro d'ordre séquentiel, de sorte que l'exécution de l'énoncé *null* qui ne change aucune valeur d'objet, change l'état de l'automate.

On peut représenter la situation où une expression arithmétique a est en attente d'évaluation dans un état σ par la paire $\langle a, \sigma \rangle$. On peut définir une relation d'évaluation entre cette paire et des nombres par

$$\langle a, \sigma \rangle \rightarrow n$$

signifiant qu'une expression arithmétique a dans un état σ s'évalue à n. Les paires $\langle a, \sigma \rangle$, sont appelées les configurations des expressions arithmétiques.

Comme précédemment, on peut représenter la situation où une expression booléenne b est en attente d'évaluation dans un état σ par la paire $\langle b, \sigma \rangle$. On peut définir une relation d'évaluation entre cette paire et l'ensemble des valeurs $\{vrai, faux\}$

$$\langle b, \sigma \rangle \rightarrow t \in \{vrai, faux\}$$

Les règles d'évaluation des expressions arithmétiques et booléennes sont spécifiées dans [29].

5.2.2 Exécution des énoncés

Définissons la relation $\langle e, \sigma \rangle \rightarrow \sigma'$ comme l'exécution d'un énoncé e dans l'état σ pour aboutir à l'état σ'. L'exécution évalue éventuellement une expression conditionnelle et change l'état de l'automate.

L'exécution d'un énoncé séquentiel VHDL, se traduit par l'exécution d'un certain nombre d'opérations définies par la signification de chaque énoncé, par exemple l'évaluation d'une condition et le changement de la valeur d'un objet.

La représentation des énoncés séquentiels par des automates amène à décomposer les opérations associées à certains énoncés. Cette décomposition peut donner lieu parfois à des opérations factices, qui font changer d'état sans changer la valeur d'un objet. Nous noterons τ, l'énoncé *null* dont l'opération associée est factice.

5.2.3 Représentation de l'exécution des énoncés

L'exécution des énoncés du langage VHDL est représentée par des automates définis par des quintuplets suivants [9]:

$$A = (S, T, s_{start}, s_{end}, \Sigma)$$

- S est un ensemble fini d'états divisé en sous-ensembles S_{local} et S_{sync} où S_{sync} est l'ensemble des états de synchronisation dans lesquels les valeurs effectives des signaux sont mis à jour.
- s_{start} et s_{end} sont respectivement l'état initial et l'état final de l'automate.
- T est un ensemble fini de transitions,

$$T \subset S \times S \times B_{ex} \times E$$

- Par exemple $t = (s_{start}, s_{end}, b_{ex}, e) \in T$ est une transition de s_{start} vers s_{end} quand la condition b_{ex} est satisfaite, qui exécute l'énoncé e.
- $\Sigma = (Obj, Type, Init)$ est l'espace des données où *Obj* est l'ensemble des objets (variables et signaux), *Type* est l'ensemble des types, Et *Init* est l'ensemble des valeurs initiales.
- E est l'ensemble des énoncés.

N.B. Sur tous les automates, nous allons inscrire sur les arcs des automates représentant chaque transition un couple (au lieu d'un quadriplet définissant une transition), dont le premier élément est la condition et le deuxième est l'opération associée à l'énoncé. Quant aux états initial et final de la transition, ils sont clairement identifiables sur les schémas.

Des règles décrivant chaque exécution ont une prémisse et une conclusion, et sont définies sous la forme $\frac{premisse}{conclusion}$, où *premisse*, soit est une condition à remplir, soit décrit l'exécution des énoncés composant l'énoncé dont nous définissons les règles d'exécution, et *conclusion* définit les états initial et final de l'automate et l'énoncé concerné.

Les relations d'exécution de l'assignation des variables et signaux, des énoncés de contrôle de flot, du séquencement des énoncés, et des énoncés de processus sont définies dans la section 5.3.

5.2.4 Exécution d'un modèle

L'élaboration d'un design crée un ensemble de processus interconnectés. Cet ensemble de processus peut être exécuté pour simuler le comportement du design. La simulation du comportement nécessite l'exécution des processus définis par l'utilisateur, qui interagissent entre eux et avec l'environnement.

Le processus noyau est une représentation conceptuelle de l'agent, qui coordonne les activités des processus définis par l'utilisateur pendant l'exécution. Cet agent provoque, la propagation des valeurs des signaux et la mise à jour des signaux implicites (tel que *S'Stable(T)*). En plus, ce processus est responsable de la détection des évènements qui surviennent, et provoque l'exécution des processus appropriés en réponse à ces évènements [12].

L'exécution d'un modèle consiste en une phase d'initialisation suivie d'exécutions répétitives des processus décrivant le modèle. Chaque cycle de simulation est composé, d'une alternance d'exécution en parallèle des processus définis par l'utilisateur, et de la mise à jour des signaux effectuée par le processus noyau, lorsque tous les processus définis par l'utilisateur sont suspendus. Le principe d'un processus de simulation est présenté dans la figure 5.1. La mise à jour des signaux est appelée synchronisation des processus définis par l'utilisateur avec l'environnement extérieur.

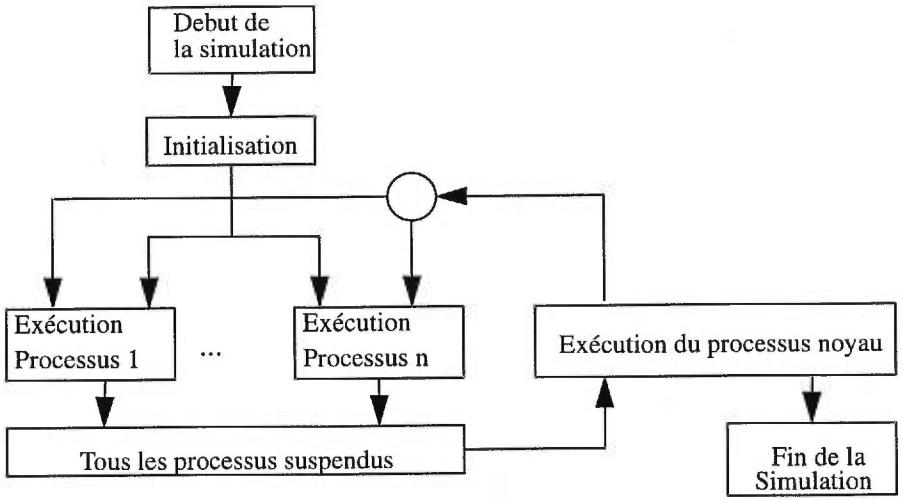


figure 5.1 Processus de simulation

5.3 Automates pour énoncés séquentiels

La modélisation des énoncés séquentiels par des automates est décrite dans [8]. Nous avons adapté, la sémantique opérationnelle des langages impératifs définie dans [29] à la modélisation des énoncés séquentiels du langage VHDL, et les modèles de composition des processus à nos spécifications de machines à état. Nous avons également contribué à la dérivation des comportements des machines à état à partir de l'automate du modèle VHDL décrite dans la section 5.6.

5.3.1 Enoncé *wait*

Soit l'énoncé *wait on S until expr* où *expr* se réfère au signal *S* et à l'attribut prédéfini *event*. L'exécution de cet énoncé peut être représentée par un automate ayant un ensemble $S = \{s_{start}, s_{end}, s_{sync}\}$ de trois états, et trois transitions dont les expressions booléennes associées sont définies par les fonctions booléennes ψ_1 et ψ_2 comme suit:

- ψ_1 est évaluée à vrai si la valeur de S a changé depuis la précédente synchronisation.
- $\psi_2 \equiv \exp r$

L'exécution de l'énoncé *wait* se termine si après une synchronisation, un évènement est survenu sur le signal S (*S'event* est valide) et la condition *expr* est évaluée à *vrai*. Ceci est modélisé par la transition $(s_{sync}, s_{end}, \psi_1 \wedge \psi_2, \tau)$ qui fait passer de l'état de synchronisation s_{sync} à l'état s_{end}, si la condition $\psi_1 \wedge \psi_2$ est valide. La transition $(s_{sync}, s_{start}, \neg(\psi_1 \wedge \psi_2), \tau)$ modélise le fait que le processus continue d'être suspendu si la valeur de S n'a pas changé depuis la dernière la dernière synchronisation ou que *expr* n'est pas valide.

La figure 5.2 représente l'automate d'énoncé séquentiel $A^{wait\ on\ S}$ de l'énoncé "wait on S until expr".

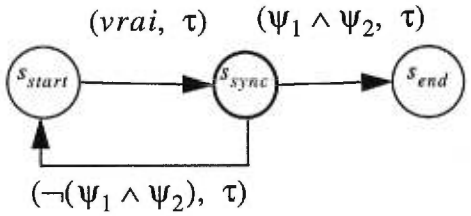


figure 5.2 Automate $A^{wait\ on\ S}$ de l'énoncé séquentiel "wait on S"

La relation d'exécution de l'énoncé *wait* est décrite par les deux règles suivantes.

$$\frac{\langle \psi_1 \wedge \psi_2, s_{sync} \rangle \rightarrow faux}{\langle wait\ on\ S\ until\ \exp r, s_{start} \rangle \rightarrow s_{start}} \qquad (5.1)$$

$$\frac{\langle \psi_1 \wedge \psi_2, s_{sync} \rangle \rightarrow vrai}{\langle wait\ on\ S\ until\ \exp r, s_{start} \rangle \rightarrow s_{end}} \qquad (5.2)$$

La règle d'exécution 5.1 énonce que, si la condition $\psi_1 \wedge \psi_2$ est non valide

dans l'état s_{sync}, alors l'énoncé *wait on S until expr* reste dans l'état de départ s_{start}, et le processus est suspendu. La règle d'exécution 5.2 énonce que, si la condition $\psi_1 \wedge \psi_2$ est valide dans l'état s_{sync}, alors l'énoncé *wait on S until expr* passe dans l'état final s_{send}, et la suspension du processus est levée.

5.3.2 Énoncé d'assignation de signal

Soient σ un état, $m \in N$ et $X \in Obj$. Nous noterons $\sigma[m/X]$ l'état obtenu à partir de σ en remplaçant le contenu de X par m, plus précisément:

$$\sigma[m/X](Y) = \begin{cases} m & si \;\; Y = X \\ \sigma(Y) & si \;\; Y \neq X \end{cases}$$

L'opération ci-dessus, décrit une assignation. Nous ne distinguons pas dans notre modèle les assignations des variables et des signaux. Les valeurs effectives des signaux étant mises à jour dans les états de synchronisation par le processus noyau, nous ne considérerons les valeurs des expressions que dans ces états où tous les objets contiennent leur valeur effective.

La figure 5.3 montre l'automate d'énoncé séquentiel $A^{delta-delay\ sig\ assign}$ généré pour un énoncé d'assignation de délais delta $S <=$ *expr*. L'état final est $s_{start}[m/S]$, où m est l'évaluation de *expr*.

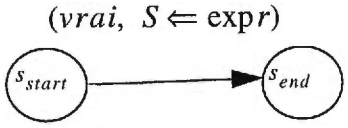


figure 5.3 Automate d'énoncé séquentiel $A^{delta-delay\ sig\ assig}$

La règle d'exécution correspondante est représentée comme suit:

$$\frac{\langle \text{expr}, s_{start} \rangle \rightarrow m}{\langle S \Leftarrow \text{exp} \;\; , s_{start} \rangle \rightarrow s_{start}[m/S]} \qquad (5.3)$$

Elle signifie que, si l'évaluation de *expr* produit m dans l'état s_{start}, alors l'exécution de l'énoncé S <= *expr* mène à un état qui dérive de l'état s_{start} par remplacement de la valeur de S par m.

La mise à jour des signaux étant réalisée par le processus noyau, elle n'apparaît pas dans les modèles des éléments décrivant les processus définis par l'utilisateur. Notre algorithme se charge de simuler le délai, en mettant en attente la mise à jour des signaux qui est réalisée, quand on rencontre un énoncé de synchronisation (*wait*).

5.3.3 Enoncés conditionnels

L'automate $A^{if\ cond\ then\ c_0\ else\ c_1}$ généré pour l'énoncé "*if cond then* c_0 *else*" est représenté dans la figure 5.4. Il exprime les deux alternatives d'exécution selon l'évaluation de l'expression booléenne *cond.* Si l'expression booléenne *cond* est satisfaite, l'état suivant est s_{end1} et l'énoncé c_0 est exécutée. Si l'expression booléenne *cond* est non valide, l'état suivant est s_{end2} et l'énoncé c_1 est exécutée.

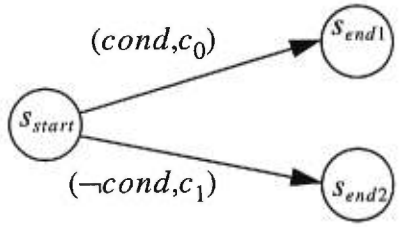


figure 5.4 Automate $A^{if\ cond\ then\ c_0\ else\ c_1}$

Les règles d'exécution sont définies par les deux règles suivantes:

$$\frac{\langle cond, s_{start}\rangle \rightarrow vrai \quad \langle c_0, s_{start}\rangle \rightarrow s_{end1}}{\langle if\ \ cond\ \ then\ \ c_0\ \ else\ \ c_1, s_{start}\rangle \rightarrow s_{end1}} \qquad (5.4)$$

$$\frac{\langle cond, s_{start}\rangle \rightarrow faux \quad \langle c_1, s_{start}\rangle \rightarrow s_{end2}}{\langle if\ \ cond\ \ then\ \ c_0\ \ else\ \ c_1, s_{start}\rangle \rightarrow s_{end2}} \qquad (5.5)$$

La règle d'exécution 5.4 spécifie que, si la condition *cond* est évaluée à vrai et que l'exécution de la commande c_0 dans l'état s_{start} conduit à l'état s_{end1}, alors l'exécution de la commande "*if cond then* c_0 *else* c_0" conduit à l'état s_{end1}. La règle 5.5 exprime l'autre alternative sur le même principe.

L'automate de l'énoncé *case* et les règles d'exécution correspondantes sont construites selon un principe semblable.

5.3.4 Séquence d'énoncés séquentiels

L'automate d'énoncé séquentiel (AES) d'une séquence de deux énoncés séquentiels AES1 et AES2, est fondamentalement construit en identifiant l'état final du premier avec l'état initial du second énoncé. La figure 5.5 illustre ce type de construction.

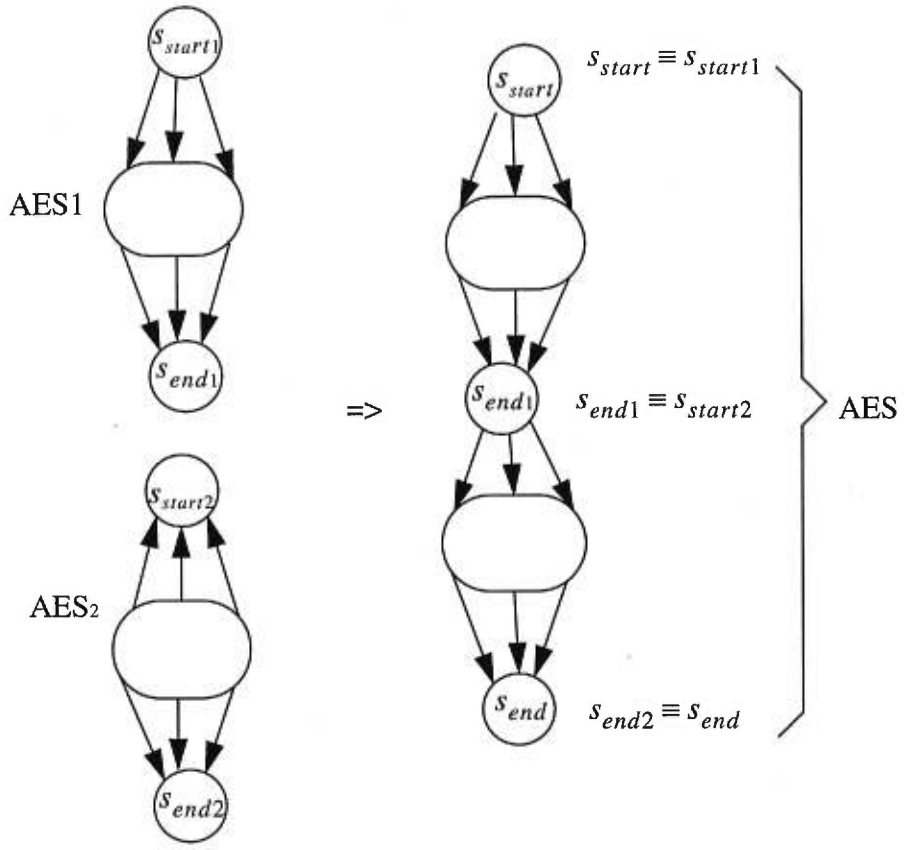


figure 5.5 Automate d'énoncé séquentiel d'une séquence de deux énoncés

La règle d'exécution pour la séquence de deux énoncés est donnée ci-dessus:

$$\frac{\langle c_0, s_{start1}\rangle \rightarrow s_{end1} \quad \langle c_1, s_{start2}\rangle \rightarrow s_{end2}}{\langle c_0;c_1, s_{start1}\rangle \rightarrow s_{end2}} \qquad (5.6)$$

où ";" représente l'opérateur de séquencement.

La règle d'exécution (5.6) énonce que si l'exécution d'un énoncé séquentiel c_0 dans l'état s_{start1} mène à l'état s_{end1}, et que l'exécution d'un énoncé séquentiel c_1 dans l'état s_{start2} mène à l'état s_{end2}, alors le séquencement des énoncés séquentiels c_0 et c_1 à partir de l'état s_{start1} mène à l'état s_{end2}.

5.4 Automate pour un processus unique

5.4.1 Construction de l'automate

Soit c la séquence des énoncés séquentiels d'un processus. L'automate pour l'énoncé de processus (AEP) est construit en reliant l'état final s_{end} des énoncés séquentiels à l'état initial s_{start} par une transition spontanée, dont la condition de transition est toujours satisfaite et qui n'effectue aucune opération. Cela reflète le comportement cyclique du processus. Un état de synchronisation additionnel est rajouté avant l'état s_{start} des énoncés séquentiels, parce qu'on entre toujours dans un processus par un état de synchronisation [8]. Le modèle obtenu va commencer son exécution avec le premier énoncé séquentiel du processus. La construction d'un automate d'énoncé de processus à partir de l'automate d'énoncé séquentiel de la séquence des énoncés composant le processus est représenté dans la figure 5.6.

La mise en correspondance entre les états d'un automate et ceux d'une machine à état fini est décrite dans la section 5.6.

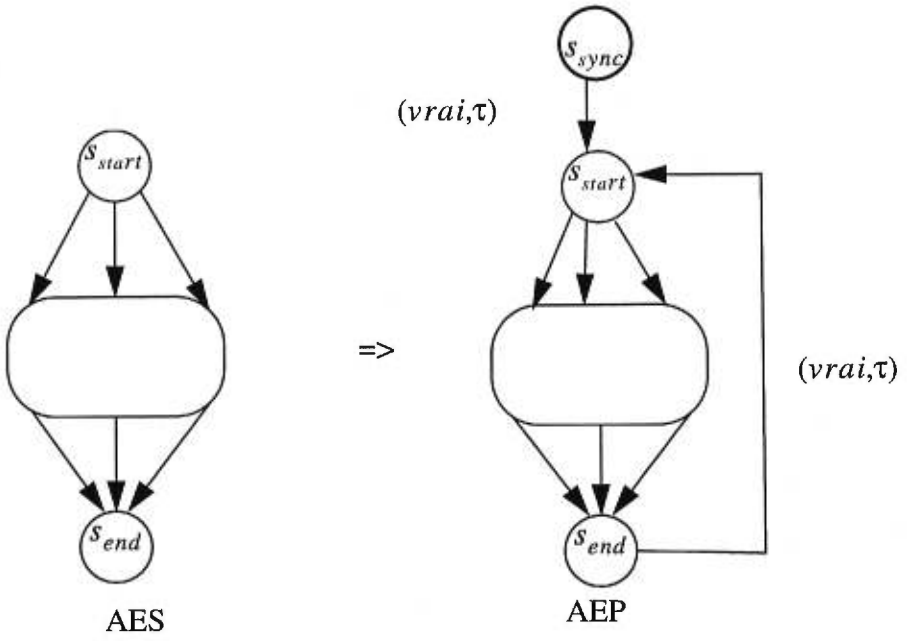


figure 5.6 Automate d'un énoncé processus

$$\frac{\langle c, s_{start}\rangle \rightarrow s_{end}}{\langle c, s_{start}\rangle \rightarrow s_{end}, \langle \tau, s_{end}\rangle \rightarrow s_{start}} \qquad (5.7)$$

La règle (5.7) énonce que, si l'exécution des énoncés séquentiels c d'un processus dans l'état s_{start} mène à l'état s_{end}, alors on ajoutera après l'état s_{end} une transition avec une opération nulle qui fera passer de s_{end} à s_{start}, pour construire l'automate d'énoncé processus à partir d'une séquence d'énoncés séquentiels.

5.4.2 Exemple d'automate d'un processus

Les processus sont en général composés de séquences, et d'imbrications d'énoncés séquentiels dans diverses alternatives d'énoncés de contrôle de flot. Le processus de la machine à état de la figure 5.7 comporte au niveau le plus élevé deux énoncés séquentiels sur les lignes 1 et 2. L'énoncé de la ligne 2 est un énoncé conditionnel *if* ne comportant qu'une seule alternative, qui imbrique des énoncés séquentiels à plusieurs niveaux. L'énoncé séquentiel de plus haut niveau de cette imbrication est l'énoncé *if* de la ligne 3 comportant deux alternatives: l'une à la ligne 3 et l'autre à la ligne 6. L'alternative de la ligne 6 comporte un énoncé *case* sur la ligne 7 de deux alternatives sur les lignes 8 et 13, etc.

L'application des règles d'exécution, des énoncés séquentiels, des séquences d'énoncés séquentiels, et des énoncés de processus au code de la machine à état de la figure 5.7 permet d'obtenir l'automate d'énoncé de processus détaillé de la figure 5.8. Les noeuds de cet automate sont étiquetés, par les noeuds des états initiaux et finaux des automates d'énoncés séquentiels du processus, et les états de synchronisation. Soit i la ligne d'un énoncé séquentiel du processus *sm* de la figure 5.7, s_{starti} étiquette l'état initial de l'automate de l'énoncé de la ligne i, $cond_i$ représente la condition d'exécution de l'énoncé de la ligne i, et e_i représente l'énoncé de la ligne i.

La figure 5.8 représente toutes les séquences d'exécutions (un graphe de flot de contrôle) du processus de la figure 5.7. Un chemin d'exécution est une séquence d'états (ou d'énoncés) dans un automate d'exécution. Par exemple la séquence d'états s_{start1}, s_{sync}, s_{start2}, s_{start3}, s_{start6}, s_{start7}, $s_{start14}$ et $s_{start17}$ est un chemin d'exécution indiqué en gris dans la figure 5.8.

```
entity mealy_b is
port(
    clk: in bit; -- horloge
    x : in bit; -- signal d'horloge
    rst : in bit; -- signal de remise à l'état initial
    z : out bit); -- signal de sortie
end mealy_b;


architecture alg_mealy_b of mealy_b is
 type states is (s0, s1);
 signal state : states := s0;

begin

-- processus définissant la synchronisation de la machine avec l'horloge, la fonction
de transition et la fonction de sortie
 sm: process
   begin
    wait on clk;                              --1
    if(clk'event and clk = '1') then          --2
     if (rst = '0') then                      --3
        state <= s0;                          --4
        z <= '0';                             --5
     elsif(rst = '1') then                    --6
        case state is                         --7
           when s0 =>                         --8
                      if(x ='1') then         --9
                          state <= s1;        --10
                          z <='1';            --11
                      else z <=  '0';         --12
                      end if;
           when s1 =>                         --13
                       if(x ='1') then        --14
                         state <= s0;         --15
                         z <='0';             --16
                       else  z <= '1';        --17
                       end if
        end case;
     end if;
    end if;
   end process sm;
```

figure 5.7 Spécification d'une machine à état fini VHDL de type MEALY

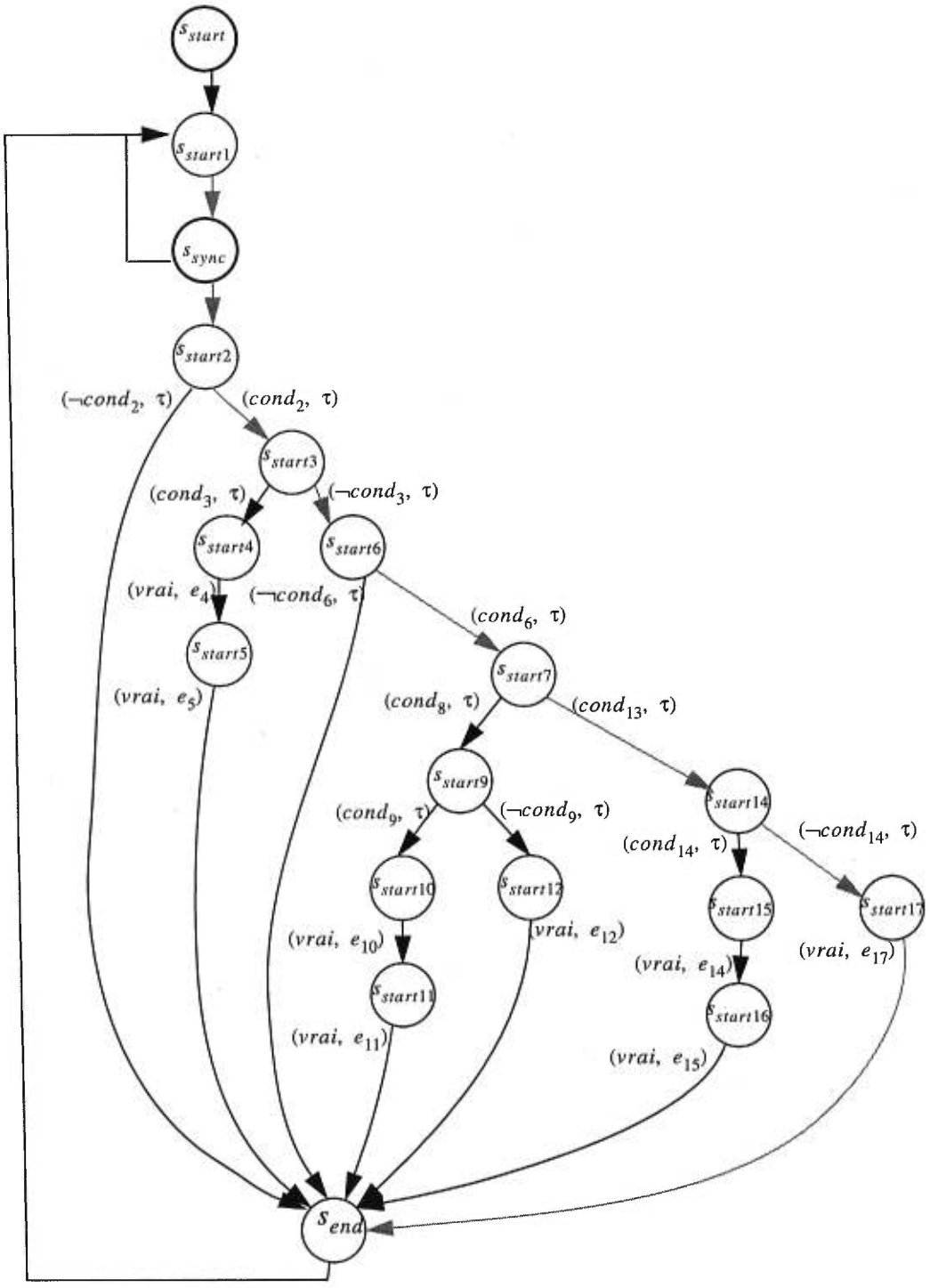


figure 5.8 Automate d'énoncé de processus détaillé du processus *sm* de la figure 5.7

La mise à jour des valeurs effectives des signaux réalisée par le processus noyau ne peut pas apparaître dans la figure 5.8 qui représente un processus défini par

l'utilisateur. Elle apparaît dans la section suivante qui définit un automate pour le modèle comportemental VHDL.

5.5 Automate pour un modèle comportemental VHDL

Le modèle comportemental en VHDL est défini par un ensemble de processus, ou une forme équivalente. En général, tous les processus d'une même architecture peuvent être modélisés par un seul automate. Cet automate représente le modèle, et son exécution doit être équivalent à l'exécution en parallèle des processus définis par l'utilisateur. L'exécution en parallèle de tous les processus commence dans l'état s_1 de la figure 5.9.

Selon le standard IEEE, l'effet des assignations des signaux est visible par tous les autres processus à la prochaine synchronisation [8]. Cela permet de simuler l'exécution en parallèle par une exécution séquentielle en mettant tous les automates d'énoncés de processus dans un ordre fixé et de passer de l'un à l'autre quand un état de synchronisation est atteint [8]. À partir du dernier automate d'énoncé de processus, la mise à jour des signaux est réalisée, puis nous retournons au premier automate d'énoncé de processus.

La composition de l'automate de tous les processus et de l'automate de mise à jour des signaux est faite de telle sorte que l'état s_2 n'est atteint que si tous les processus sont suspendus. La condition ψ est valide si l'environnement est prêt à mettre à jour les signaux. Comme il n' y a que des délais delta, la transition $(s_2, s_1, vrai, updatesig)$ met à jour les signaux et calcule les nouvelles valeurs effectives.

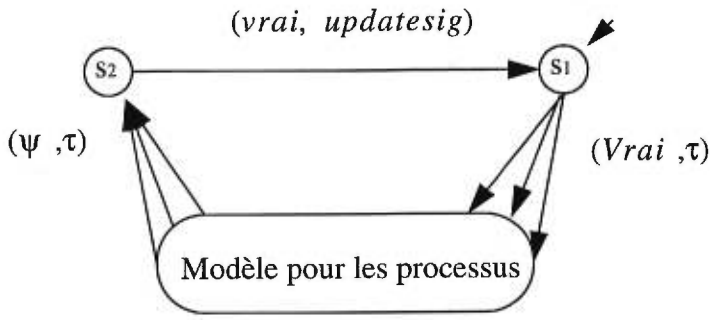


figure 5.9 Composition du processus noyau et des processus utilisateurs

Notre objectif dans la modélisation de l'automate pour un ensemble de processus est de parvenir à composer correctement le comportement d'un ensemble de processus parallèles. Pour simplifier la composition parallèle des processus, et surtout pour parvenir à la réaliser avec les formalismes de notre langage hôte (langage C) qui est séquentiel, nous avons défini dans le Chapitre III, une spécification de machine à état avec les propriétés suivantes:

1. Le processus séquentiel n'est activé par aucun autre processus, et active tous les processus combinatoires et ne contient qu'un seul énoncé *wait*.

2. Les processus combinatoires sont indépendants les uns des autres, c'est-à-dire qu'ils n'échangent pas de signaux entre eux. En effet, les signaux d'état sont indépendants, et les listes de sensitivité des processus combinatoires ne contiennent que des signaux d'entrée et d'état comme décrit dans la section 3.3.3. La composition des processus combinatoires se réduit en un séquencement de ces processus, dans n'importe quel ordre.

3. Chaque activation de processus combinatoire sert à calculer la (ou les) variables (ou signaux) d'état ou les signaux de sortie. Chaque processus combinatoire peut être vu comme un sous-programme qui sert à calculer la valeur effective d'un signal d'état ou de sortie.

Pour chaque exécution de l'ensemble des processus, une seule mise à jouer des signaux est réalisée, puisque nous n'avons qu'un seul énoncé *wait* dans tout le modèle. En effet, nous ne permettons qu'un seul énoncé *wait* dans un processus

séquentiel, et pas d'énoncé *wait* dans les processus combinatoires, comme décrit dans les sections 3.2.1.1 et 3.2.1.2. L'exécution parallèle dans chaque cycle de simulation peut tout simplement être remplacée, par l'exécution séquentielle du processus séquentiel et des processus combinatoires, suivie de la mise à jour des signaux.

5.6 Dérivation du comportement d'une machine à état à partir de l'automate d'exécution d'un modèle VHDL.

Nous décrivons ci-dessous la dérivation du comportement d'une machine à état, à partir de l'automate d'exécution d'un modèle comportemental. Le cas décrit ci-dessous est un cas simple, car le modèle VHDL ne comportement qu'un seul processus.

La mise à jour des valeurs des signaux étant effective, à la prochaine synchronisation, nous considérons toutes les opérations sur un chemin d'exécution donné entre deux états de synchronisation. Tout chemin d'exécution comportant au moins une opération d'affectation d'un signal ou d'une variable définit une transition de la machine entre les deux états synchronisation de l'automate. Les cinq chemins d'exécution suivants du graphe de la figure 5.8 définissent des transitions de la machine à état:

$$ce_1 = \{s_{start1}, s_{sync}, s_{statr2}, s_{start3}, s_{start4}, s_{start5}, s_{end}\}$$

$$ce_2 = \{s_{start1}, s_{sync}, s_{start2}, s_{start3}, s_{start6}, s_{start7}, s_{start9}, s_{start10}, s_{statr11}\}$$

$$ce_3 = \{s_{start1}, s_{sync}, s_{start2}, s_{start3}, s_{start6}, s_{start7}, s_{start9}, s_{start12}\}$$

$$ce_4 = \{s_{start1}, s_{sync}, s_{start2}, s_{start3}, s_{start6}, s_{start7}, s_{start14}, s_{start15}, s_{statr16}\}$$

$$ce_5 = \{s_{start1}, s_{sync}, s_{start2}, s_{start3}, s_{start6}, s_{start7}, s_{start14}, s_{start17}\}$$

La modification d'une variable ou d'un signal d'état, le long d'un chemin d'exécution entre deux états de synchronisation, définit un nouvel état de la machine à état. Contrairement, tout chemin d'exécution entre deux états de synchronisation, ne comportant aucune affectation à une variable ou à signal d'état va conserver l'état précédent de la machine. Nous distinguons dans la figure 5.8, trois chemins d'exécution

ce_1, ce_2, et ce_4 qui modifient la variable d'état *state* (transitent en général entre deux états différents), en exécutant respectivement les énoncés des lignes 4, 10 et 15, et deux chemins d'exécution ce_3 et ce_5 qui ne modifient pas la variable d'état (bouclent sur le même état). L'ancienne valeur de la variable d'état étant connue le long des chemins ce_2, ce_3, ce_4 et ce_5, ces chemins définissent chacun une seule transition, le chemin ce_1 définit un nouvel état quelque soit la valeur courante de l'état. Comme la variable d'état *state* peut prendre deux valeurs possibles, le chemin ce_1 définit deux transitions effectives. La machine à état comporte donc un total de six transitions.

On dérive les transitions de la machine à état, en exécutant toutes les opérations entre deux états de synchronisation et en mettant à jour les valeurs effectives des signaux d'état et de sortie. À partir de l'état de synchronisation, et on donne aux variables et aux signaux d'état leur valeur initiale dans la machine à état, par exemple $state = 0$, puis on dérive les transitions de la machine à état, jusqu'à ce que toutes les transitions soient couvertes. La machine à état correspondant à l'automate de la figure 5.8 est représentée dans la figure 5.10. La variable *nstate* représente la variable d'état suivant de la variable d'état *state*.

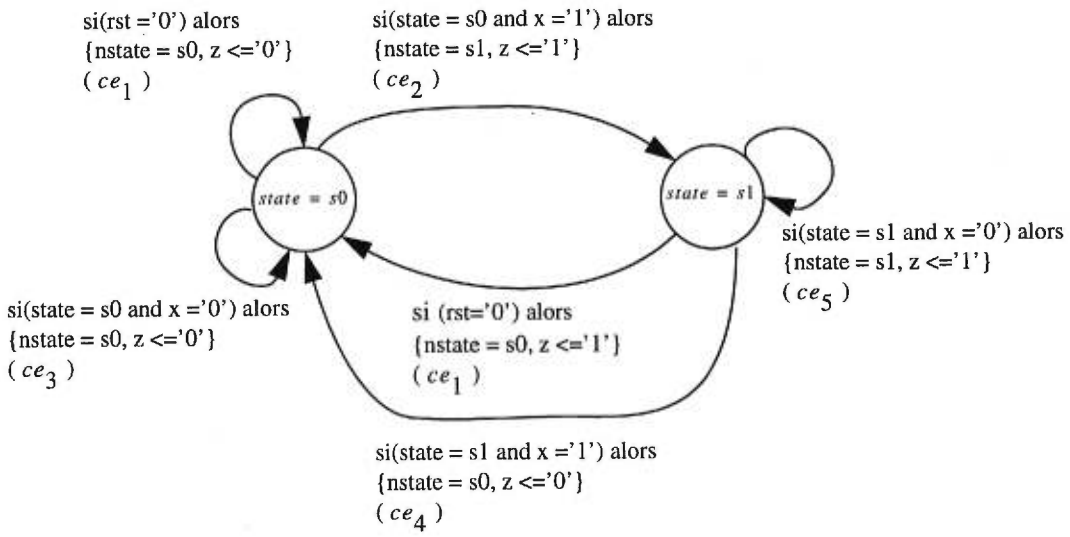


figure 5.10 Représentation graphique de la machine à état dérivée à partir de l'automate de l'énoncé de processus de la figure 5.8

Nous avons défini dans ce chapitre une sémantique des machines à état VHDL basée sur la description de leur comportement. Cette sémantique décrit le comportement de chaque machine à état, en exécutant une machine théorique représentée par un automate. Elle définit le comportement de chaque énoncé séquentiel, des séquences de ces énoncés, et des processus. Le comportement de la machine à état est déduit de l'automate du modèle de description VHDL.

La sémantique définie dans ce chapitre sert de base pour la construction des chemins d'exécution, autour desquels est bâtie la traduction. Le chapitre suivant présente les outils utilisés et décrit l'environnement de traduction des modèles VHDL vers les modèles MDG.

Chapitre VI

Environnement de la traduction

Ce chapitre présente les tâches classiques d'une traduction, les outils que nous avons utilisés et les caractéristiques des langages source et cible qui influencent fortement la construction de notre traducteur.

6.1 Les phases de la traduction

La compilation est usuellement implantée comme une suite de transformations $(SL,\ L_1), (L_1,\ L_2), \ldots, (L_k,\ TL)$, où SL est le langage source, et TL, le langage cible. Chaque langage L_i est appelé langage intermédiaire. Les langages intermédiaires servent à décomposer les tâches de la compilation du langage source au langage cible. Pour notre traducteur, SL est un modèle VHDL de machine à état fini, TL est une machine à état abstrait, l'arbre de syntaxe abstraite, les relations de transition et de sortie et l'état initial de la machine à état, sont des langages intermédiaires (Les différents langages intermédiaires sont indiqués dans la figure 6.4).

Toute compilation peut-être décomposée en deux tâches majeures [28]:

- l'analyse: rapporte la structure et les primitives du langage source.
- la synthèse crée un programme cible équivalent au programme source.

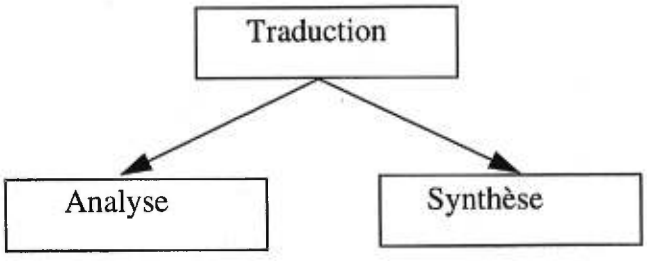


figure 6.1 Décomposition des tâches de la traduction

6.1.1 L'analyse

Elle convertit le programme source, en une représentation abstraite exprimant les propriétés essentielles du programme source. L'arbre de syntaxe abstraite est la forme la plus utilisée parmi les différentes implantations de la représentation abstraite. La structure de l'arbre représente les aspects de flots de contrôle et de données du programme.

L'analyse est généralement divisée en deux parties (la décomposition des tâches de l'analyse se retrouve dans la figure 6.2), l'analyse structurale qui détermine la structure statique du programme, et l'analyse sémantique qui établit les informations additionnelles et vérifie leur consistance. L'analyse lexicale et l'analyse syntaxique constituent les tâches de l'analyse structurale. L'analyse lexicale définie par des automates finis, reconnaît les symboles de base du langage source. Elle agit comme un module de l'analyseur syntaxique qui reconnaît la syntaxe du langage source. L'analyse syntaxique est décrite en termes d'automates à pile.

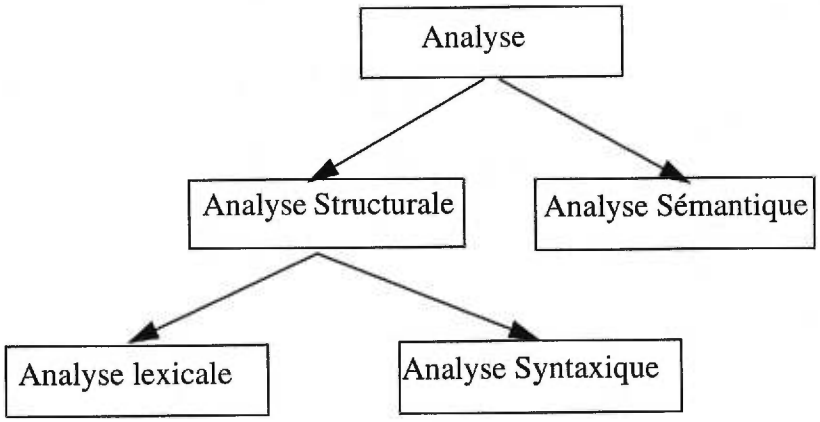


figure 6.2 Décomposition des tâches de l'analyse

6.1.2 La synthèse du langage cible

Le point de départ de la synthèse du langage cible est la représentation abstraite du langage source. La synthèse du langage cible reflète la correspondance entre le langage source et le langage cible. Elle consiste à créer un arbre cible permettant d'identifier les concepts pertinents du langage cible, plus précisément les symboles de fonction, les constantes, les sorts, les tables, les composants, les blocs fonctionnels, la liste et l'ordonnancement des symboles.

6.2 Les outils de la traduction

L'organisation logique d'un compilateur est présentée dans la figure 6.3. Dans une implantation, les activités de plusieurs phases peuvent être regroupées.

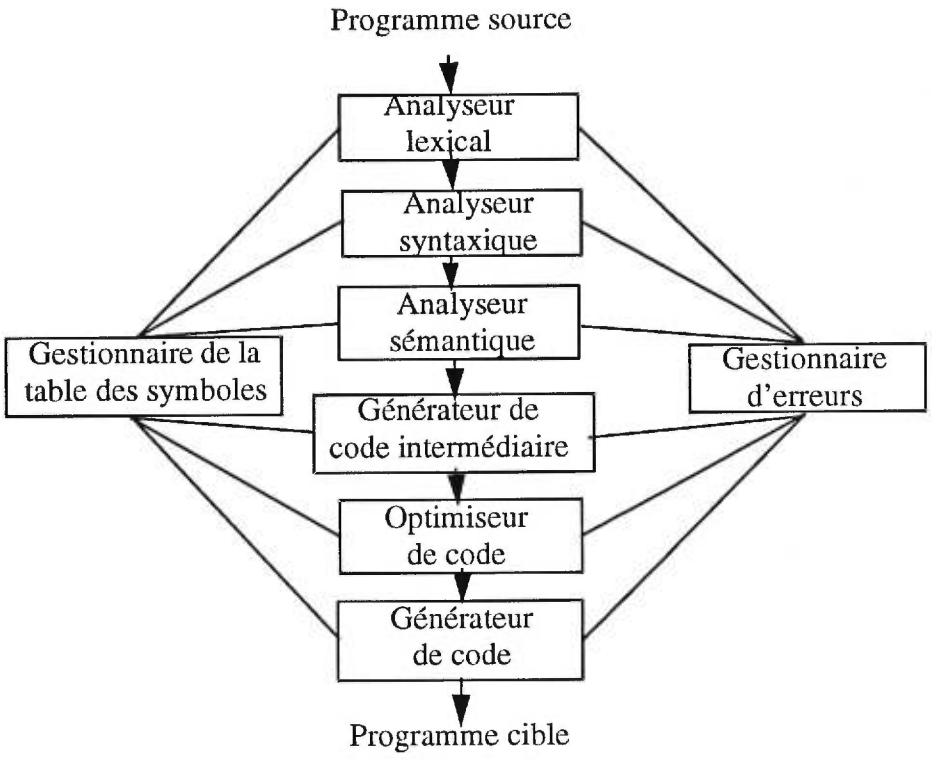


figure 6.3 Organisation logique d'un compilateur

De nombreux systèmes sont destinés à aider à la réalisation des compilateurs. Certains outils généraux ont été crées pour la conception automatique de composants spécifiques de compilateur. Les outils largement utilisés cachent les détails de leurs algorithmes et produisent des composants faciles à intégrer au reste du compilateur.

Pour l'implantation de notre compilateur, nous avons utilisé un outil réalisant l'analyse lexicale, l'analyse syntaxique et une partie de l'analyse sémantique du code source. Cet outil appelé *LEDA'S VHDL System* (LVS) [19] [20], accepte en entrée un code source en VHDL et produit un arbre de syntaxe abstraite. Les phases de réalisation de notre compilation qui en résultent sont représentées sur la figure 6.4.

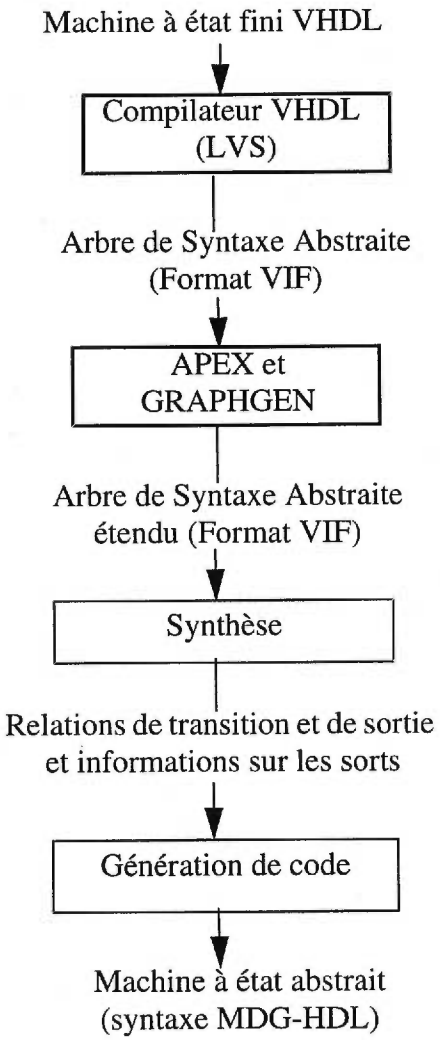


figure 6.4 Les phases de réalisation de notre compilateur

6.3 Le système VHDL de LEDA

Le système LEDA de VHDL est un environnement de compilation pour les applications de VHDL. Il analyse les descriptions VHDL, produit une représentation abstraite du code source dans un format binaire, et la stocke dans une bibliothèque. Ce format binaire bâti autour de VIF (*VHDL Intermediate Format*), est issu des documents des groupes de travail de IEEE pour les représentations intermédiaires de VHDL. Les données de format binaire sont accessibles, à travers l'Interface procédurale de LEDA (*LEDA Procedural Interface* - LPI).

L'outil GRAPHGEN génère un graphe de flot de contrôle et de données, pour chaque processus à partir de l'arbre de syntaxe abstraite. Il procède en ajoutant des noeuds à l'arbre de syntaxe abstraite et produit un arbre de syntaxe abstraite étendu dans lesquels les séquences d'exécution des énoncés sont représentées.

APEX est outil qui extrait les propriétés comme l'usage des objets (signaux et variables) et les informations de synchronisation. Il permet par exemple de relier tous les usages d'un objet donné. Il procède également par extension du graphe de syntaxe abstraite. Nous l'utilisons en particulier pour identifier le signal d'horloge.

6.3.1 Le format VIF

Au niveau conceptuel, le format intermédiaire est un arbre dont les noeuds, structurés en différentes classes, possèdent des attributs pointant sur les valeurs des noeuds de l'un des trois types suivants:

1. valeur de primitive.
2. lien unique vers un autre noeud.
3. liste de liens d'autres noeuds.

Par exemple, le tableau suivant présente cinq attributs du noeud *architecture_body* de la représentation VIF, qui représente l'architecture d'un design. Les attributs *has_entity_name* et *has_id* pointent chacun vers un noeud, alors que les

attributs *has_decl_s*, *has_stm_s* et *has_user_att_s* pointent sur une liste de noeuds. La liste détaillé du format VIF est donnée dans [19].

Attribut	Description de l'attribut
has_entity_name	Pointe vers un noeud qui représente le nom de l'entité associé à la déclaration de l'architecture.
has_id	Pointe vers un noeud qui représente le nom associé à l'architecture courante.
has_decl_s	Pointe vers une liste ordonnée de noeuds qui représente la séquence des items déclaratifs, qui composent la partie déclarative de l'architecture courante.
has_stm_s	Pointe vers une liste ordonnée de noeuds qui représente la séquence des énoncés concurrents de l'architecture courante.
has_user_att_s	Pointe vers une liste ordonnée qui représente la séquence des attributs définis par l'utilisateur.

tableau 6.1 Attributs du *noeud architecture_body*

6.3.2 L'interface procédurale de LEDA

L'interface procédurale LEDA fournit un ensemble de fonctions d'accès au format intermédiaire en ANSI C. Ces fonctions accèdent:

1. aux unités de librairies,
2. aux noeuds,
3. aux attributs,
4. et au traitement d'erreurs.

6.3.3 Le générateur de graphes de flot de contrôle et de données GRAPHGEN

GRAPHGEN crée un graphe de flot de contrôle pour chaque énoncé de processus de la description VHDL. Pour chaque noeud du graphe de flot de contrôle, un graphe de flot de données doit être crée. Les noeuds du graphe de flot de contrôle peuvent être regroupés dans des blocs, et les graphes de flot de contrôle peuvent être en conséquence fusionnés, pour des raisons d'optimisation.

6.3.3.1 Les noeuds du graphe de flot de contrôle

Les graphes de flot de contrôle sont utilisés, pour représenter les différentes séquences possibles d'exécution pour chaque ensemble d'énoncés dans un processus VHDL. Un graphe de flot de contrôle (*Control Flow Graph* - CFG) est défini comme suit:

$$CFG = (V_c, E_c)$$

où V_c est un ensemble de noeuds correspondant aux différents énoncés de VHDL.

$$V_c = V_w \cup V_b \cup V_l \cup V_e$$

où

V_w = noeuds de synchronisation (*wait*)

V_b = noeuds de branchement (*if*, *case*, *exit*, *next*)

V_l = noeuds de boucles (*while*, *loop*)

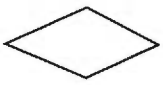
V_e = autres noeuds (*affectation*, *appel de procédure*, etc.)

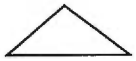
et E_c est un ensemble d'arcs identifiant le flot de contrôle.

$e = (v_1, v_2, c)$ où $v_1, v_2 \in V_c$.

L'arc e représente le fait que le noeud v_2 est exécuté après le noeud v_1 si la condition c est satisfaite. Le noeud v_2 est dit être le successeur de v_1.

La figure 6.5 représente le graphe de flot de contrôle du processus de la machine à état de la figure 5.7. Dans cette figure, chaque symbole porte le numéro de ligne de l'énoncé qu'il représente. Les correspondances entre énoncés et symboles graphiques sont données ci-dessous.


: représente un énoncé de synchronisation *wait.*


: représente un énoncé conditionnel (*if* ou *case*).


: représente un énoncé d'affectation.

Pour l'exemple de la figure 5.7, les ensembles de noeuds correspondant aux différents énoncés de VHDL sont donnés comme suit:

$$V_w = \{N_1\}$$

$$V_b = \{N_2, N_3, N_6, N_7, N_9, N_{14}\}$$

$$V_l = \varnothing \text{ où } \varnothing \text{ représente l'ensemble vide}$$

$$V_e = \{N_4, N_5, N_{10}, N_{11}, N_{12}, N_{15}, N_{16}, N_{17}\}$$

où N_i représente le noeud du graphe de flot de contrôle correspondant à l'énoncé de la ligne i.

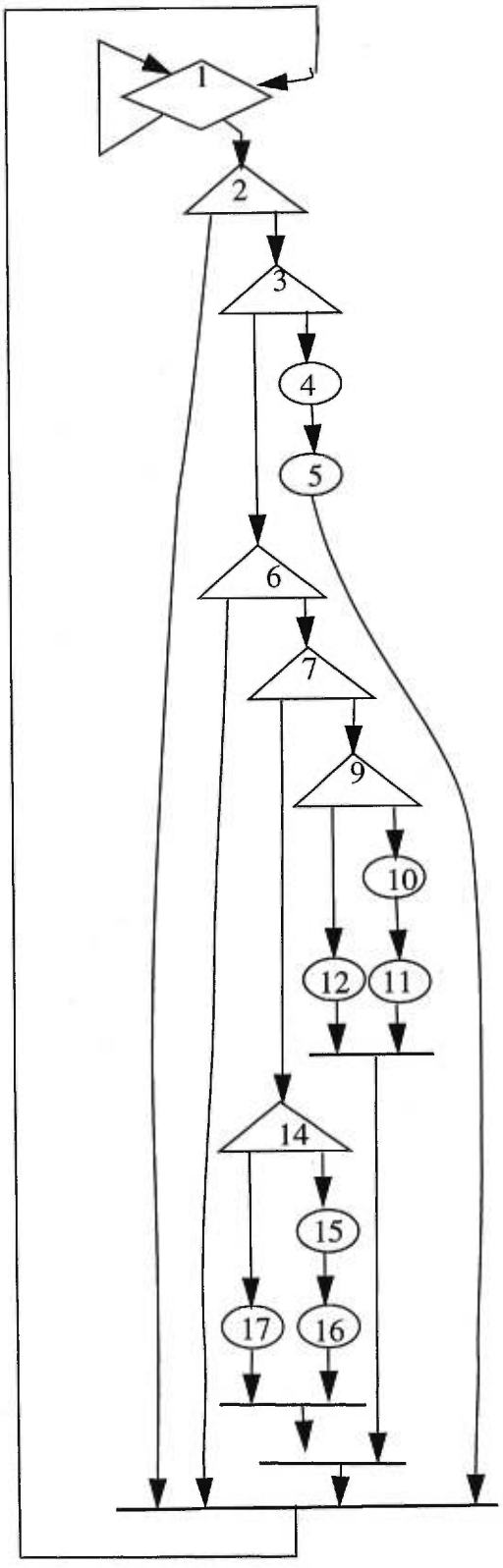


figure 6.5 CFG du processus de la machine à état VHDL de la figure 5.7

6.3.3.2 Les noeuds du graphe de flot de données

Pour chaque noeud du graphe de flot de contrôle, un graphe de flot de données peut être généré dépendant s'il y a un flot de données dans le noeud du graphe de flot de contrôle. Les noeuds des graphes de flots de données peuvent être divisés en quatre ensembles, en fonction du matériel qui leur sera alloué: opérateurs, opérandes, arcs, adresse. Les noeuds adresse aident à déterminer le type de ressource requis.

Un graphe de flot de données (*Data Flow Graph* - DFG) est défini comme suit:

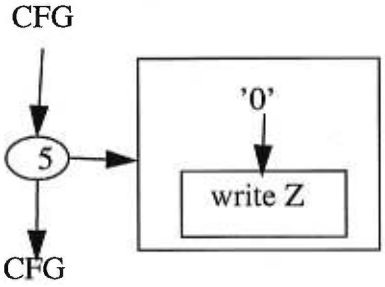
$$DFG = (V_d \cup O_d \cup A_d, E_d)$$

où

- V_d est l'ensemble des noeuds opérateurs,
- O_d est l'ensemble des noeuds opérandes,
- A_d est l'ensemble des noeuds adresse,
- E_d est l'ensemble des arcs liant les opérateurs et les opérandes.

Chaque graphe de flot de données est un graphe acyclique orienté.

Les graphes de flot de données apparaissent dans nos descriptions, dans l'affectation des variables et signaux. Ce sont les opérations dans les graphes de données qui définissent les changements d'état et les nouvelles valeurs des sorties dans les machines. Soit l'énoncé d'affectation suivante *Z <='0'*, sur la ligne numéro 5. Le graphe de flot de données associé à cet énoncé est représenté dans un carré à droite du noeud 5, comme suit:



Pour ce graphe l'ensemble des noeuds est défini comme suit:
$V_d = \varnothing$,
$O_d = \{'0'\}$
$A_d = \varnothing$
$E_d = \varnothing$

write Z est un noeud qui décrit l'écriture d'une valeur dans le signal *Z*

figure 6.6 Graphe de flot de données de l'affectation d'une constante

6.3.3.3 Partitionnement des graphes de flot de contrôle

Le graphe de flot de contrôle généré, est partitionné en ensemble de noeuds contigus. Les partitions pertinentes sont les blocs de base et les chemins d'exécution.

1. Blocs de base

Un bloc de base est une séquence d'énoncés consécutifs dans lequel, le flot de contrôle est activé au début de celle-ci, et inhibé à la fin de celle-ci, sans possibilité d'arrêt ou de branchement autrement qu'à la fin de la séquence. La figure 6.7 représente les partitions en blocs de base du graphe de flot de contrôle la figure 6.5. Tous les noeuds du graphe appartenant au même bloc de base sont regroupés sur le graphe.

2. Chemins d'exécution

Les chemins d'exécution regroupent les noeuds du graphe de flot de contrôle qui peuvent être exécutés pendant une passe complète du graphe. En conséquence, à chaque noeud ayant plus d'un successeur, un nouvel ensemble de chemins est généré, un pour chaque successeur. Tous les chemins commencent après le premier noeud. Les noeuds successeurs sont séquentiellement ajoutés jusqu'à ce que, soit le noeud n'aie plus de successeur, soit le prochain noeud apparaisse déjà dans le chemin. Dans le graphe de flot de contrôle de la figure 6.5, un chemin d'exécution est n'importe quelle séquence de noeuds entre lesquels existent un lien de succession.

Dans la section 5.3, nous avons construit un automate pour une séquence d'énoncés séquentiels. Cette représentation établit comment s'enchaînent les énoncés séquentiels dans l'exécution du processus. Chaque séquence d'enchaînement des énoncés séquentiels est un chemin d'exécution.

GRAPHEN fournit des chemins d'exécution pour chaque bloc de base. Il faut donc concatener les chemins des blocs de base pour obtenir les chemins d'exécution pour tout le processus VHDL.

Les graphes de flot de contrôle et de données, sont utilisés dans la section 7.3.3 pour construire les relations de transition et de sortie de la machine à état.

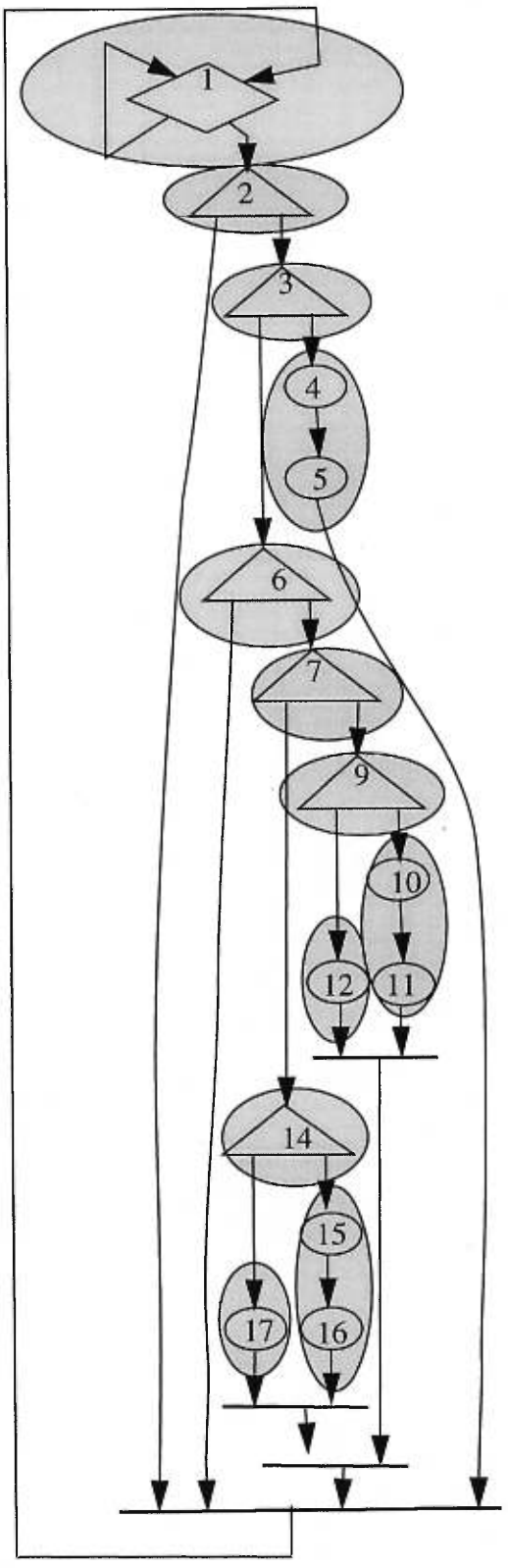


figure 6.7 Partitions en blocs de base du graphe de flot de contrôle la figure 6.5

6.4 Contraintes de traduction imposées par MDG-HDL

6.4.1 Les sorts et leur spécification dans le code VHDL

Le concept de sort est une caractéristique particulière de MDG. Sa spécification dans le code VHDL est réalisée, soit de façon explicite par la déclaration de l'attribut utilisateur *sort* attaché à un type de donnée, qui est soit concret (*concrete*) soit abstrait (*abstract*), soit déduit de la nature du type. Parmi les types de données dont le sort est sytématiquement connu, on distingue les types *integer* et *natural*, dont les sorts sont prédéfinis *abstrait*, et ceux dont les sorts sont déduits du fait de la définition du type. En effet, tout type d'énumération ou tout type définissant un intervalle fini de valeurs entières est systématiquement concret. Ainsi donc, les types prédéfinis *bool*, *bit* et *std_logic*, de VHDL sont de sort concret.

6.4.2 Transformations induites par la partition des types de données

Un type de données VHDL est toujours caractérisé par un ensemble de valeurs et un ensemble d'opérations applicables aux objets du type. De façon intrinsèque, tous les types de données au sens de VHDL devraient appartenir par conséquent au sort *concret*. De plus pour les types scalaires prédéfinis, le programmeur ne fait ni référence à l'ensemble des valeurs ni aux opérateurs associés avant de l'utiliser. Contrairement, toutes symboles utilisés dans MDG doivent avoir été déclarées.

La partition des types de données de MDG en sorts abstrait et concret conduit à:

- l'évaluation des expressions de sort concret,
- l'interprétation des opérateurs concrets,
- la création de termes croisés ou de termes abstraits correspondant à des opérateurs prédéfinis du langage VHDL,
- et à la création de constantes génériques et de composants signaux constants correspondant à des valeurs de données abstraites.

6.4.3 Représentation plate de MDG-HDL

MDG-HDL ne supporte pas une description hiérarchique des circuits. En conséquence, les descriptions hiérarchiques de VHDL doivent être aplaties.

6.4.4 Opérateur implicite de MDG-HDL

De façon intrinsèque, MDG ne connaît que l'opérateur *égalité* qui est implicite dans les expressions de condition. Comme les symboles fonctionnels concrets doivent être explicitement définis, et ne doivent pas apparaître dans les description MDG, il faut donc interpréter tous les opérateurs de VHDL appliqués à des opérandes concrets avec l'opérateur implicite de VHDL. Par exemple si *a* est un opérande de type bit, l'expression booléenne *a /='1'* doit être interprétée par *a='0'*. L'opérateur concret *égalité* n'apparaît pas dans les représentations de MDG.

6.5 Caractéristiques spécifiques de VHDL

6.5.1 Machines à état décrites sous forme comportementale

Le code d'une spécification en VHDL contient des énoncés concurrents et des énoncés séquentiels. Tous les énoncés concurrents comportementaux ont une forme équivalente sous forme de processus. Chaque processus peut décrire soit un circuit combinatoire soit un circuit séquentiel. Les relations de transition et de sortie sont décrites par le comportement séquentiel défini à l'intérieur de chaque processus. Le graphe de flot de contrôle des processus définit les conditions associées aux représentations tabulaires des relations de transition et de sortie.

6.5.2 Correspondance entre états des machines et énoncés *wait*

La description de comportement RTL, définit les opérations effectuées à chaque cycle d'horloge. Ces opérations sont décrites par une séquence d'énoncés séquen-

tiels du processus entre deux énoncés *wait*. Un état de synchronisation est atteint à l'exécution d'un énoncé *wait*. L'effet d'affectation aux signaux est alors visible, chaque signal contient sa valeur effective, et une analyse des valeurs des variables et signaux d'état détermine l'état de la machine.

L'horloge dans les circuits synchrones définit quand les sorties et les états suivants sont calculés. L'évaluation de l'état suivant peut conduire à un changement d'état donc à une transition de la machine à état. On peut faire correspondre les énoncés *wait* des codes VHDL à un état de la machine comme décrit dans la section 5.6.

6.5.3 Traitement des séquences de commandes

Les états des machines correspondant aux états de synchronisation (exécution des énoncés *wait*), et les descriptions RTL décrivant les opérations à chaque cycle d'horloge, les opérations des énoncés séquentiels entre deux énoncés *wait* ne peuvent considérées individuellement. Entre deux énoncés *wait,* on fait la composition séquentielle des opérations des énoncés séquentiels. Cette composition est fonction de la dépendance des données.

Ainsi donc, une dépendance de flux données doit conduire à une composition des opérations. Par exemple les énoncés "y:= f(x)" et "z:= g(y);" pris dans cet ordre vont être remplacés par "z:= g(f(x))"; Et une dépendance de sortie ne prendra en compte que la dernière affectation.

Nous avons, décrit dans ce chapitre les tâches dans la réalisation d'un traducteur, présenté des outils que nous avons utilisés, et donné des quelques caractéristiques des langages source et cible importantes pour la traduction. Le chapitre suivant décrit en détail les mécanismes de la traduction.

Chapitre VII

Traduction des formalismes de VHDL

La description d'un circuit en MDG-HDL est repartie dans trois fichiers: le *fichier des spécifications algébriques*, le *fichier de description du circuit* et le *fichier des symboles ordonnés* selon l'ordre sur mesure. Dans une première phase, le résultat de la traduction des descriptions VHDL est mémorisée dans une structure de données. La répartition des informations dans les différents fichiers est faite pendant la phase de la génération du code MDG-HDL.

7.1 La traduction des déclarations

La correspondance entre les formalismes VHDL et leur traduction en MDG-HDL est présentée de part et d'autre d'une double flèche, à gauche de la double flèche est inscrit le formalisme VHDL, et à sa droite on trouve le formalisme MDG-HDL correspondant.

-- formalisme VHDL => *% formalisme correspondant en MDG-HDL*

7.1.1 Les déclarations de type

7.1.1.1 Les Types d'énumération

La spécification de sort n'est pas requise pour les types d'énumération. Tout type d'énumération est concret par définition. La traduction d'un type d'énumération est représenté ci-dessous.

type booleen is (vrai, faux); => *concret_sort(booleen, [vrai, faux]).*

7.1.1.2 La déclaration de sous-type

Les sous-types numériques définis sur un segment fini (ayant des bornes numériques finies) sont de sort concret.

- Les sous-types numériques.

  L'exemple ci-dessous représente la traduction d'un sous-type numérique.

  subtype FreeRange is integer range 1 to 6; => *concrete_sort(FreeRange,[1,2,3,4,5,6]).*

- Déclaration des tableaux contraints

  Les éléments du tableau peuvent être soit de type *std_logic* soit de type *bit*, et les index sont définis sur un segment fini. La structure de tableau n'apparaît pas dans la traduction qui construit l'ensemble des valeurs. Un exemple de traduction apparaît ci-dessous.

  subtype memvector2 is std_logic_vector(1 downto 0); => *conc_sort(mem_vector2, ["00","01","10","11"]*

7.1.1.3 Les autres types

Les sorts des autres types sont soit prédéfinis ou définis par l'utilisateur. L'exemple ci-dessous traduit la spécification VHDL d'un type de sort abstrait.

attribute sort of positive: type is abstract; => *abs_sort(positive).*

7.1.2 Les déclaration des constantes

Les déclarations VHDL des constantes concrètes n'apparaissent pas dans les descriptions MDG. Toutes les constantes concrètes des descriptions VHDL doivent être remplacées par leur valeur. Les constantes abstraites dans la spécification VHDL sont traduites par une déclaration de constantes génériques comme dans l'exemple suivant:

constant zero_int : integer := 0; => *gen_const(zero_int).*

L'utilisation d'une constante individuelle abstraite entraîne la déclaration d'une constante générique.

signal x: natural;
...
if(x = 0) then ...

=>

gen_const(natural_0).
...
eq_natural(x, natural_0) ...

7.1.3 Les déclarations des signaux et variables

Les déclarations de variables et signaux VHDL sont traduites par une déclaration de signaux MDG comme le montre l'exemple suivant:

signal x: std_logic; => *signal(x, std_logic).*

7.1.4 Les déclaration des fonctions

Les fonctions définies par l'utilisateur ne sont pas interprétées. Elles doivent par conséquent avoir au moins un paramètre d'entrée abstrait. Seules les signatures des fonctions sont traduites.

function inc2(input: integer) return positive is => *function(inc2[integer], positive);*
...
end inc2;

7.2 La traduction des expressions

Les expressions concernées sont les expressions booléennes conditionnelles définissant le contrôle de la machine. Elles sont sous une forme normale disjonctive. La première opération consiste à diviser les expressions en sous-expressions qui sont des conjonctions de termes binaires ou unaires. Par exemple, l'expression conditionnelle dans la portion de code suivante:

a , b: bit;
n : natural;
constant natural N := 16;
...
if((a = '1' and b /= '1')or n = N) then ...

est divisée en deux sous-expressions *(a ='1' and b /='1')* et *n = N*. Les sous-expressions concrètes sont interprétées, et des termes croisés ou termes abstraits sont construits pour les autres sous-expressions. Ainsi donc la sous-expression *a ='1' and b /='1'* devient *a ='1' and b ='0'* et la sous-expression *n = N* est remplacée par *eq_natural(n, N)* où *eq_natural* est le symbole opérateur correspondant à "=". Si

l'expression conditionnelle est complémentée, une étape préliminaire consiste à calculer la forme normale disjonctive de l'expression complémentée.

7.3 La traduction des modèles comportementaux

Cette traduction consiste à définir dans le modèle supporté par MDG, l'état initial, la relation de transition et la relation de sortie de la machine à état.

La fonction de transition individuelle de chaque variable d'état suivant (respectivement de sortie) est représentée sous forme tabulaire. Les étapes successives des transformations menant à la construction de ces tables sont présentées dans la section suivante.

7.3.1 Les étapes de traduction des modèles comportementaux

La traduction comporte les étapes suivantes, après les analyses lexicale et syntaxique, la construction des graphes de flot de contrôle et de flots de données partitionnée en blocs de base et chemins d'exécution par le système LEDA-VHDL.

1. Construire des chemins d'exécution complets en concatenant les chemins d'exécution des blocs de base.
2. Sur chaque chemin d'exécution entre deux états de synchronisation, résoudre les dépendances de données.
3. Interpréter les expressions concrètes, créer les termes abstraits et termes croisés, créer des signaux constants.

4. Pour chaque variable d'état suivant et de sortie, construire une liste des expressions affectées et des conditions d'affectation correspondantes.
5. Pour chaque variable de sortie ou d'état suivant, construire la table de transition ou de sortie correspondante, à partir des listes construites à l'étape précédente.
6. Générer le code MDG-HDL des représentations tabulaires des fonctions de transition et de sortie individuelles.

7.3.2 La traduction de l'état initial de la machine

L'état initial de la machine à état est constitué des variables ou signaux d'état et de leur valeur initiale.

7.3.2.1 Les variables d'état concrètes

Ce sont les variables d'état des machines à état explicite, dans lesquelles les variables d'état de la machine ont un type d'énumération. L'exemple suivant donne la traduction de l'initialisation d'une variable d'état concrète, *state*.

```
type state is (s0, s1);
signal state: states := s0;        =>    % déclaration de la valeur initiale d'une variable d'état
                                         init_val(state, s0);
```

7.3.2.2 Les variables d'état abstraites

Elles ne figurent que parmi les variables du processus qui servent à représenter l'état des machines à état implicite. Leur traduction nécessite la création d'une variable d'état initiale dans la description MDG, *init_y* dans l'exemple ci-dessous.

signal y: integer:= 0;	=>	*% déclaration d'une variable d'état initiale* *init_var(init_y, integer).* *init_val(y, init_y).*

7.3.3 La traduction des relations de transition et de sortie

Le principe de la traduction des relations de transition et de sortie est le même. Les variables concernées sont respectivement les variables d'état dans le cas des relations de transition, et les variables de sortie dans le cas des relations de sortie.

La fonction de transition représente le contrôle de la machine. Elle se traduit concrètement par un ensemble de conditions sur les variables d'entrée et d'état, qui conduisent à l'affectation à une variable d'état une nouvelle valeur. Autrement dit, le dernier énoncé de la portion du chemin d'exécution donné entre deux états de synchronisation, est un énoncé d'affectation à une variable d'état. Les conditions sont la conjonction de toutes les conditions des énoncés de contrôle d'état, depuis le premier énoncé jusqu'à l'énoncé d'affectation.

```
entity mealy_b is
port(
    clk : in bit;
    x   :in bit;
    rst : in bit;
    z   : out bit);
end mealy_b;


architecture alg_mealy_b of mealy_b is
 type states is (s0, s1);
 signal state : states := s0;

begin
-- process to define latch, next state, and output
 sm: process

  begin
   wait on clk;                                    --1
   if(clk'event and clk = '1') then                --2
    if (rst = '0') then                            --3
       state <= s0;                                --4
       z <= '0';                                   --5
    elsif(rst = '1') then                          --6
       case state is
          when s0 => if(x = '1') then              --7
                          state <= s1;             --8
                          z <= '1';                --9
                     else z <= '0';                --10
                     end if;                       --11
          when s1 => if(x = '1') then              --12
                       state <= s0;                --13
                       z <= '0';                   --14
                     else z <= '1';                --15
                     end if
       end case;
    end if;
   end if;
  end process sm;
 end alg_mealy_b;
```

figure 7.1 Code VHDL d'une machine à état de type Mealy

Dans le processus de la machine à état de la figure 7.1 apparaissent cinq (5) chemins d'exécution définis sur la table 7.1 par le numéro de ligne des énoncés.

Chemin 1	Chemin 2	Chemin 3	Chemin 4	Chemin 5
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	6	6	6	6
5	7	7	7	7
	8	8	12	12
	9	11	13	15
	10		14	

table 7.1 Chemins d'exécution de la machine à état de la figure 7.1

Nous appellerons *environnement d'affectation* pour chaque énoncé d'affectation, la liste contenant le nom de la variable affectée, l'expression affectée et les conditions d'affectation. Soit *ea* l'environnement d'affectation de la variable *v* dans le chemin *ce*. Soit *Ea*, l'ensemble de tous les environnements d'affectation de la variable *v*, dans tous les chemins d'exécution.

La fonction *Eaff(v, Ce)* dans la figure 7.2 ci-dessous, prend en entrée la variable *v* et l'ensemble des chemins d'exécution *Ce*, et retourne l'ensemble des environnements d'affectation *Eaff* de la variable *v*. Elle utilise le prédicat *affecte(v, ce)* qui indique s'il existe un énoncé d'affectation de la variable *v* dans de chemin d'exécution *ce* et la fonction *eaff(v, ce)* qui calcule l'environnement d'affectation de *v* dans *ce*.

```
fonction Eaff(v, Ce) :Eaff
{Initialisation, ∅ représente l'ensemble vide}
Eaff ← ∅
Pour chaque element ce de Ce faire
   Si affecte(v, ce) alors {Si il existe un énoncé d'affectation de v dans ce}
      ea = eaff(v, ce) {calculer l'environnement d'affectation de v dans ce}
      Eaff ← Eaff ∪ {ea}

Retourner Eaff
```

figure 7.2 Algorithme de construction de l'ensemble des environnements d'affection d'une variable v

La figure 7.3 définit l'ensemble des environnements d'affectation de la variable *z*. La première ligne signifie que l'on affecte à *z* la valeur '0' quand la condition *rst ='0'* est satisfaite.

```
{{z,'0 ', {rst = '0'}},
  {z,'1 ', {rst = '1', state = s0, x = '1'},
    {z,'1 ', {rst = '1', state = s0, x = '0'},
    {z,'0 ', {rst = '1', state = s1, x = '1'},
      {z,'1', {rst ='1', state = s1, x ='0'}
}
```

figure 7.3 Ensemble des environnements d'affectation de la variable *z* de l'exemple de la figure 7.1

On fait l'union des conditions d'affectation, après une interprétation éventuelle pour obtenir l'en-tête de la représentation tabulaire de la fonction de sortie de la variable *z*. On ne retient que les variables concrètes lorsque les conditions sont concrètes, par exemple l'en-tête pour la condition *rst = 0* est tout simplement *rst*. Chaque environnement d'affectation sert à construire une ligne du corps de la table. L'expression affectée est inscrite dans la dernière colonne de la ligne. Pour trouver la valeur à inscrire sur une ligne pour les autres colonnes, on distingue les deux cas suivants. Si l'entête de la colonne est abstraite alors, si la condition est satisfaite, on inscrit 1, 0

dans le cas contraire, et * si la condition est sans importance. Si l'en-tête est concrète, alors on inscrit la valeur du terme contenue dans la condition. La table 7.1 illustre ce qui est décrit ci-dessus, pour la variable de sortie *z*. Dans ce cas, les en-têtes dans toutes les colonnes sont des variables concrètes.

rst	state	x	z
'0'	*	*	'0'
'1'	s0	'1'	'1'
'1'	s0	'0'	'0'
'1'	s1	'0'	'1'

table 7.2 Représentation tabulaire de la relation de sortie de z dérivée de la figure 7.3

```
%-----------------------------------------------
%           Algebraic specification
%           file: Run/Mealy_b/mealy_b_alg.pl
%-----------------------------------------------


%------ Standard header for Prolog system ----
:- multifile abs_sort/1.
:- multifile conc_sort/2.
:- multifile function/3.
:- multifile gen_const/2.
:- multifile rr/3.
:- multifile xtrr/3.


%------------ sorts section ----------
conc_sort(states, [s0, s1]).
conc_sort(bit, ['0', '1']).
```

figure 7.4 La section algébrique de spécification de la machine à état abstrait correspondant à la description VHDL de figure 7.1

```
%-----------------------------------------------
%            Circuit description
%            file: Run/Mealy_b/mealy_b_cir.pl
%-----------------------------------------------

%----- Include here Standard header required by Prolog System -----

%---------------- signals section -------------
signal(x, bit).
signal(rst, bit).
signal(z, bit).
signal(state, states).

%  table components section
component(state_comp, table([
[rst , state , x , next_state ] ,
[ '0' , * , * , s0 ],
[ '1' , s0 , '1' , s1 ],
[ '1' , s1 , '1' , s0 ]
| state ])).

component(z_comp, table([
[ rst , state , x , z ] ,
[ '0' , * , * , '0' ],
[ '1' , s0 , '1' , '1' ],
[ '1' , s0 , '0' , '0' ],
[ '1' , s1 , '1' , '0' ],
[ '1' , s1 , '0' , '1' ]])).

%------------ initial state section ----------

%----------- variables initial values section ------------
init_val(state, s0).

%------------ outputs signals section ----------
outputs([
z
]).

%------------ output partition section ----------
output_partition([
[[z]]
]).

%------------ next state variable section ----------
next_state_partition([
[[next_state]]
]).

%------------ state variable and next state variable mapping ----------
st_nxst(state, next_state).
```

figure 7.5 Machine à état abstrait du modèle VHDL de la figure 7.1

7.4 La traduction des modèles structuraux

La traduction des modèles comportementaux comporte la traduction des états initiaux, et des relations de transition et de sortie de la machine à état décrite. La traduction des modèles comportementaux aplatit la structure, rapporte l'interconnexion des composants, et traduit les modèles comportementaux des composants de plus bas niveaux.

Soit $C = \{c_i\}_{1 \leq i \leq n}$ un ensemble de n composants d'un niveau donné. L'algorithme suivant traduit un composant en une structure plate de boîtes noires et de composant sous forme tabulaire. Soit *BN* l'ensemble des boîtes noires de la description MDG, *CC* l'ensemble des composants dont le comportement est défini par une table dans la description MDG, et *cci* l'ensemble des composants comportementaux d'un composant *ci*.

La procédure *Traduire_composant* prend en entrée un circuit C définit par une interconnexion de composant et calcule *BN* et *CC*. *BN* et *CC* sont des ensemble vides avant l'appel, et sont initialisés comme suit: $CC \leftarrow \varnothing$ et $BN \leftarrow \varnothing$.

```
Procédure Traduire_composant(C, BN, CC)
  Pour chaque composant ci de C faire
    Si comportemental(ci)
      traduire_comp(ci, cci) {cci est l'ensemble des composants tables de ci}
      CC ← CC ∪ cci
    Sinon
      Si boite_noire(ci) traduire_boite_n(ci, bni) {bni est la boite noire correspondant au compo-
      sant ci}
        BN ← BN ∪ {bni}
      Sinon
        {ci a une description structurale}
        Pour chaque sous-composant cij de ci faire
        Traduire_composant(cij, BN, CC)
```

figure 7.6 Algorithme de traduction d'une description structurale

Considérons la description structurale d'un additionneur donnée dans la figure 7.7.

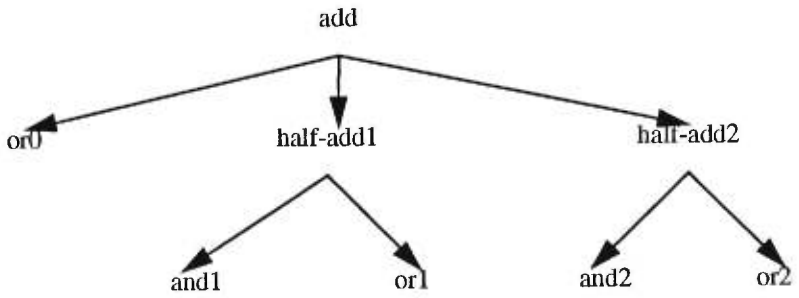


figure 7.7 Structure hiérarchique d'un additionneur

L'additionneur est composé d'une porte *ou* (or0) et de deux demi-additionneurs *half-adder1* et *half-adder2*. Chaque demi-additionneur est composé d'une porte *et* d'une porte *ou*. Les descriptions VHDL correspondantes sont réalisées à l'aide des descriptions structurales de l'additionneur (se retrouve dans la figure 7.8) et du demi-additionneur (se retrouve dans la figure 7.9) et des descriptions comportementales de la porte *ou* (se retrouve dans la figure 7.10) et la porte *et* (se retrouve dans la figure 7.11).

```
-------------------------------------------------------------------
--        design full_adder
-------------------------------------------------------------------

-- entity declaration
entity full_adder is
 port(
      a: in bit := '0'; b : in bit := '0'; carry_in : in bit := '0';
      ab : out bit := '0'; carry_out: out bit := '0');
 end full_adder;

-- architecture body
architecture str_full_adder of full_adder is

 -- signal declarations
  signal temp_sum: bit := '0';
  signal temp_carry_1: bit := '0';
  signal temp_carry_2: bit := '0';

 -- local components declarations
  component half_adder1
   port(x: in bit; y : in bit;
        sum : out bit; carry : out bit);
   end component;

  component or2
   port(or_i1 : in bit; or_i2 : in bit; or_o : out bit);
  end component;

  use work.all;
  for all: half_adder1 use entity half_adder(str_half_adder);

  for all: or2 use entity or_gate(beh_or_gate)
   port map(
          or_i1 => temp_carry_1, or_i2 => temp_carry_2,
          or_o => carry_out);

  begin
   -- component installation statements
   u0: half_adder1
    port map(
           x => a, y => b,
           sum => temp_sum, carry => temp_carry_1);
   u1: half_adder1
    port map(
           x => temp_sum, y => carry_in,
           sum => ab, carry => temp_carry_2);
   u2: or2
     port map(
          or_i1 => temp_carry_1, or_i2 => temp_carry_2,
          or_o => carry_out);

   end str_full_adder;
```

figure 7.8 Description structurale d'un additionneur

```
-------------------------------------------------------------------
--                    design half adder
-------------------------------------------------------------------

-- entity declaration
entity half_adder is
 port(
     x : in bit := '0';
     y : in bit := '0';
     sum : out bit := '0';
     carry : out bit := '0'
     );
 end half_adder;


-- architecture body
architecture str_half_adder of half_adder is

 -- local components declarations

 component and2
  port(and_i1 : in bit; and_i2 : in bit; and_o : out bit);
  end component;

 component or2
  port(or_i1 : in bit; or_i2 : in bit; or_o : out  bit);
  end component;

 use work.all;
 for all: and2 use entity and_gate(beh_and_gate);

 for all: or2 use entity or_gate(beh_or_gate);

   begin
 l1: or2
  port map(
        or_i1 => x, or_i2 => y,
        or_o => sum);
 l2: and2
  port map(
        and_i1 => x, and_i2 => y,
        and_o => carry);

  end str_half_adder;
```

figure 7.9 Description structurale d'un demi-additionneur

```
-------------------------------------------------------------------
--        design or_gate
-------------------------------------------------------------------

-- entity declaration
entity or_gate is
  port(
      or_i1 : in bit := '0';
      or_i2 : in bit := '0';
      or_o : out bit := '0');
end or_gate;

-- architecture body
architecture beh_or_gate of or_gate is

begin

  process(or_i1, or_i2)
    begin
      if (or_i1 = '1') or (or_i2 = '1') then or_o  <= '1';
      else or_o <= '0';

      end if;
    end process;
  end beh_or_gate;
```

figure 7.10 Une description comportementale de porte *ou*

```
-------------------------------------------------------------------
--        design and_gate
-------------------------------------------------------------------

-- entity declaration
entity and_gate is
 port(
     and_i1 : in bit := '0';
     and_i2 : in bit := '0';
     and_o : out bit := '0');
end and_gate;

-- architecture body
architecture beh_and_gate of and_gate is

begin

 process(and_i1, and_i2)
  begin
   if (and_i1 = '1') and (and_i2 = '1') then and_o <= '1';
   else and_o <= '0';

   end if;
  end process;
 end beh_and_gate;
```

figure 7.11 Une description comportementale de porte *et*

Nous allons appliquer l'algorithme de traduction d'une description structurale de la figure 7.6 à l'additionneur décrit ci-dessus. Cet exemple qui se retrouve dans la figure 7.12 représente les appels des différentes procédures dans l'algorithme de la figure 7.6. Certains paramètres sont de mode entrée/sortie par exemple les paramètres *BN* et *CC* de *Traduire_composant*, nous avons donc indiqué les paramètres au moment et au retour de chaque appel. Les flèches descendantes indiquent les appels et portent chacune un numéro d'ordre d'appel. Dans cette figure *ori* et *andi* (i = 0, 1, 2) représente les instances de composant de VHDL et *ccori* et *ccandi* (i = 0, 1,2) représente les instances de composants MDG-HDL.

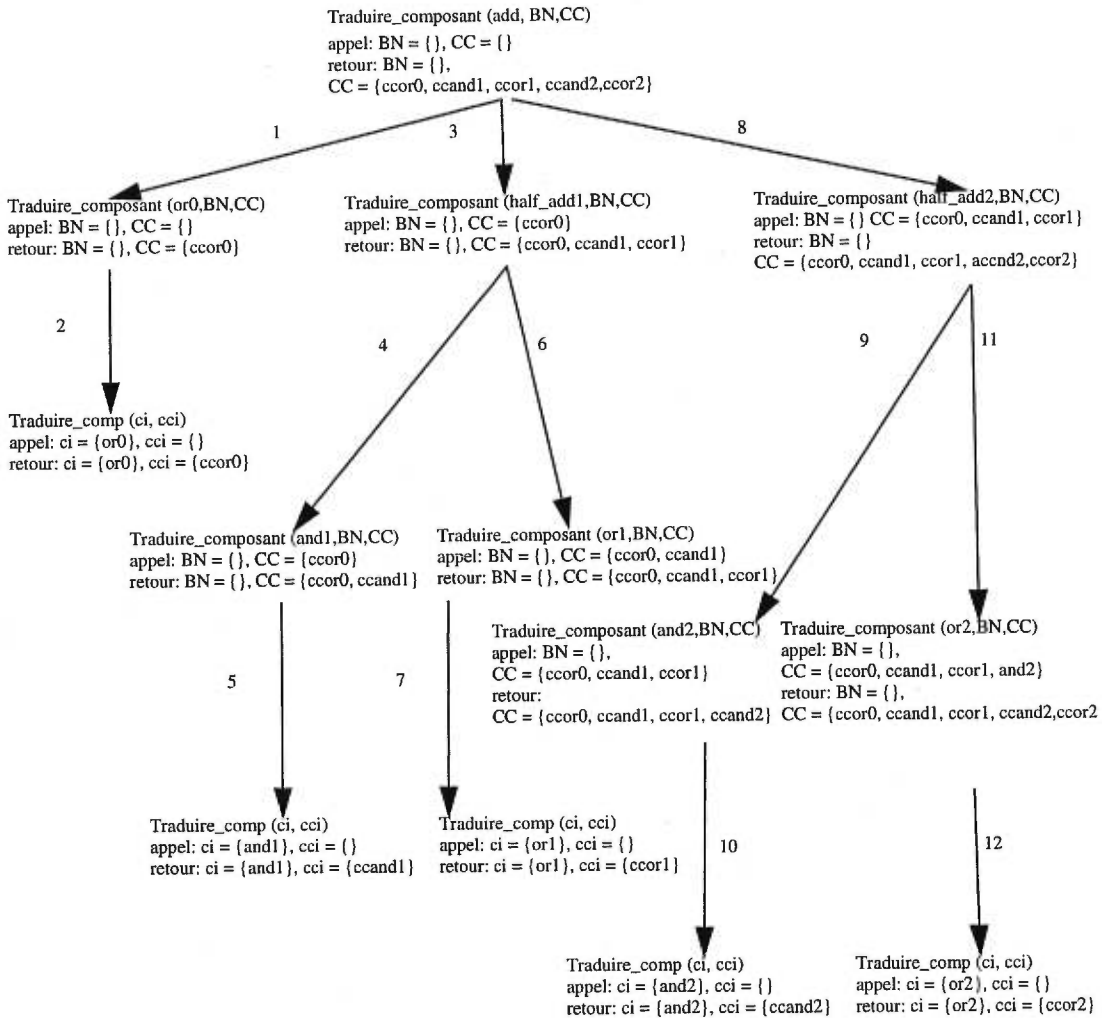


figure 7.12 Exemple d'application de l'algorithme de traduction d'une description structurale à un additionneur

```
%-----------------------------------------
%            Algebraic specification
%            file: Run/Full_adder/full_adder_alg.pl
%-----------------------------------------


%------ Standard header for Prolog system ----
:- multifile abs_sort/1.
:- multifile conc_sort/2.
:- multifile function/3.
:- multifile gen_const/2.
:- multifile rr/3.
:- multifile xtrr/3.


%------------ sorts section ----------
conc_sort(bit, ['0', '1']).
```

figure 7.13 Section algébrique de la spécification machine à état abstrait correspondant à la description structurale de l'additionneur

```
%-------------------------------------------
%           Circuit description
%           file: Run/Full_adder/full_adder_cir.pl
%-------------------------------------------


%----- Include here Standard header required by Prolog System -----

%---------------- signals section ------------
signal(a, bit).
signal(b, bit).
signal(carry_in, bit).
signal(carry_out, bit).
signal(temp_sum, bit).
signal(temp_carry_1, bit).
signal(temp_carry_2, bit).
signal(ab, bit).
signal(next_ab, bit).


%  table components section
component(temp_sum_comp, table([
[ a , b , temp_sum ] ,
[ '1' , * , '1' ],
[ * , '1' , '1' ],
[ '0' , '0' , '0' ]])).


component(temp_carry_1_comp, table([
[ a , b , temp_carry_1 ] ,
[ '1' , '1' , '1' ],
[ '0' , * , '0' ],
[ * , '0' , '0' ]])).

component(ab_comp, table([
[ temp_sum , carry_in , next_ab ] ,
[ '1' , * , '1' ],
[ * , '1' , '1' ],
[ '0' , '0' , '0' ]])).


component(temp_carry_2_comp, table([
[ temp_sum , carry_in , temp_carry_2 ] ,
[ '1' , '1' , '1' ],
[ '0' , * , '0' ],
[ * , '0' , '0' ]])).
```

```
component(carry_out_comp, table([
[ temp_carry_1 , temp_carry_2 , carry_out ] ,
[ '1' , * , '1' ],
[ * , '1' , '1' ],
[ '0' , '0' , '0' ]])).

component(comp_reg_ab, reg(input(next_ab), output(ab))).

%---------- initial state section ----------

%---------- variables initial values section ----------
init_val(ab, '0').

%---------- outputs signals section ----------
outputs([
carry_out
]).

%---------- output partition section ----------
output_partition([
[[carry_out]]
]).

%---------- next state variable section ----------
next_state_partition([
[[next_ab]]
]).

%---------- state variable and next state variable mapping ----------
st_nxst(ab, next_ab).
```

figure 7.14 Machine à état abstrait de la description structurale de l'additionneur

7.5 Les autres aspects de la traduction

7.5.1 L'ordonnancement des symboles

Aux règles d'ordonnancement énoncées dans la section 4.1.1, nous ajoutons les suivantes:

- tous les signaux d'entrée d'un composant donné viennent avant tous ses signaux de sortie.
- étant donné un ordre sur mesure entre deux variables x et y, cet ordre est même entre les variables d'état suivant correspondantes x' et y'.

La relation d'ordre entre les symboles, permet de construire des graphes de dépendance acycliques entre les symboles des composants de la structure plate. Nous appliquons un tri topologique sur chaque graphe pour obtenir un ordre des symboles sur ce graphe. L'ordre pour tous les symboles doit être compatible avec l'ordre de tous les graphes de dépendance. L'ordonnancement des symboles obtenu de cette façon n'est pas optimal à priori.

Notons < la relation *vient avant* qui exprime l'ordre entre deux symboles. Le terme croisé *eq(y1, y2)* induit les relations d'ordre suivantes $y1 < eq$ et $y2 < eq$, qui constituent deux arcs du graphe de dépendance. On peut construire à partir du composant *n_y1* dé de la figure 4.8 un graphe de dépendance dont les arcs traduisent les relations d'ordre suivantes: $y0 < n\_y1$, $y1 < eq$, $y2 < eq$, $x0 < n\_y1$, $y1 < n\_y1$, $y2 < n\_y1$, $y1 < eq$, $y2 < eq$, $y1 < sub$ et $y2 < sub$. Le calcul des relations d'ordre est effectué pour tous les termes croisés et composants, puis on en déduit un ordre total compatible à tous ces relations d'ordre. La figure 4.9 donne une liste ordonnée pour tous les symboles de machine à état abstrait GCD.

7.5.2 Le contrôle sémantique

Il est effectué à toutes les étapes de la traduction, pour vérifier principalement les propriétés de la machine à état à savoir:

- La machine à état est déterministe,

- La machine à état est complètement spécifiée,
- La machine à état est synchronisée avec un signal d'horloge.

7.6 Résultats expérimentaux

Nous décrivons brièvement dans cette section les principes généraux de notre implantation, et donnons un tableau comparatif de quelques caractéristiques de nos exemples. Le code et les exemples sont présentés dans [2].

7.6.1 Implantation

L'implantation est basée essentiellement sur des listes chaînées et des algorithmes récursifs. Nous avons choisi de construire les langages intermédiaires en utilisant des listes doublement chaînées pour faciliter le ramassage des miettes (*garbage collection*). Les algorithmes récursifs sont nécessaires pour ne pas figer les paramètres des circuits, par exemple le nombre de niveau dans une structure hiérarchique.

Le code compte environ vingt mille (20,000) lignes reparties dans onze fichiers, dont un contient les déclarations des structures de données. Il comporte au niveau le plus élevé quatre modules, appelés dans l'ordre suivant:

1. Le module d'initialisation. Il demande à l'utilisateur des informations qui permettent d'identifier les unités de bibliothèques du design VHDL et la bibliothèque qui les contient, ouvre la bibliothèque, et retourne des pointeurs sur les unités de bibliothèque.
2. Le modules d'acquisition des déclarations et des annotations de sorts.

3. Le module d'analyse de la représentation abstraite du design VHDL, et de synthèse du langage cible. Il produit les relations de transition et de sortie de la machine à état.
4. Le module de génération du code de la machine à état abstrait.

7.6.2 Les exemples

Nous présentons dans la table 7.3 des caractéristiques comparatives de quatre exemples que nous avons expérimentés avec notre traducteur. Le temps d'exécution de la traduction est estimé entre deux et cinq minutes, suivant les exemples. Nous ne pouvons donné un temps d'exécution exacte, car la licence du logiciel LEDA est actuellement expirée, ce qui rend les exécutions impossibles.

Exemple	Taille du fichier VHDL (octets)	Taille des fichiers MDG-HDL (oct.)	Observations
Compteur	**2160**	**5180**	**- machine à état implicite - deux processus**
Vérificateur de parité	**1010**	**2320**	**- machine à état explicite - un processus - tous les sorts sont concrets**
Contrôleur de lecture-écriture	**3310**	**7360**	**- Description structurale - Deux niveaux**
Additionneur	**3630**	**2910**	**- Description structurale - trois niveaux - tous les sorts sont concrets**

table 7.3 Exemples de traductions réalisées

Les mécanismes de la traduction, et les résultats expérimentaux présentés dans ce chapitre, terminent le résumé des travaux réalisés au cours du projet. Le chapitre suivant conclut le mémoire.

Chapitre VIII

Conclusions et perspectives

Ce chapitre conclut le mémoire et donne des orientations possibles pour des travaux ultérieurs.

8.1 Conclusions

Nous avons décrit dans ce mémoire un traducteur des modèles VHDL de machines à état fini au niveau RTL, supportant une description hiérarchique, vers des machines à état abstrait à structure plate, distinguant les signaux selon leur sort.

Nous avons défini un sous-ensemble de VHDL compilable vers les modèles MDG. Nous avons associé des sorts aux types prédéfinis de VHDL, défini des règles pour déterminer le sort d'une classe de types, et défini des règles d'annotation pour définir les sorts pour les autres types de données. Le traducteur supporte la totalité du sous-ensemble de VHDL décrit dans l'annexe A.

Nous avons choisi et adapté une sémantique opérationnelle des langages impératifs à la description des comportements de machines à état VHDL, afin de parvenir à réaliser la traduction entre deux modèles de machines à état dont les syntaxes sont totalement différentes, en la basant sur cette sémantique.

Les algorithmes développés sont indépendants du nombre de processus utilisés pour décrire la machine à état, et du nombre de niveau dans la hiérarchie des descriptions VHDL. La complétude de chaque relation de transition et de sortie de la machine à état est vérifiée.

Un tri topologique opéré sur un graphe de dépendance acyclique des symboles, permet d'obtenir un ordre total satisfaisant les contraintes d'ordre total sur mesure de tous les symboles qui apparaissent dans le modèle MDG.

Nous avons réalisé plusieurs traductions de modèles comportementaux et structuraux de descriptions VHDL avec succès: compteur, vérificateur de parité, additionneur, et contrôleur de lecture-écriture. Les caractéristiques comparatives des exemples sont présentées dans la section 7.6.2 et les exemples complets figurent dans [2].

8.2 Perspectives

Nous abordons dans cette section, les points que nous n'avons traités et qu'ils peuvent faire l'objet de prochains travaux de recherche.

1. L'établissement d'une preuve de la traduction peut être réalisée en prouvant la justesse de la correspondance des transitions dans les deux modèles. Les transitions dans les modèles VHDL sont décrites par la sémantique exposée. Pour compléter la preuve, il faut définir une sémantique opérationnelle des machines à état abstraits et une fonction de correspondance adéquate.
2. L'ordre des variables obtenu n'est pas optimal dans le cas général. La recherche de critères à appliquer pendant le tri topologique peut permettre d'optimiser l'ordonnancement des variables.

3. Nous avons décrit les modèles VHDL, par des machines à états finis et une spécification des sorts des signaux par des attributs utilisateurs. La définition d'une machine à état fini comme une interprétation d'une machine à état abstrait, pourrait être mieux représentée par un modèle VHDL dans le lequel, les types et les opérateurs sont génériques (abstraits), de telle sorte qu'une instantiation des types et des opérateurs abstraits de VHDL corresponde à une interprétation de la machine à état abstrait dans le modèle MDG-HDL. L'approche décrite dans [13], peut servir de base.

4. Les expressions conditionnelles sont sous une forme normale disjonctive, qui ne permet que les opérateurs, *et*, *ou* et, *non* et les opérateurs binaires de comparaison. Nous effectuons l'interprétation des opérateurs *et*, *ou* et, *non*, et le cas particulier où l'opérateur binaire de comparaison est l'égalité ou la différence. L'interprétation des opérateurs binaires de comparaison dans le cas général, ainsi que celle des expressions arithmétiques pourraient faire l'objet de travaux ultérieurs. Nous suggérons que ces interprétations soient réalisées dans le paquetage MDG de façon qu'elles soient disponibles pour toutes les applications utilisant le système de vérification MDG.

Bibliographie

[1] A. V. Aho, R. Sethi, J. Ullman. *Compilateurs: principes, techniques et outils*. Paris: InterEditions, 1989.

[2] K. D. Anon. *VHDL to MDG-HDL translator code and examples*. Université de Montréal, 1997.

[3] K.D. Anon, N. Boulerice, E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Y. Xu, Z. Zhou. *MDG Tools for the verification of RTL Designs*. In *Proc. of Conference on Computer-Aided Verification (CAV'96)*. New Jersey, USA, July 1996.

[4] R. K. Brayton. *Lecture Notes in Computer Science*. Volume1166: *VIS*. Springer-Verlag, 1996.

[5] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMilan, D. L. Dill. *Symbolic Model Checking for sequential circuit verification*. In *Transactions on Computer Aided Design of Integrated Circuits and Systems*. Vol. 13, No. 4, April 1994.

[6] O. Coudert, C. Berthet. *Lecture Notes in Computer Science*. Volume 407: *Verification of synchronous sequential machines based on symbolic execution*. Springer-Verlag, 1989.

[7] D. Déharbe, D. Borrionne. *Symbolic Model CHecking of VHDL Design entities*. Report, Laboratoire Artemis, 1993.

[8] G. Dohmen, R. Hermann. *A Deterministic Finite-state Model for VHDL*. Kluwer Academic Publishers, 1995.

[9] R. Enders, T. Filkorn, D. Taubner. *Lecture Notes in Computer Science*. Volume 575: *Generating BDDs for Symbolic Model Checking in CCS*. Springer-Verlag, 1991.

[10] A. Ghosh, S. Devadas, A. R. Newton. *Sequential logic testing and verification*. Kluwer Academic Publishers, 1992.

[11] A. Gupta. *Formal, Hardware verification Methods: A Survey*. Formal methods in System Design, 1992.

[12] I*EEE: Standard VHDL Language Reference Manual*. IEEE Standard 1076-1987, 1987.

[13] M. Israel, J. Benzakki, M. François. *Introduction of abstract types and genericity in VHDL*. In *Proceedings of second second european conference on VHDL methods*, 1991.

[14] K. Keutzer. *Lecture Notes in Computer Science*.Volume 1166: *The Need for Formal Methods for Integrated Circuit Design*. Springer-Verlag 1996.

[15] D. W. Knapp. *Behavioral Synthesis, Digital System Design using the Synopsys Behavioral Compiler*. Prentice Hall PTR 1996.

[16] R. Kumar, T. Kropf, K. Schneider. *First Steps towards Automating Hardware Proofs in HOL.* In *Proceedings of the 1991 Int. Workshop on the HOL Proving System and its applications*. IEEE Computer Society Press.

[17] LEDA: A*PEX, Atomic Property Extractor for Synthesis, Implementor's Guide*, Version B.1.0. LEDA.S.A. 1995.

[18] LEDA: *GRAPHGEN Control/Data Flow Graph Generator for Full VHDL. Implementor's Guide*, Version B.1.0. LEDA.S.A. 1994.

[19] LEDA: *LEDA VHDL System Implementor's guide*. Version 3.2.0. LEDA.S.A. 1993.

[20] LEDA: *LEDA VHDL System User's Manual, Unix Version.* Version 3.2.0. LEDA.S.A. 1993.

[21] R. Lipsett. *VHDL Hardware Description and Design*. Kluwer Academic Publishers, 1990.

[22] P. Loewenstein. *Learning to use HOL.* In *Proceedings of the 1991 Int. Workshop on the HOL Proving System and its applications*. IEEE Computer Society Press.

[23] K. L. McMillan. *Symbolic Model Checking, An approach to the state explosion problem.* Kluwer Academic Publishers, 1993.

[24] Z. Navabi. *VHDL: Analysis and modeling of digital systems*. New York, Montreal: McGraw-Hill, 1993.

[25] K. Schneider, T. Kropf. *Lecture Notes in Computer Science.*Volume 1166: *A Unified Approach for combining Different Formalism for Hardware Verification.* Springer-Verlag 1996.

[26] N. Shankar. *Lecture Notes in Computer Science.*Volume 1166: *PVS: Combining Specification, Proof Checking, and Model Ckecking*. Springer-Verlag 1996.

[27] W. M. Waite, G. Goos. *Compiler construction*. New York: Springer-Verlag, 1984.

[28] R. Wilhelm. *Les compilateurs: théorie, construction, génération.* Paris: Masson, 1994.

[29] G. Winskel. T*he formal semantics of a programming language, an introduction*. MIT Press 1993.

[30] Z. Zhou. *Multiway Decision Graphs and their Applications in Automatic Formal Verification of RTL Design*. Ph. D thesis Dept. IRO, Université de Montréal Québec Canada, Décembre 1996.

[31] Z. Zhou, X. Song, F. Corella, E. Cerny, M. Langevin. *Partitioning transition relations automatically and efficiently.* In *IEEE Proceedings of Fifth Great Lakes Symposium on VLSI (GLSVLSI'95).* March 1995, Buffalo, USA.

[32] Z. Zhou, X. Song, F. Corella, E. Cerny, M. Langevin. *Description and verification of RTL designs using Multiway Decision Graphs*. In *Proc. of IFIP Conference on Hardware Description Languages and their Applications (CHDL'95)*. August 1995, Chiba, Japan.

[33] Z. Zhou, N. Boulerice. *MDG Tools (V1.0) User's Manual.* Université de Montréal, 1995.

[34] Z. Zhou. *MDG Tools (V1.0) Developer's manua*l. Université de Montréal 1995.

[35] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, M. Langevin. *Formal verification of the Island Tunnel Controller using Multiway Decision Graphs*. In *Proc. of International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*.

Appendice A

Le sous-ensemble VHDL

VHDL Subset

We present below the syntax production rules of the VHDL subset used. This presentation follows the structure of the VHDL Language Reference Manual [12].

The syntax production rules respect the following conventions: | separate different alternatives in its right hand side and in its left hand side, brackets [] delimit an optional element, braces { } delimit a possible empty set of element. The key words of VHDL are in a bold form and an italicized term in the text indicates definition of the term.

A.1 Design entities and configurations

A.1.1 Entity Declarations

entity_declaration ::=
    entity identifier **is**
        entity_header
        entity_declarative_part
    end [*entity*_simple_name];

A.1.1.1 Entity Header

entity_header ::=
    [*formal*_generic_clause]
    [*formal*_port_clause]

generic_clause ::=
    generic(generic_list);

port_clause ::=
    port(port_list);

generic_list ::= *generic*_interface_list

port_list ::= *port*_interface_list

A.1.1.2 Entity Declarative Part

entity_declarative_part ::=
    {entity_declarative_item}

entity_declarative_item ::=
    function_declaration
    | function_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

A.1.2 Architecture Bodies

architecture_body ::=
architecture identifier **of** *entity*_name **is**
architecture_declarative_part
begin
architecture_statement_part
end [*architecture*_simple_name];

A.1.2.1 Architecture Declarative Part

architecture_declarative_part ::=
{block_declarative_item}

block_declarative_item ::=
function_declaration
| function_body
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| component_declaration
| attribute_declaration
| configuration_specification
| attribute_specification
| use_clause

A.1.2.2 Architecture Statement Part

architecture_statement_part ::=
{concurrent_statement}

A.2 Functions and Packages

A.2.1 Functions Declarations

```
function_declaration ::=
    function_specification;

function_specification ::=
    function designator [(formal_parameter_list)] return
    type_mark

designator ::= identifier

operator_symbol ::= string_literal
```

A.2.1.1 Formal Parameters

formal_parameter_list ::= *parameter*_interface_list

A.2.2 Function Bodies

```
function_body ::=
    function_specification is
        function_declarative_part
    begin
        function_statement_part
    end [designator];


function_declarative_part ::=
    {function_declarative_item}

function_declarative_item ::=
    function_declaration
    | function_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
```

A.2.3 Package Declarations

```
package_declaration ::=
    package identifier is
        package_declarative_part
    end[package_simple_name];

package_declarative_part ::=
    {package_declarative_item}

package_declarative_item ::=
    function_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    |use_clause
```

A.2.4 Package Bodies

```
package_body ::=
    package body package_simple_name is
        package_body_declarative_part;
    end[package_simple_name];

package_body_declarative_part ::=
    {package_body_declarative_item}

package_body_declarative_item ::=
    function_declaration
    | function_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | use_clause
```

A.3 Types

A.3.1 Scalar Types

scalar_type_definition ::=
    enumeration_type_definition
    | integer_type_definition

range_constraint ::= **range** range

range ::=
    *range*_attribute_name
    | simple_expression direction simple_expression

direction ::= **to** | **downto**

A.3.1.1 Enumeration types

enumeration_definition ::=
    (enumeration_literal {, enumeration_literal})

enumeration_literal ::= identifier | character_literal

The predefined enumerated types are CHARACTER, BIT and BOOLEAN.

A.3.1.2 Integer Types

integer_type_definition ::= range_constraint

A.4 Declarations

declaration ::=
    type_declaration
    | subtype_declaration
    | object_declaration
    | interface_declaration
    | attribute_declaration
    | component_declaration
    | entity_declaration
    | function_declaration
    | package_declaration

A.4.1 Type Declarations

type_declaration ::= full_type_declaration

type_definition ::= **type** identifier **is** type_definition;

type_definition ::= scalar_type_definition
    scalar_type_definition

A.4.2 Subtype Declarations

subtype_declaration ::=
    subtype identifier **is** subtype_indication;

subtype_indication ::=
    type_mark[constraint]

type_mark ::=
    type_name
    | *subtype*_name

constraint ::=
    range_constraint
    | index_constraint

A.4.3 Objects

A.4.3.1 Object Declarations

object_declaration ::=
constant_declaration
| signal_declaration
| variable_declaration

constant_declaration ::=
constant identifier_list :subtype_indication [:= expression];

signal_declaration ::=
signal identifier_list: subtype_indication [:= expression];

variable_declaration ::=
variable identifier_list : subtype_indication[:= expression];

A.4.3.2 Interface Declaration

interface_declaration ::=
    interface_constant_declaration
    | interface_signal_declaration

interface_constant_declaration ::=
    [**constant**] identifier_list : [**in**] subtype_indication [:= *static*_expression]

interface_signal_declaration ::=
    [**signal**] identifier_list : [mode] subtype_indication [:= *static*_expression]

interface_variable_declaration ::=
    [**variable**] identifier_list: [mode] subtype_indication[:= *static*_expression]

mode ::= **in** | **out**

- Interface Lists

interface_list ::=
    interface_element {; interface_element}

interface_element ::= interface_declaration

- Association Lists

    association_list ::=
        association_element {, association_element}

    formal_part ::=
        formal_designator
        | *function*_name(formal_designator)

    formal_designator ::=
        *generic*_name
        | *port*_name
        | *parameter*_name

    actual_part ::=
        actual_designator
        | *function*_name(actual_designator)

    actual_designator ::=
        expression
        | *signal*_name
        | *variable*_name

A.4.4 Attribute Declarations

attribute_declaration ::=
attribute identifier : type_mark;

A.5 Specifications

A.5.1 Attribute Specification

attribute_specification ::=
attribute attribute_designator **of** entity_specification **is** expression

entity_specification ::=
entity_name_list : entity_class

entity_class ::=
type

*entity*_name_list ::=
entity_designator {,entity_designator}
entity_designator ::= simple_name | operator_symbol

entity_designator ::= simple_name | operator_symbol

A.5.2 Configuration Specification

configuration_specification ::=
for component_specification **use** binding_indication

component_specification ::=
instantiation_list : *component*_name

instantiation_list ::=
*instantiation*_label

binding_indication ::=
entity *entity*_name[(*architecture*_identifier)]

A.6 Names

A.6.1 Names

name ::=
simple_name
| operator_symbol
| indexed_name
| slice_name
| attribute_name

prefix ::=
name
| function_call

A.6.2 Simple Names

simple_name ::= identifier

A.6.3 Indexed Names

Indexed_name ::= prefix(expression {, expression})

A.6.4 Slice Names

slice_name ::= prefix (discrete_range)

A.6.5 Attributes Names

attribute_designator ::= *attribute*_simple_name

A.7 Expressions

A.7.1 Expressions

*dnf*_expression ::=
    *cnj*_relation {**or** *cnj*_relation}

*cnj*_relation ::=
    simple_expression [**and** simple_expression]

simple_expression ::=term

term ::= primary
    | **not** primary

primary ::=
    name
    | literal
    | function_call
    | (simple_expression)

A.7.2 Operands

A.7.2.1 Literals

literal ::=
numeric_literal
| enumeration_litteral
| string_literal
| bit_string_literal

numeric_literal ::=
abstract_literal

A.7.2.2 Function Calls

function_call ::=
*function*_name[(actual_parameter_part)]

actual_parameter_part ::= *parameter*_association_list

A.8 Sequential Statement

sequence_of_statement ::=
{sequential_statement}

sequential_statement ::=
wait_statement
| signal_assignment_statement
| variable_assignment_statement
| if_statement
| case statement
| return_statement
| null_statement

A.8.1 Wait Statement

wait_statement ::=
wait on *clock*_signal
| **wait until** *clock*_signal'**event** and *clock*_signal_condition

*clock*_signal_condition ::= *clock*_signal = '0' | *clock*_signal = '1'

A.8.2 Signal Assignment

signal_assignment_statement ::=
    target <= waveform

target ::=
    name

waveform ::= waveform_element {, waveform_element}

A.8.3 Updating a projected Output Waveform

waveform_element ::=
    *value*_expression

A.8.4 Variable Assignment Statement

variable_assignment_statement ::=

target := expression

A.8.5 If Statement

if_statement ::=
    if *dnf*[1]_condition **then**
        sequence_of_statements
    {**elsif** *dnf*_condition **then**
        sequence_of_statement}
    [**else**
        sequence_of_statements]
    end if;

---

1. stands for disjunctive normal form expression

A.8.6 Case statement

```
case_statement ::=
    case dnf[1]_expression is
        case_statement_alternative
        {case_statement_alternative}
    end case;

    case_statement_alternative ::=
        when choices =>
        sequence_of_statements
```

1. stands for disjunctive normal form expression

A.8.7 Return Statement

```
return_statement ::=
    return [expression];
```

A.8.8 Null Statement

```
null_statement ::= null;
```

A.9 Concurrent Statements

```
concurrent_statement ::=
    process_statement
    | concurrent_signal_assignment_statement
    | component_instantiation_statement
```

A.9.1 Process Statement

process_statement ::=
[*process*_label]
process [(sensitivity_list)]
process_statement_part
begin
process_statement_part
end process [*process*_label];

process_declarative_part ::=
{process_declarative_item}

process_declarative_item ::=
function_declaration
| function_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| attribute_declaration
| attribute_specification
| use_clause

process_statement_part ::=
{sequential_statement}

A.9.2 Component Instantiation Statement

component_instantiation_statement ::=
*instantiation*_label :
*component*_name
port_map_aspect

port_map_aspect ::=
port map (*port*_association_list)

A.10 Design units

A.10.1 Design units

design_unit ::= context_clause library

library_unit ::=
    primary_unit
    | secondary_unit

primary_unit ::
    entity_declaration
    | package_declaration

secondary_unit ::=
    architecure_body

A.10.2 Design Libraries

library_clause ::= **library** logical_name_list;

logical_name_list ::= logical_name{, logical_name}

logical_name ::= identifier

A.10.3 Context Clauses

context_clause ::= {context_item}

context_item ::=
    library_clause
    | use_clause

Appendice B

Manuel d'utilisateur du compilateur VHDL-MDG

VHDL-MDG Compiler user's Manual

B.1 Getting started

Before using the VHDL-MDG compiler, the following operations should be performed:

1. Build the predefined libraries (STD and IEEE). Use the following command in the Unix command line.
   v createstd -k 93

2. Create a new working library
   v newlib <new_working_library_name>

3. Adds references to STD library
   v add -k 93 STD

4. Add references to IEEE library
   v add -k 93 IEEE

5. Compile entity and architecture files in the following order.
   v comp <entity_file>

   v comp <architecture_file>

6. Run APEX and GRAPHEN to extend the schema
   mygphapex <LibraryUnit> <architecture_unit>/<entity_unit>

B.2 Compiler usage

First, run the **fsm** command from the Unix command line. Then you will be asked for various information to enter after the colon. There is always the last used information between brackets.

ASM> **Library name** [*../Lib/Lib95/Lib95*] :

ASM> **Entity Library Unit name** [*counter*] :

ASM> **Architecture Library Unit name** [*alg_counter*] :

Mdg files directory[*Run/Counter*] :

Algebraic specification file name[*counter_alg.pl*] :

Circuit specification file [*counter_cir.pl*] :

Symbol Order file [*counter_ord.pl*] :

B.3 Complete Example

We describe below a complete sequence of commands for the VHDL description in Figure 7.1.

1. **v createstd -k 93**
2. **v createstd -k 93** *mealy_lib*
   where *mealy_lib* is the name of the target library for the VHDL descriptions files
3. **v add -k 93 STD**
4. **v add -k 93 IEEE**
5. **v comp** *mealy_b.vhd*
   where *mealy_b.vhd* is the name of the file that contains the VHDL description in Figure 7.1.
6. **mygphapex** *mealy_lib alg_mealy_b/mealy_b*
   alg_mealy_b is the name of the architecture for the entity *mealy_b.*
7. run the ***fsm*** command, then enter various information
8. ASM> **Library name** [*../Lib/Lib95/Lib95*] : *mealy_lib*
9. **Entity Library Unit name** [*counter*] : *mealy_b*

10. ASM> **Architecture Library Unit name** [*alg_counter*] : *alg_mealy_b*

11. **Mdg files directory**[*Run/Counter*] : *Mdgfiles*
    where *Mdgfiles* directory exists.

12. **Algebraic specification file name**[*counter_alg.pl*] :*mealy_b_alg.pl*

13. **Circuit specification file** [*counter_cir.pl*] : *mealy_b_cir.pl*

14. **Symbol Order file** [*counter_ord.pl*] : *mealy_b_ord.pl*