Validation de descriptions VHDL fondée sur des techniques issues du domaine du test de logiciels

Christophe Paoli

UNIVERSITÉ DE CORSE – PASQUALE PAOLI
U.F.R. SCIENCES ET TECHNIQUES

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

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE CORSE

ÉCOLE DOCTORALE ENVIRONNEMENT ET SOCIÉTÉ

Discipline : Science pour l'Environnement

Spécialité : Informatique

présentée et soutenue publiquement

par

Christophe PAOLI

le 20 décembre 2001

Titre :

Validation de descriptions VHDL fondée sur des techniques issues du domaine du test de logiciels

_____

Directeur de thèse :

Jean-François SANTUCCI

_____

devant le jury ci-dessous

Rapporteurs	M. Denis FLOUTIER, Professeur, École Supérieure d'Ingénieurs d'Annecy
	M. Andrzej RUCINSKI, Professor, University of New Hampshire
Examinateurs	M. Paul BISGAMBIGLIA, Maître de Conférence (HDR),Université de Corse
	Mlle Marie-Laure NIVET, Maître de Conférence, Université de Corse
	M. Jean-François SANTUCCI, Professeur, Université de Corse
	M. Matteo SONZA REORDA, Professore Straordinario, Politecnico di Torino

à mes Parents,
à ma Famille,
à mes Amis.

REMERCIEMENTS

Cette thèse s'est déroulée au sein de l'équipe Modélisation Informatique du laboratoire Systèmes Physiques pour l'Environnement (Unité Mixte de Recherche CNRS n°6134) de l'Université de Corse. Ce travail a été réalisé grâce au soutien financier de la Collectivité Territoriale de Corse et de l'EOARD (*European Office of Aerospace Research and Development*). Que ces deux institutions trouvent ici le témoignage de ma reconnaissance.

Je tiens tout particulièrement à exprimer ma plus profonde gratitude à Monsieur Jean-François SANTUCCI, Professeur à l'Université de Corse et Directeur du laboratoire Systèmes Physiques pour l'Environnement qui m'a accueilli au sein de son laboratoire, pour avoir dirigé ces travaux et m'avoir soutenu dans cette étude. Sa disponibilité et ses conseils ont été indispensables à la concrétisation de cette recherche.

À Monsieur Denis FLOUTIER, Professeur à l'École Supérieure d'Ingénieurs d'Annecy, et Monsieur Andrzej RUCINSKI, Professeur à l'Université du New Hampshire, j'adresse ma plus respectueuse reconnaissance pour l'intérêt qu'ils ont porté à ce travail en acceptant d'en être les rapporteurs dans ce jury.

Que Monsieur Matteo SONZA REORDA, *Professore Straordinario* à l'Institut *Politecnico di Torino*, qui me fait l'honneur de participer à ce jury, trouve ici l'expression de ma profonde gratitude.

Je voudrais également exprimer toute ma reconnaissance à Monsieur Paul BISGAMBIGLIA, Maître de Conférence (HDR) à l'Université de Corse, et Mademoiselle Marie-Laure NIVET, Maître de Conférence à l'Université de Corse pour leur grande disponibilité et pour avoir accepté de participer à ce jury.

Je remercie tous les doctorants, chercheurs, et personnels de l'Université de Corse pour leur gentillesse et leur contribution à l'élaboration de ce travail.

TABLE DES MATIÈRES

CHAPITRE 1 : INTRODUCTION GENERALE 1

1.1 CONTEXTE ET MOTIVATION DE L'ETUDE 1

1.2 CONTRIBUTION 4

1.3 ORGANISATION DU DOCUMENT 7

CHAPITRE 2 : LA VALIDATION DE DESCRIPTIONS HDL 8

2.1 NOTION DE VUES ET HIERARCHIE D'ABSTRACTION 8

2.2 VALIDATION ET VERIFICATION DE DESCRIPTIONS HDL DE HAUT NIVEAU 11

2.3 LES TEST-BENCH 13

2.4 ÉTAT DE L'ART 14

2.5 PRESENTATION DE NOTRE APPROCHE 17

CHAPITRE 3 : LE TEST DE LOGICIELS 19

3.1 INTRODUCTION 19

3.2 CLASSIFICATION DES METHODES DE TEST 21

3.2.1 Le test fonctionnel 23
3.2.2 Le test structurel 24
3.2.3 Le test structurel basé sur un critère de couverture 25

3.3 LES TECHNIQUES DE MCCABE : LE TEST STRUCTURE 26

3.3.1 Le graphe flot de contrôle 26
3.3.2 La complexité cyclomatique : v(G) 28
3.3.3 Le critère du test structuré 31
3.3.4 L'algorithme de Poole 32

3.4 LA GENERATION AUTOMATIQUE DES DONNEES DE TEST 35

3.5 CONCLUSION 39

CHAPITRE 4 : APPROCHE DE RESOLUTION DU PROBLEME 41

4.1 CONCEPTS DE BASE DU LANGAGE VHDL 42

4.1.1 Les principaux éléments de l'architecture 44
4.1.1.1 Le *process* 44
4.1.1.2 L'instruction wait 45
4.1.1.3 Les différentes classes d'objets 45
4.1.2 La simulation des descriptions VHDL 47
4.1.2.1 La phase d'initialisation 48
4.1.2.2 La phase d'exécution 48
4.1.2.3 Le cycle de simulation 48
4.1.3 Un sous ensemble de VHDL pour des spécifications comportementales 49
4.1.3.1 Type de données 50
4.1.3.2 Les opérateurs 50
4.1.3.3 Les références au temps 50
4.1.3.4 Les process 51
4.1.3.5 Les fonctions et les procédures 52

4.2 GRAPHE DE FLOT DE CONTROLE DE PROGRAMME VHDL 52

4.2.1 L'architecture 53
4.2.2 Le process 54
4.2.3 Les instructions séquentielles 55
4.2.3.1 Les affectations 55
4.2.3.2 Les instructions conditionnelles 55
4.2.3.3 Les instructions itératives 57

4.3 GENERATION ET ANALYSE DES CHEMINS 60

4.3.1 La définition d'un chemin d'exécution 61
4.3.2 Les chemins à ordonnancer 63
4.3.2.1 Le graphe de dépendance 65
4.3.2.2 La liste d'ordonnancement 66
4.3.3 Les chemins à modifier 69
4.3.3.1 Généralisation 75
4.3.3.2 La liste des chemins solution 75

4.4 TRADUCTION DE NOTRE PROBLEME DE GENERATION DE DONNEES VERS UN CSP 76

4.4.1 Le problème de satisfaction de contraintes 77
4.4.2 Le modèle de contraintes de Vemuri et Kalynaraman 78

4.5 CONCLUSION 80

CHAPITRE 5 : LA GENERATION DES VECTEURS DE TEST 82

5.1 VUE D'ENSEMBLE 83

5.2 PRODUCTION DE LA BASE DE CHEMINS 84

5.3 ANALYSE DES CHEMINS 85

5.4 EXTRACTION DES VECTEURS DE TEST 91

5.4.1 La génération des contraintes 92
5.4.1.1 Contraintes de domaine 92

5.4.1.2 Contraintes relationnelles 93

5.4.1.3 Exemple de génération de contraintes 98

5.4.2 Résolution des contraintes 99

5.5 PRODUCTION DE LA SEQUENCE DE TEST 99

5.6 CONCLUSION 105

CHAPITRE 6 : REALISATION INFORMATIQUE ET RESULTATS 107

6.1 ARCHITECTURE GENERALE DE GENESI 108

6.2 LA STRUCTURE INTERMEDIAIRE AU FORMAT LISP 110

6.2.1 Description générale d'un lexeur-parseur 110
6.2.2 Le VHDL-1076.1 Parser/Pretty-Printer 111
6.2.3 Le format LISP 112

6.3 CONSTRUCTION DES GRAPHES ET ANALYSE DES CHEMINS 114

6.3.1 Construction des graphes 116
6.3.2 Production et analyse des chemins 118

6.4 GENERATION ET RESOLUTION DES CONTRAINTES 119

6.4.1 Aperçu général de GNU-Prolog 120
6.4.2 Le langage clp(FD) 121
6.4.3 Exemples d'utilisation 122

6.5 RESULTATS 126

6.6 CONCLUSION 127

CHAPITRE 7 : CONCLUSION GENERALE 129

7.1 LE BILAN DES TRAVAUX EFFECTUES 129

7.2 PERSPECTIVES DE RECHERCHE 132

LISTE DES TRAVAUX SCIENTIFIQUES ET PUBLICATIONS 135

REFERENCES BIBLIOGRAPHIQUES 139

TABLE DES FIGURES 151

LISTE DES TABLEAUX 153

ANNEXE 1 154

ANNEXE 2 169

CHAPITRE

1

INTRODUCTION GENERALE

1.1 Contexte et motivation de l'étude

La densité d'intégration des circuits VLSI (*Very Large Scale Integration*) atteinte au début des années 90 a permis de développer des circuits composés de plus d'un million de transistors. Une si grande complexité rend pour le moins très difficile la conception de tels systèmes en dessinant chaque transistor ou en définissant chaque signal en terme de portes logiques. De ce fait, et de par la nécessité de réduction de l'effort et du temps de conception des circuits, l'utilisation des outils de CAO (Conception Assistée par Ordinateur) microélectronique s'est avérée indispensable. Ces outils ont suivi l'évolution technologique en gagnant un niveau d'abstraction par décennie [Rouzeyre]. Ainsi, ont été commercialisés successivement des outils de dessin de masques (niveau *layout*), puis des outils de synthèse logique (niveau porte logique) et enfin, à l'heure actuelle, des outils de synthèse travaillant au niveau transfert de registres (RTL en anglais pour *Register Transfert Level*). Le niveau RTL

est un niveau intermédiaire entre le niveau le plus élevé (appelé niveau algorithmique) et le niveau porte logique.

La Figure 1-1 illustre le processus de conception typique[1] pour un circuit complexe. Des outils de CAO sont utilisés à chaque étape du processus de conception.

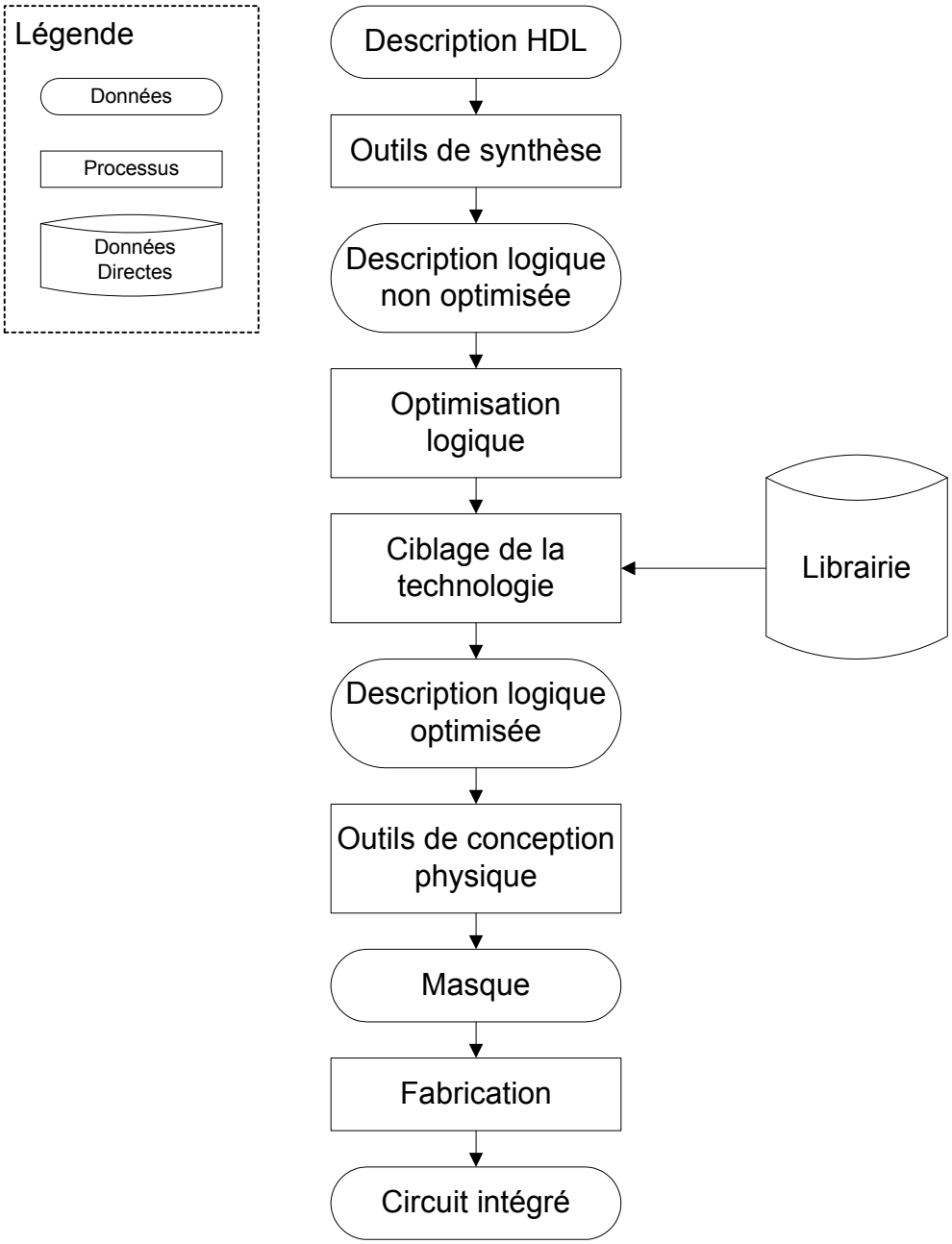


Figure 1-1. Processus de conception d'un circuit complexe.

[1] Ce processus de conception est celui utilisé lors de notre participation au projet : « Overview of Test Methods Using Boundary Scan » [DFT 1999].

Typiquement, les concepteurs de circuits commencent par l'écriture, à partir de spécifications textuelles (sous la forme d'un cahier des charges), d'une description au niveau algorithmique ou RTL de la fonctionnalité du système en utilisant un langage de description de matériel (HDL en anglais pour *Hardware Description Language*). Ensuite, les outils de synthèse sont utilisés pour transformer la description HDL de niveau RTL en une description au niveau porte logique, puis au niveau transistor, et produisent finalement le masque au niveau *layout* qui est employé pour fabriquer le circuit. Des outils de CAO sont utilisés intensivement pour exécuter les transformations entre les différents niveaux d'abstraction. Ils aident également à optimiser la description afin de prendre en compte les contraintes de vitesse, de consommation de puissance, et de surface.

Afin d'obtenir un circuit intégré qui fonctionne, le concepteur doit veiller à ce que la description HDL initiale soit correcte et qu'aucune erreur n'ait été introduite pendant les transformations aux niveaux inférieurs. Il est souhaitable de détecter les erreurs de conception le plus tôt possible. En effet, il est plus facile et moins onéreux de déceler une erreur lors de la conception de la description HDL qu'au niveau porte logique. De même, il est beaucoup moins onéreux de déceler une erreur au niveau porte logique qu'après la fabrication du circuit.

De ce fait, de par l'évolution des outils d'aide à la conception et au vu du niveau d'intégration atteint, il est maintenant nécessaire de migrer vers le niveau algorithmique. La validation de descriptions HDL avant leur synthèse est un des principaux problèmes lié à cette évolution.

Du point de vue du test, une description HDL au niveau algorithmique est similaire à un programme écrit avec un langage de programmation de haut niveau tel que C ou ADA. Cela suggère que les techniques appliquées avec succès dans le domaine du test de logiciels peuvent se révéler efficaces pour tester les conceptions de type matériel écrites avec un langage HDL [Walsh 1996]. Les validations de descriptions HDL sont en général faites par simulation voire quelque fois par preuve formelle et sont très consommatrices en temps. Elles sont de plus incomplètes, car elles ne peuvent pas être exhaustives pour des raisons économiques (temps de simulation trop importants) mais aussi pour des raisons techniques (paramètres environnementaux trop importants) [Landrault].

Parmi les nombreux HDLs, le langage VHDL (*Very High Speed Integrated Circuit* HDL) est très populaire dans le domaine de l'industrie. Standardisé par l'IEEE, il a été conçu pour être non seulement un langage de description de matériel, mais aussi un langage de conception de systèmes. VHDL possède la capacité de modéliser des circuits digitaux à différents niveaux d'abstraction. De ce fait VHDL est un langage représentatif des HDLs.

Ainsi, développer une approche efficace pour tester des programmes VHDL au niveau algorithmique et créer un logiciel qui automatise le processus de test serait une contribution significative pour les utilisateurs d'outils de CAO.

1.2 Contribution

L'objectif de cette dissertation est de développer une approche de validation à partir des descriptions VHDL décrites au niveau algorithmique. L'approche de validation retenue est basée sur la simulation. Un des principaux problèmes lié à cette approche est la génération des valeurs à appliquer en entrée de la description, appelées vecteurs de test. La génération des vecteurs de test est une des tâches les plus coûteuses et est un des problèmes le plus difficile à résoudre lors de la validation de systèmes complexes. Les descriptions de circuits au niveau algorithmique représentent une nouvelle source d'information sur le circuit qui peut être intéressante pour la génération de vecteurs de test.

Le but de ce travail est d'aider le concepteur de circuits électroniques qui désire générer des vecteurs de test à partir d'une description au niveau algorithmique afin de les utiliser pour valider une description correspondante au niveau RTL. Notre approche permet de produire ces vecteurs de test formés des stimuli (transitions des valeurs d'entrée) et des réponses attendues. Les deux descriptions (niveau algorithmique et niveau RTL) sont simulées et leurs résultats sont comparés afin de vérifier que les deux descriptions conservent le même comportement (voir Figure 1-2).

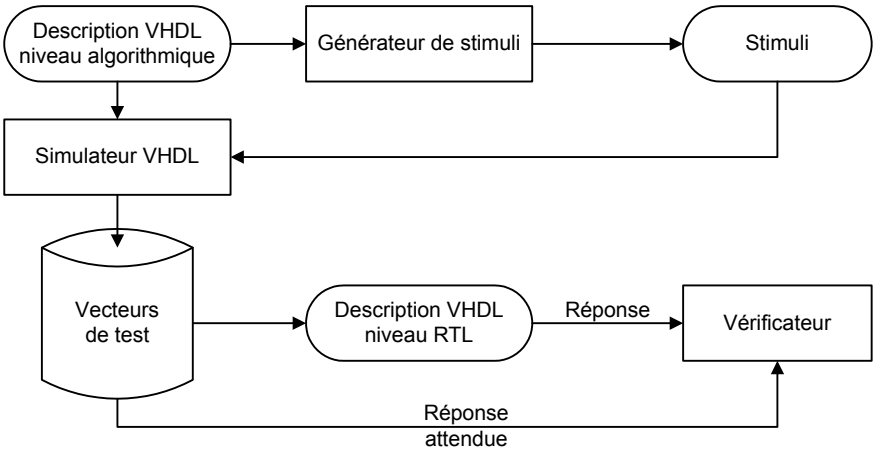


Figure 1-2. Aperçu de notre approche.

Cette thèse emprunte les techniques utilisées dans le domaine du test de logiciels. Parmi les différentes techniques de test, les méthodes de test dynamique consistent en l'exécution du programme à valider à l'aide de données qui sont fournies en entrée du programme (appelées jeux de tests). Ces données visent la détection d'erreurs dans l'implémentation du logiciel en confrontant les résultats obtenus par l'exécution du programme à ceux donnés par la spécification de l'application. Des systèmes ont été conçus pour des langages de programmation tels que le langage C, et le langage ADA, qui fournissent la couverture structurelle des programmes et génèrent automatiquement les jeux de tests pour obtenir ces couvertures [Watson 1996]. Notre approche pour la génération des vecteurs de test à partir des descriptions VHDL est basée sur le critère du test structuré. Ce critère appartient à la catégorie des techniques de test structurel (ou test de boîte blanche) où la structure du programme, c'est à dire le code, est utilisée comme base pour développer et évaluer les jeux de tests.

L'approche pour la génération de vecteurs de test, développée dans cette thèse, permet non seulement de découvrir des erreurs dans l'implémentation de la description sous test, mais peut aider à découvrir des fautes de fabrication. Ce type de faute est généralement détecté grâce à des méthodes utilisant la description physique du circuit après sa fabrication. Les vecteurs de test générés à partir d'une description au niveau algorithmique peuvent être utilisés tout au long du processus de conception, quand la description est simulée, aussi bien que lorsque le système physique est testé. Les méthodes de test visant la détection de fautes de fabrication utilisent des modèles structurels au niveau porte logique qui réclament un environnement de test lourd et coûteux (en espace mémoire par exemple) ainsi que des temps

d'exécution prohibitifs et inacceptables pour l'industrie. De plus les détails du circuit ne sont pas toujours disponibles. Il est donc crucial de chercher à pallier ces insuffisances, en définissant de nouvelles stratégies applicables lors des premières phases du processus de conception. On trouve dans la littérature ce domaine de recherche sous le nom de test comportemental. Le lecteur intéressé peut trouver une discussion dans [Sonza 1999] [Lajolo 2000] et [Federici 1999]. La détection de défaillances physiques n'est pas l'objet de notre étude.

Une autre contribution de cette étude est le désir d'automatisation du processus de conception des circuits dès le niveau algorithmique. Notre approche se veut indépendante des niveaux d'abstraction inférieurs à ce niveau, donc également indépendante des outils de synthèse.

Ce travail s'insère dans le cadre d'un projet soutenu par l'EOARD (*European Office of Aerospace Research and Development*) sous le numéro de contrat : F61775-00-C0002 ; Project 00-4005 "Validation of VHDL descriptions" (2000-2003) [Y1Q2_N°1&2&3 2000], [Y1Q3_N°4&5 2000], [Y1Q4_N°6&7 2001].

Dans cette étude, nous définissons une approche originale permettant de générer automatiquement, à partir de descriptions VHDL au niveau algorithmique, les vecteurs de test à appliquer sur une description RTL, pour effectuer sa validation par simulation. Pour accomplir cette tâche, nous proposons de développer une approche intégrant les cinq étapes suivantes :

- exploration des techniques existantes, proposées et utilisées dans le domaine du test de logiciels. Le choix d'une technique adaptée aux programmes VHDL sera effectué ;
- détermination d'un sous-ensemble VHDL englobant toutes les constructions utilisées pour des descriptions au niveau algorithmique ;
- étude d'une modélisation adéquate permettant d'appliquer les techniques de test de logiciels à des descriptions au niveau algorithmique. Ces techniques nécessitent la transformation en une structure de type graphe du programme VHDL. Cette structure est appelée modèle interne ;
- détermination d'une méthodologie pour la génération de vecteurs de test à partir du modèle interne ;

- implémentation de ces principes en utilisant l'approche orientée objet pour son caractère évolutif et pour les facilités offertes concernant la maintenabilité et la réutilisabilité du logiciel.

1.3 Organisation du document

Le reste de ce document est organisé de la façon suivante. Dans le chapitre 2 nous effectuons un état de l'art de la validation de descriptions VHDL afin de dégager les concepts fondamentaux qui ont servi de base de réflexion à ce travail.

Dans le chapitre 3 une vue d'ensemble des techniques de test utilisées dans le domaine du test de logiciels est présentée et le choix d'une technique adaptée aux descriptions de circuits digitaux est explicité. Cette technique s'appuie sur un modèle de type graphe représentant le séquencement des instructions d'un programme.

Le chapitre 4 présente la modélisation des principales constructions VHDL en vue de la génération de ce graphe. Il met en évidence les principales difficultés liées à l'application des techniques de test de logiciels à des programmes écrits en langage VHDL. Une approche de résolution du problème est également avancée dans ce chapitre.

Dans le chapitre 5 nous détaillons étape par étape notre méthodologie pour la génération des vecteurs de test. Une vue d'ensemble de notre approche est présentée ainsi que les différents algorithmes que nous avons développés, illustrés par de nombreux exemples pédagogiques.

Le chapitre 6 est consacré à l'implémentation de notre approche : nous présentons l'infrastructure du logiciel permettant de valider notre approche, sa réalisation et les premiers résultats obtenus.

Enfin, le chapitre 7 dresse le bilan des travaux effectués et les perspectives qui peuvent être données aux travaux déjà réalisés, avec en particulier le couplage de notre système de génération de vecteurs de test à un simulateur de fautes comportementales réalisé au sein de notre laboratoire.

CHAPITRE

2

LA VALIDATION DE DESCRIPTIONS HDL

Le but de ce chapitre est de situer notre domaine d'étude. Nous détaillons dans la première partie de ce chapitre les niveaux d'abstraction utilisés pour la description de circuits. La seconde partie présente la validation de descriptions HDL au niveau algorithmique, dans le contexte général du processus de conception de circuits VLSI. La troisième partie introduit la notion de *test-bench* nécessaire à la simulation des descriptions HDL. Dans la quatrième partie, nous donnons un état de l'art de la validation de descriptions HDL basée sur la simulation. Enfin la dernière partie présente notre approche.

2.1 Notion de vues et hiérarchie d'abstraction

Un modèle de circuit est une abstraction, c'est-à-dire une représentation qui donne les caractéristiques d'un circuit sans entrer dans les détails. Les modèles peuvent être classés suivant différents niveaux d'abstraction c'est-à-dire suivant différents niveaux de détail. Le diagramme en Y de [Gajski 1988a] (cf. Figure 2-1) illustre les différents types de synthèse et

les différents niveaux d'abstraction. Les axes représentent les trois domaines (ou vues) selon lesquels on peut modéliser un système digital :

- la **vue physique** qui permet de spécifier les caractéristiques physiques et géométriques du système. Dans le domaine physique on décrit les objets réels, par exemple les transistors d'un circuit.
- la **vue structurelle** dans laquelle un composant est décrit comme une interconnexion d'autres composants (structure potentielle du système physique).
- la **vue comportementale** dans laquelle un composant est décrit en définissant comment ses sorties réagissent par rapport aux entrées.

Sur chaque axe sont reportés les différents niveaux du domaine de description. Ils sont d'autant plus abstraits que l'on s'éloigne du centre du Y. Les arcs représentent les informations utilisées et produites par chaque outil de synthèse.

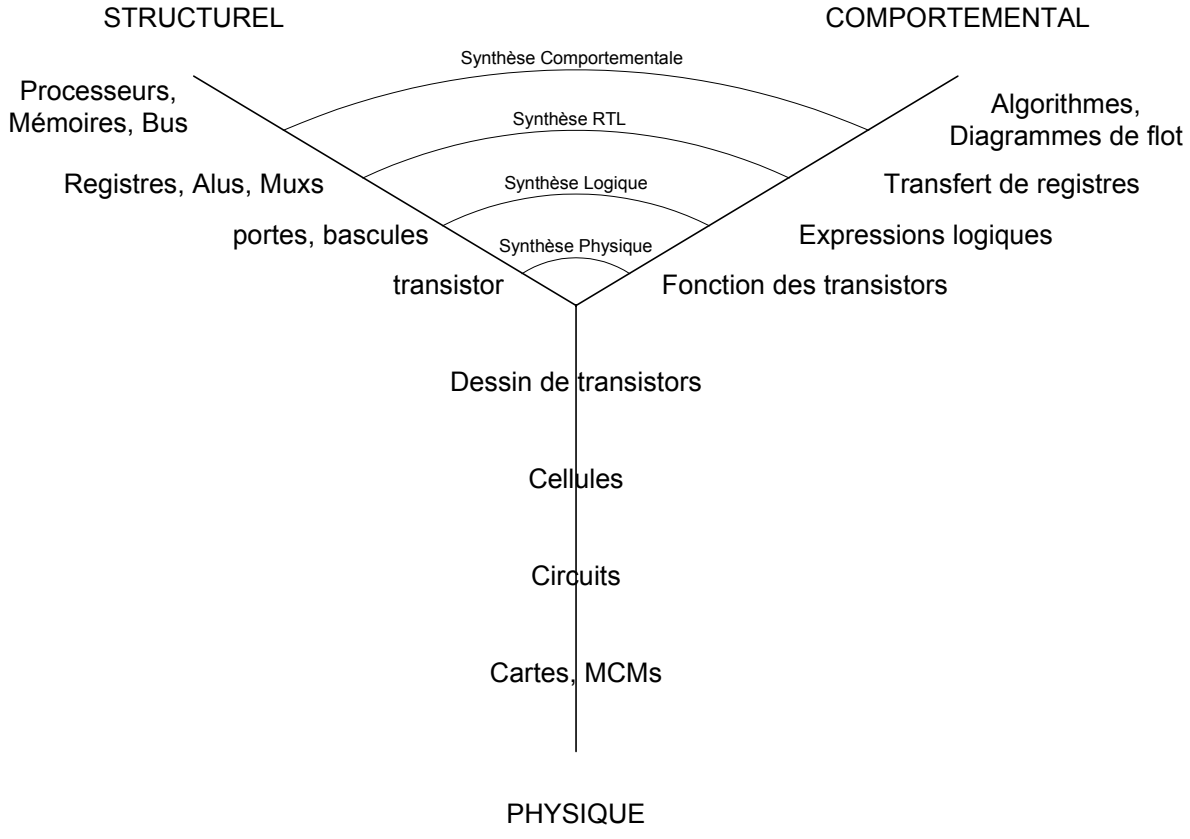


Figure 2-1. Diagramme en Y.

Dans le domaine comportemental, on s'intéresse à ce que le circuit fait et non pas à la façon dont il est conçu. On traite le circuit comme une boîte noire pour laquelle sont spécifiés

les entrées/sorties et un ensemble de fonctions décrivant le comportement de chaque sortie en fonction des entrées et du temps. On utilise des fonctions de transfert et des diagrammes d'états au niveau circuit, des expressions logiques de type booléen et des diagrammes d'états au niveau logique.

Au niveau RTL, le temps est divisé en intervalles appelés étapes de contrôle. À chaque étape de contrôle, on spécifie les conditions qui doivent être testées, tous les transferts entre registres qui doivent être exécutés et l'étape suivante.

Au plus haut niveau d'abstraction, c'est-à-dire au niveau algorithmique, on utilise des variables et des opérateurs (de type mathématique) pour exprimer la fonctionnalité des composants du système. Les variables et les structures de données ne sont pas liées à des registres ou à des mémoires particuliers pas plus que les opérateurs ne le sont à des unités fonctionnelles (de types ALU ou MUX) ou à des étapes de contrôles. À ce niveau, le temps est représenté par l'ordre dans lequel les affectations des variables sont effectuées.

Une représentation structurelle permet de passer d'une représentation comportementale à une représentation physique. Une représentation structurelle (comme un schéma logique) peut éventuellement servir de représentation fonctionnelle. D'un autre côté une représentation comportementale, par exemple une expression booléenne, suggère une implantation directe, comme une structure composée de portes NOT, AND et OR. C'est pourquoi la différentiation entre niveaux et domaines n'est pas toujours claire. De même, on retrouve souvent dans la littérature le plus haut niveau d'abstraction d'une description sous le nom de niveau comportemental, ce qui ajoute à la confusion avec la vue comportementale [Rouzeyre]. En ce qui nous concerne, le terme de niveau algorithmique est préféré au terme de niveau comportemental.

Pour plus d'information, le lecteur peut se référer à l'ouvrage de [Armstrong *et al.* 2000] qui couvre les principes de la conception des circuits digitaux et de l'utilisation des outils de synthèse.

2.2 Validation et vérification de descriptions HDL de haut niveau

La Figure 2-2 illustre le flot de conception de haut niveau d'un circuit complexe. Le concepteur du circuit dispose de spécifications textuelles du circuit (cahier des charges) qui lui permettent de décrire, à l'aide d'un langage de description de matériel (tel que Verilog ou VHDL), son comportement.

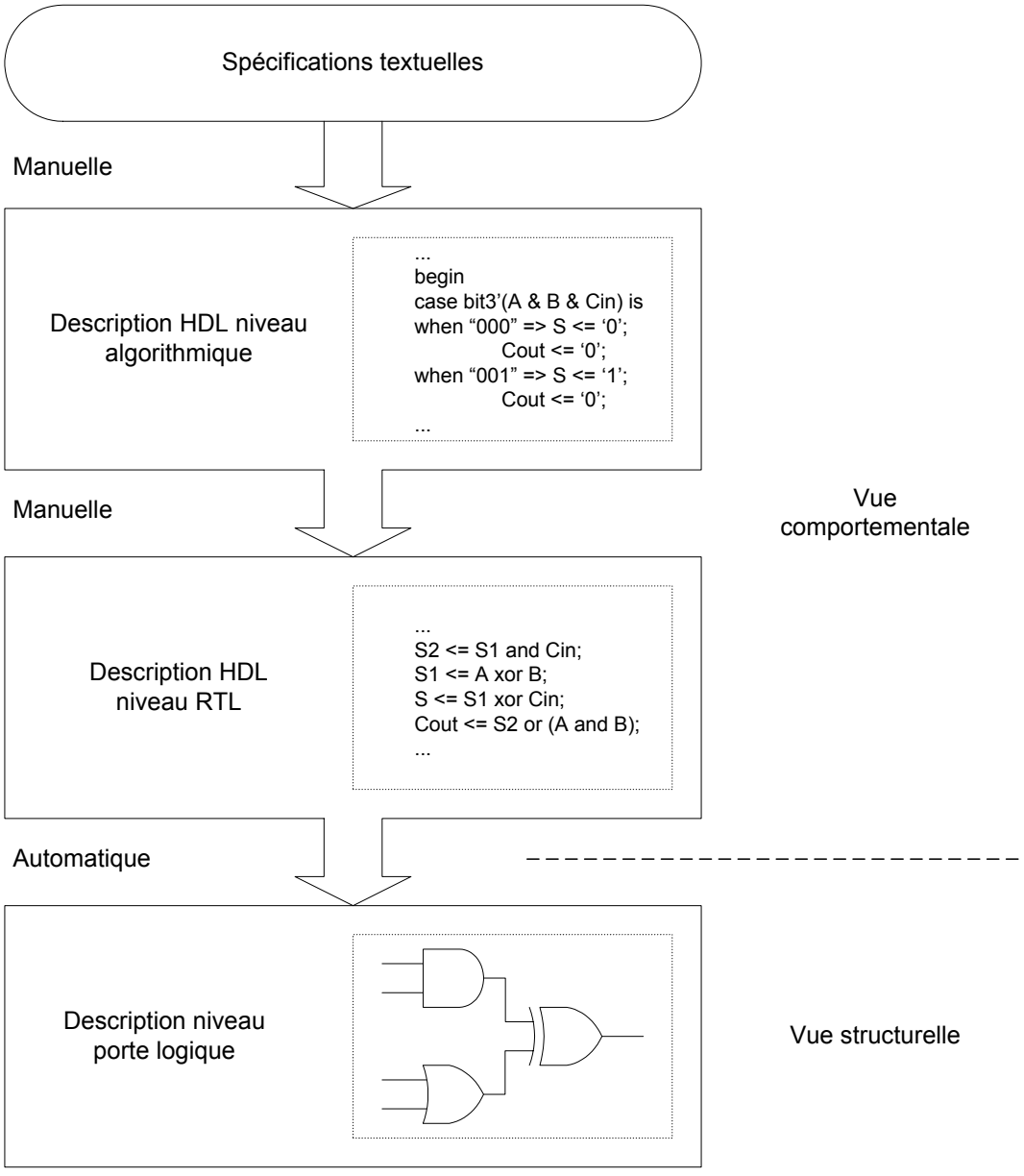


Figure 2-2. Flot de conception de haut niveau.

Cette description se présente sous la forme d'une description algorithmique. La transformation de cette première description en une description au niveau RTL est appelée synthèse architecturale (ou comportementale) et est effectuée manuellement par le concepteur.

L'étape suivante consiste à transformer cette description comportementale au niveau RTL en une description structurelle au niveau porte logique. Ce passage est appelé synthèse RTL. Des outils commerciaux bien connus de l'industrie (outils de synthèse logique) permettent de produire automatiquement de telles descriptions structurelles au niveau porte logique.

On notera l'existence très récente (5 à 6 ans d'âge) d'outils de synthèse comportementale (HLS en anglais pour *high level synthesis*). Des outils HLS commerciaux (*Behavioral Compiler - Synopsys, Visual Architect - Cadence, Monet - Mentor Graphics*) commencent à apparaître et permettent la traduction d'une description de niveau algorithmique en une description de niveau RTL ou de niveau porte logique. Cependant, à l'instar des outils de synthèse logique, qui ont mis plus de 7 ans à s'imposer dans le monde industriel, les outils de synthèse comportementale ne semblent pas aujourd'hui exploités pleinement par leurs utilisateurs potentiels. En effet, un usage efficace des outils HLS commerciaux demande une connaissance approfondie des méthodes d'optimisation qu'ils intègrent et du style de descriptions de niveau algorithmique pouvant être pris en compte. Ces impératifs limitent, de ce fait, leur utilisation.

La vérification qu'aucune erreur n'a été introduite entre le niveau algorithmique et le niveau RTL reste donc une issue importante qui exige une automatisation. On retrouve dans la littérature ce domaine d'étude sous le nom de validation fonctionnelle ou encore vérification fonctionnelle. L'IEEE [IEEE729 1983][IEEE 1994] définit l'activité de vérification comme le moyen d'établir la correspondance entre un produit et sa spécification, et la validation comme le moyen d'assurer que le produit accomplit bien la fonction pour laquelle il a été conçu. En général, le terme de vérification fait référence aux méthodes consistant à « démontrer » qu'un circuit va se comporter comme il est souhaité, alors que le terme de validation fait référence aux méthodes consistant à exciter une description de circuit par une série de stimuli. On peut donc classer les nombreuses méthodes proposées suivant deux approches :

- les approches de type vérification formelle ;
- les approches basées sur la simulation de la description sous test.

Malgré les progrès accomplis, la vérification formelle qui propose de prouver mathématiquement l'exactitude d'une description n'est seulement réalisable que pour de petites descriptions. En effet l'automatisation de la méthode implique l'analyse exhaustive d'un espace d'état très large et est donc restreinte à des parties réduites d'une description. De

ce fait, la validation basée sur la simulation reste encore la meilleure méthode pour la vérification de conception. Le lecteur intéressé peut se reporter aux différents articles publiés dans [IEEE-D&T 2001] qui présentent les dernières avancées en matière de vérification formelle.

Nous nous intéressons, pour notre part, aux approches de type validation basée sur la simulation. Afin de simuler une description on doit construire un programme supplémentaire appelé *test-bench* dont le principe fait l'objet de la section suivante.

2.3 Les test-bench

Un *test-bench* est un fichier de simulation écrit dans le même langage (Verilog ou VHDL) que la description HDL à simuler. Il applique un ensemble de transitions de valeurs d'entrée (appelé stimuli) à cette description et compare les résultats générés avec les réponses attendues. Les stimuli et les réponses attendues sur les sorties d'un circuit intégré sont appelés vecteurs de test. La Figure 2-3 montre les trois principaux éléments constitutifs d'un *test-bench* HDL :

- une Unité Sous Test (UST) représentant la description HDL à simuler ;
- un générateur de stimuli contrôlant l'UST ;
- un vérificateur testant et signalant automatiquement toute erreur rencontrée pendant la simulation. Il compare les réponses de l'UST avec les résultats attendus.

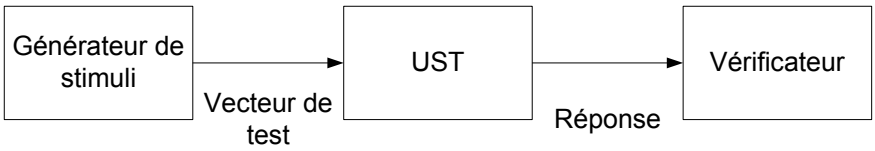


Figure 2-3. Un test-bench.

Pour vérifier et signaler les erreurs, le générateur de stimuli et le vérificateur utilisent des fichiers sous forme textuelle qui stockent les vecteurs de test. Les *test-bench* peuvent permettre de vérifier toute différence de comportement dans une description à différents niveaux du processus de conception. Le générateur de stimuli fournit les mêmes signaux d'entrée à chaque description testée. Ainsi, les réponses de toutes les descriptions sont générées simultanément. Le rôle du vérificateur est de grouper les résultats des simulations pour chaque description et de les comparer pour détecter d'éventuelles différences. Un autre avantage concernant l'utilisation d'un *test-bench* est sa portabilité par rapport au simulateur.

En effet, le *test-bench* étant écrit dans le même langage que l'UST, il demeure indépendant de l'outil de simulation.

La Figure 2-4 illustre la configuration utilisée si on désire vérifier qu'aucune erreur n'a été introduite entre le niveau algorithmique et le niveau RTL.

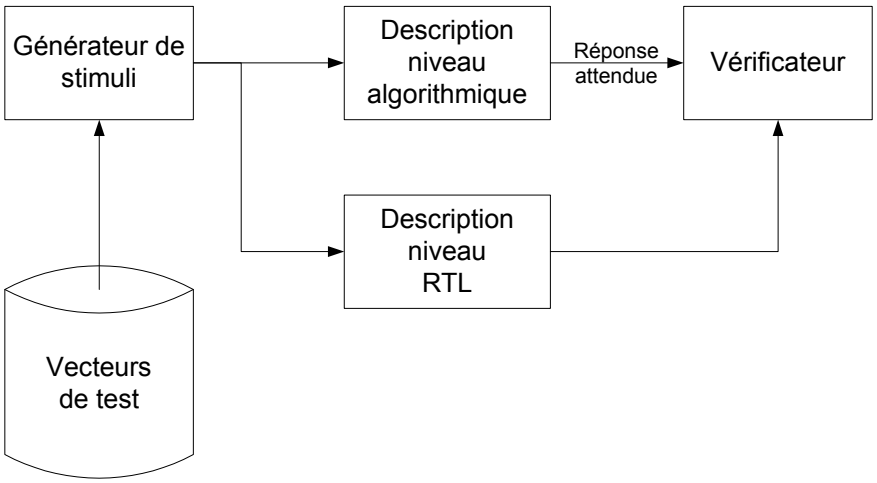


Figure 2-4. Configuration de test-bench.

Les vecteurs de test validant la description HDL au niveau algorithmique sont utilisés pour valider par simulation cette même description décrite à un niveau d'abstraction plus bas, c'est-à-dire au niveau RTL.

2.4 État de l'art

La production des vecteurs de test est un des principaux problèmes de la validation basée sur la simulation. Les inconvénients de cette approche sont qu'une simulation exhaustive est requise pour garantir l'exactitude de la description, et que la complexité des descriptions actuelles rend cette tâche infaisable. La génération aléatoire de vecteurs de test est relativement facile mais ne garantit pas la vérification de toutes les fonctionnalités de la description du circuit. En général, les concepteurs de circuits génèrent manuellement les vecteurs de test en se basant sur leur connaissance des fonctionnalités du circuit. Le problème de la génération automatique de vecteurs de test reste donc un sujet largement étudié par la communauté scientifique.

Nous présentons un état de l'art, qui se veut non exhaustif, de la génération automatique de vecteurs de test. Notre intention n'est pas de décrire en détail toutes les

méthodes existantes, mais simplement de montrer les plus représentatives, leurs limites, et les raisons qui nous ont amené à développer une nouvelle méthodologie.

Une approche pour la génération de vecteurs de test est l'utilisation de modèles de fautes comportementales [Ghosh 1991]. Le lecteur intéressé peut se référer à [Santucci 1993] et [Pla 1993]. Cho et Armstrong [Cho *et al.* 1991, 1994] ont proposé un algorithme qui permet de générer des vecteurs de test directement à partir des descriptions VHDL au niveau RTL. Dans cette approche, les fautes comportementales sont définies à partir de la perturbation des instructions VHDL de la description du circuit. La sensibilisation, la propagation et la justification de la faute sont effectuées en utilisant un graphe de flot de contrôle et de données (CDFG en anglais pour *Control Data Flow Graph*). Les vecteurs de test générés visent la détection des fautes de bas niveau, c'est-à-dire au niveau porte logique.

Dans [Ferrandi *et al.* 1998, 2000], les auteurs basent leur approche sur une description de type diagramme de décision binaire (BDD en anglais pour *Binary Decision Diagram*), des informations du flot de contrôle et de donnée de la description VHDL au niveau algorithmique et sur un modèle de faute simple. Il inclut en particulier : les défaillances de bit (chaque bit peut être « collé-à 0 » ou « collé-à 1 »), et les défaillances de condition (chaque condition peut être « collée-à vrai » ou « collée-à faux »). Les vecteurs de test sont générés en se basant sur la couverture de bit et visent la détection d'erreurs de conception.

L'ensemble des méthodes basées sur un modèle de faute [Ghosh 1991], [Pla 1993], [Cho *et al.* 1991, 1994], [Ferrandi *et al.* 1998, 2000] souffrent du fait qu'il n'est pas possible de mesurer le degré de confiance du modèle de faute. Contrairement aux défauts de fabrication (modèle de faute « collé-à »), il est difficile d'établir un modèle formel pour les erreurs de conception dues au fait qu'elles sont moins localisées et plus difficiles à caractériser. Une autre faiblesse de cette approche est qu'elle est basée sur la facilité de définition et d'usage des modèles de faute plutôt que sur leur réelle correspondance avec une erreur de conception. De plus, la facilité de mesure requiert une hypothèse de faute simple ou l'hypothèse que les fautes ne seront pas masquées plus tard lors de la simulation [Tasiran 2001].

L'utilisation des métriques de couverture dérivées du domaine du test de logiciels [Beizer 1990] constituent une autre approche pour la génération de vecteurs de test. Dans

[Fallah *et al.* 1998a, 1999], la génération des vecteurs de test repose sur une métrique de couverture d'instructions basée sur l'observabilité [Fallah *et al.* 1998b]. Des *tags* sont associés à chaque affectation de variable dans la description Verilog de niveau RTL et doivent être propagés, comme pour les fautes comportementales, vers une sortie pendant la simulation. Les vecteurs sont générés grâce à un algorithme hybride SAT (problème de satisfaction d'une expression booléenne).

La méthode développée par Kapoor et Armstrong [Kapoor *et al.* 1994] prend en compte des descriptions VHDL de niveau RTL et vise la détection d'erreurs de conception. Elle utilise le CDFG défini dans [Cho *et al.* 1994] pour générer des vecteurs de test à partir du critère de couverture d'instructions. Des valeurs symboliques sont utilisées pour représenter les valeurs des signaux et leurs transitions. Ces valeurs sont affectées par l'utilisateur pour justifier les chemins à travers le CDFG qui permettent de respecter le critère de couverture d'instructions. Un *test-bench* est finalement généré afin de simuler la description.

Dans [Kalynaraman 1993] et [Vemuri *et al.* 1995] des chemins d'exécutions pour lesquels on désire générer des vecteurs de test sont spécifiés à partir de certaines annotations, gérées par l'utilisateur, dans la description VHDL comportementale. Chaque chemin annoté est traduit sous forme de contraintes, un résolveur de contraintes est alors utilisé pour produire les vecteurs de test qui permettent leur exécution. Les vecteurs de test sont finalement convertis en format WAVES [IEEE1029.1 1992] pour faciliter la génération automatique de *test-bench*.

Les méthodes de [Shen *et al.* 1999] et de [Ho *et al.* 1995] utilisent des concepts empruntés à la vérification formelle pour extraire, à partir d'une description Verilog, des machines d'états finis (FSM en anglais pour *Finite State Machine*), à partir desquels les vecteurs de test sont générés. Alors que la méthode de [Shen *et al.* 1999] est basée sur la couverture de chemins, celle développée dans [Ho *et al.* 1995] est basée sur la couverture de transitions (changement d'état).

Les approches de [Rudnick 1998] et [Corno *et al.* 2000] prennent en compte des descriptions VHDL de niveau RTL. La première approche combine des techniques utilisées dans le domaine du test de logiciels : critère de couverture d'instructions à partir d'un graphe de transition d'état (STG en anglais pour *State Transition Graph*) ; et des techniques utilisées

au niveau porte logique : outil de synthèse RTL. Les vecteurs de test générés permettent de détecter des fautes de bas niveau, alors que l'approche développée dans [Corno *et al.* 2000] vise la détection d'erreurs de conception. Cette dernière utilise un algorithme génétique, qui par interaction avec un simulateur VHDL, génère automatiquement des vecteurs de test respectant le critère de couverture de branche.

L'ensemble des méthodes décrites précédemment peut faire l'objet des critiques et des commentaires suivants. Dans [Kalynaraman 1993] et [Vemuri *et al.* 1995], le concepteur doit annoter le nombre de fois exact ou maximal qu'une instruction de contrôle doit être exécutée. Ce qui implique une connaissance sur la fonctionnalité de la description et peut donner des chemins dont l'exécution est impossible. Cependant, un modèle de contrainte efficace qui permet de modéliser les instructions VHDL traversées par un chemin y a été défini. Nous avons utilisé ce modèle dans notre approche. Les méthodes de [Corno *et al.* 2000][Fallah *et al.* 1998a, 1999][Rudnick 1998] et [Cho *et al.* 1991, 1994], prennent en compte des descriptions HDL de niveau RTL qui est inférieur au niveau algorithmique. De plus l'approche de [Rudnick 1998] est dépendante d'un outil de synthèse. Néanmoins une excellente sémantique du langage VHDL est donnée dans [Cho *et al.* 1991, 1994]. Nous utilisons d'ailleurs le même formalisme pour la définition des instructions de type process qui sont des instructions propres à VHDL. Dans [Ho *et al.* 1995] et [Shen *et al.* 1999], les méthodes développées ne sont applicables que pour des conceptions relativement petites à cause de la taille de l'espace d'état de la FSM, malgré l'amélioration apportée dans [Shen *et al.* 1999].

2.5 Présentation de notre approche

L'avancement actuel des travaux de recherche dans le domaine de la validation de description HDL comportemental, présenté dans l'état de l'art précédent, met en évidence la nécessité du développement d'une nouvelle méthodologie capable :

i. de prendre en compte une description HDL de niveau algorithmique ;
ii. d'être indépendante des outils de synthèse et des niveaux d'abstraction inférieurs au niveau algorithmique ;
iii. de valider une description synthétisable par rapport au comportement attendu ;
iv. de générer automatiquement les vecteurs de test et le *test-bench* associé.

Afin de pouvoir répondre à ces attentes nous nous sommes tournés vers les techniques développées dans le domaine du test de logiciels. Ces techniques ont été utilisées avec succès sur des logiciels écrits avec des langages conventionnels tels que C ou ADA. Bien que l'idée d'appliquer des techniques appartenant au domaine du test de logiciels, sur des descriptions HDL comportementales, ne soit pas nouvelle [Ferrandi *et al.* 2000] [Fallah *et al.* 1998a] [Kapoor *et al.* 1994] [Kalynaraman 1993] [Vemuri *et al.* 1995] [Shen *et al.* 1999] [Ho *et al.* 1995] [Rudnick 1998] [Corno *et al.* 2000], ce travail est à notre connaissance, le premier qui intègre les quatre aspects précédents.

Dans le chapitre suivant, nous donnons une vue d'ensemble des techniques utilisées dans le domaine du test de logiciels afin de choisir une technique adaptée aux descriptions de circuits digitaux.

CHAPITRE

3

LE TEST DE LOGICIELS

Le but de ce chapitre est de donner une vue d'ensemble des techniques utilisées dans le domaine du test de logiciels et choisir une technique efficace pour les descriptions de circuits digitaux écrites en langage VHDL. La première section de ce chapitre présente le vocabulaire lié au test de logiciels utilisé dans ce document. La seconde partie dresse une liste non exhaustive des principales méthodes de test. Dans la troisième partie, on décrit en détail le critère de test que nous avons sélectionné pour les descriptions VHDL. La quatrième partie est dédiée au problème de la génération des données de test à partir d'un critère de test. Enfin, pour conclure ce chapitre nous dressons un bilan sur l'apport des techniques de test de logiciels concernant la validation de descriptions VHDL.

3.1 Introduction

Nous commençons cette section par quelques définitions issues de la terminologie définie par Laprie [Laprie 1992] et du glossaire standard IEEE [IEEE 1994] :

- une faute ou un défaut est la cause supposée d'une erreur dans le programme (par exemple une instruction erronée ou manquante) ;
- une erreur est la partie de l'état du logiciel susceptible d'entraîner une défaillance. Typiquement, une erreur est la prise d'une mauvaise valeur par une variable ou par une expression du programme ;
- une défaillance correspond à une déviation du service assuré par le programme de la fonction supposée de ce dernier ;

Par conséquent, une erreur est la cause d'une faute et résulte en une défaillance (voir figure ci-dessous).

Les erreurs se manifestent de différentes façons. Ces défaillances peuvent être dues à des erreurs de programmation, des erreurs de spécification ou des erreurs d'omission. Étant donnés les spécifications et le code correct de la Figure 3-1, nous pouvons voir comment chacune de ces erreurs peut se manifester.

```
Spécification: Donne la division de A par B.
      Programme:
      si(B ≠ 0)
      renvoyer(A/B)
```

Figure 3-1: Programme correct.

La Figure 3-2 montre qu'une erreur de programmation a été faite en divisant B par A au lieu A par B. Sur la Figure 3-3 on peut voir que le programme est correct mais que les spécifications données sont incorrectes. Enfin, la Figure 3-4 illustre le cas d'une erreur d'omission. C'est-à-dire que le programmeur a oublié de se soucier de la division par zéro.

```
Spécification: Donne la division de A par B.
Programme:
      si(B ≠ 0)
      renvoyer(B/A)
```

Figure 3-2: Erreur de programmation.

```
Spécification: Donne la multiplication de A par B.
Programme:
      renvoyer (A*B)
```

Figure 3-3: Erreur de spécification.

```
Spécification: Donne la division de A par B.
Programme:
      renvoyer (A/B)
```

Figure 3-4: Erreur d'omission.

L'erreur de programmation pourrait facilement être trouvée par n'importe quel ensemble de valeur tel que A ≠ B. Alors que l'erreur de spécification doit être détectée par une personne qui a une grande connaissance de la fonctionnalité du programme, l'erreur d'omission peut être détectée seulement si B = 0. Supposons que B est un nombre de 32 bits, les chances de détecter cette erreur sont faibles sans un peu de perspicacité. En raison de la nature variée des erreurs, il existe de nombreuses méthodes de validation de logiciel.

On définit la validation ou vérification de logiciel comme le processus d'évaluation du logiciel pour s'assurer qu'il satisfait aux exigences spécifiées. La validation ou vérification d'un produit cherche à s'assurer qu'on a construit le bon produit. Le test est un cas particulier de validation. C'est un processus manuel ou automatique, qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification, ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification [IEEE729 1983]. On désigne par spécification tout document qui prescrit les exigences auxquelles le produit doit se conformer. Le test vise à mettre en évidence les erreurs d'un logiciel et n'a pas pour objectif de diagnostiquer la cause des erreurs, ni de corriger les fautes, ni de prouver l'exactitude d'un programme.

La section suivante présente les principales techniques de test. À partir de toutes ces techniques, nous choisissons celle qui semble la plus appropriée pour les programmes VHDL et pour la génération automatique de vecteurs de test.

3.2 Classification des méthodes de test

Selon la phase du cycle de développement du logiciel pendant laquelle il est effectué, le test est qualifié d'*unitair*e, d'*intégration* ou de *systèm*e [Parissis 1996]. Le test unitaire

concerne chacun des modules[2] qui constituent le logiciel en le traitant isolément. Son but est la recherche d'erreurs dans la réalisation de ce module en ne considérant que ses spécifications. L'objectif du test d'intégration, au contraire, est la mise en évidence d'erreurs dans les appels entre modules ou dans la répartition des traitements entre modules pour réaliser les principales fonctions du logiciel. Le test de système, enfin, porte sur l'ensemble du logiciel et vise à identifier les cas où sa spécification initiale n'est pas respectée.

Quelle que soit l'étape du cycle de développement, l'activité de test se déroule généralement en deux temps :

- un ensemble de données d'entrée (souvent appelées *jeux de tests* ou *jeux d'essais*) est d'abord sélectionné ;
- le logiciel est ensuite exécuté avec ces données en entrée et son comportement est observé. Cette observation du comportement est souvent effectuée par un humain qui doit prononcer un « oracle » (i.e. il doit décider si les résultats de l'exécution sont corrects).

Notre étude se situe au niveau unitaire car nous considérons qu'un programme VHDL de type comportemental est un module, ce module pouvant faire partie d'un programme plus large. Cette hypothèse est clarifiée dans le prochain chapitre où les concepts du langage VHDL sont présentés. Concernant l'oracle, le langage VHDL prévoit l'utilisation de fichiers de simulation, appelés *test-bench* (cf. chapitre 2), qui permettent de vérifier et de signaler toute erreur rencontrée durant la simulation.

La classification générale des techniques utilisées pour le test unitaire est illustrée par la Figure 3-5. Elles sont divisées en deux familles : les méthodes basées sur l'analyse statique et les méthodes basées sur l'analyse dynamique.

[2] Un module (ou unité) est le plus petit élément d'un logiciel.

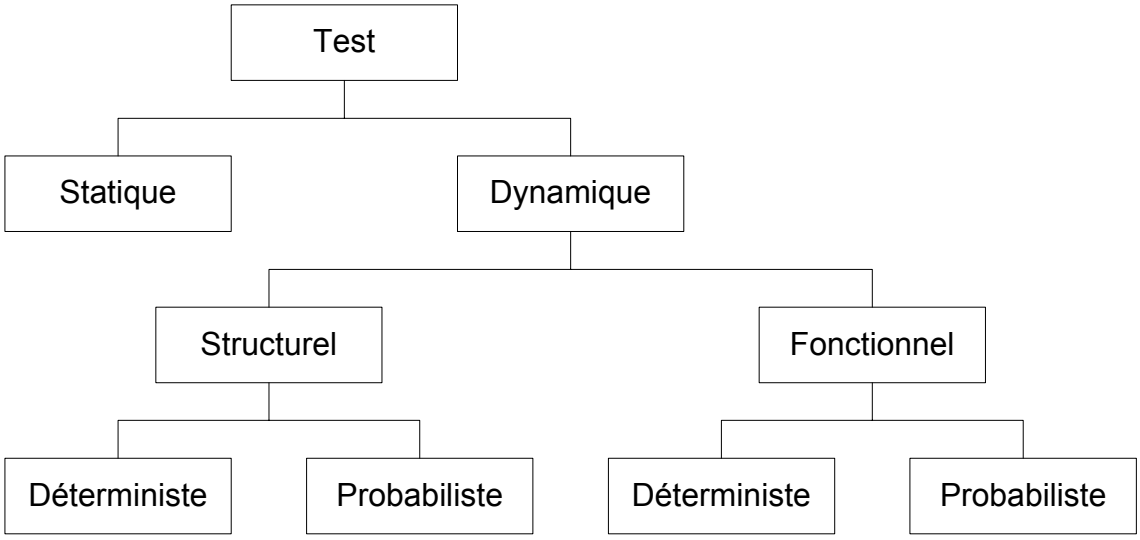


Figure 3-5. Classification des techniques de validation.

Les méthodes statiques consistent à examiner le code sans exécuter le programme tandis que les méthodes dynamiques consistent à examiner le comportement du programme pendant son exécution. Puisque le problème présenté dans cette thèse nécessite l'exécution du programme sous test avec des données de test concrètes, seules les méthodes dynamiques de cette classification nous intéressent. On distingue deux grandes catégories de techniques selon que l'on sélectionne les données de test à partir des spécifications ou à partir du programme : le test fonctionnel et le test structurel.

3.2.1 Le test fonctionnel

Le test fonctionnel [Parissis 1996], [Péraire 1998], [Beizer 1990,1995] n'utilise aucune information sur la structure du programme codé. Lorsque cette stratégie est utilisée le programme est vu comme une boîte noire. Le test fonctionnel vise à valider les fonctionnalités d'un programme. Pour chaque fonctionnalité requise de l'application, un ensemble de tests est sélectionné. Les jeux de tests sont dérivés des spécifications du programme sous test. Une spécification décrit complètement les comportements d'un système et peut être :

- informelle (spécification en langage naturel), dans ce cas un ensemble de jeux de tests est sélectionné manuellement pour chaque fonctionnalité décrite ;
- formelle (spécification algébrique), dans ce cas une sélection automatique de jeux de tests est envisageable.

L'automatisation du processus de génération des jeux de tests n'est pas pratique, parce qu'elle exige des caractéristiques très formelles pour les spécifications. Ce type de formalité

ferait que les spécifications ressembleraient au code du programme, ce qui est censé être le produit final, et non pas le début du processus de conception. La génération de jeux de tests exige donc une connaissance des fonctionnalités du programme et une idée sur les défaillances possibles du logiciel.

3.2.2 Le test structurel

Le test structurel [Parissis 1996], [Péraire 1998], [Beizer 1990,1995] utilise la structure de commande du programme comme base pour développer ou évaluer des tests. Lorsque cette stratégie est utilisée le programme est vu comme une boîte blanche.

Il existe des outils de test boîte blanche qui se basent sur l'analyse du code pour fournir des graphes. Ils permettent d'aiguiller les développeurs vers les tests à réaliser pour obtenir la couverture d'éléments définis sur ces graphes. De telles stratégies de test incluent le test d'instruction, le test de branche, et le test de chemin. On peut automatiquement produire des statistiques qui donneront les pourcentages de la structure couverte. Mais plus significativement, c'est la capacité d'automatiser le processus de génération de test qui rend le test structurel plus attrayant. Le test structurel a cependant un certain nombre d'inconvénients. Premièrement, la sélection d'un jeu de tests de taille raisonnable couvrant tous les chemins exécutables n'est pas toujours possible lors de la présence de boucles (il faut limiter le nombre de passages dans ces boucles). En second lieu, ces méthodes ne permettent pas de détecter des oublis par rapport à la spécification de l'application, cette dernière n'intervenant pas dans le processus de sélection des jeux de tests.

Une autre technique appartenant au test de boîte blanche est le test par mutation. Suivant cette méthode, des défauts simples sont introduits au logiciel par une opération appelée « mutation » qui consiste à remplacer une constante ou une variable par une autre de même type ou bien à modifier un opérateur arithmétique ou relationnel (par exemple + par - ou < par <=). Chaque défaut ainsi introduit donne naissance à un nouveau programme, appelé « mutant ». Après avoir ainsi constitué un ensemble conséquent de mutants, ces derniers sont exécutés avec le jeu de test dont on veut évaluer l'efficacité et leurs résultats sont comparés à ceux du programme original. Idéalement, le jeu de test devrait provoquer la production de résultats différents par les mutants (on dit alors que les mutants sont "tués"). Le programme de la Figure 3-6 représente une mutation du programme donné sur la Figure 3-1. La condition ($B \neq 0$) qui a été changée en ($B = 0$) représente une erreur de programmation possible.

```
Spécification: Donne la division de A par B.
Programme:
      si(B = 0)
      renvoyer (A/B)
```

Figure 3-6: Programme mutant.

L'avantage de la technique des mutants est qu'elle est très facilement automatisable, aussi bien en ce qui concerne la production des mutants que leur exécution.

3.2.3 Le test structurel basé sur un critère de couverture

Étant données les diverses méthodes de test de logiciel, le test structurel semble le plus approprié pour des descriptions VHDL comportementales. En effet, l'utilisation du code VHDL comportemental exige l'utilisation d'une méthode de test de boîte blanche. Nous avons cependant distingué le test par mutation, plus complexe, des techniques de test structurel basées sur un critère de couverture. Les mutants doivent être créés à partir du programme initial et leur nombre, même dans un petit programme, peut être très grand. De ce fait la création, la gestion, et l'exécution des mutants peuvent se révéler consommatrices en temps. Et puisque le test par mutation exige une analyse structurelle pour créer les mutants, le test structurel basé sur un critère de couverture semble le point de départ le plus approprié.

En utilisant cette stratégie, le testeur sélectionne les données de test par l'examen de la logique du programme. Cette approche est basée sur une représentation graphique du programme appelée graphe flot de contrôle (cf. section 3.3.1). Une multitude de critères de sélection définis sur ce graphe ont été proposés. Les principales méthodes sont la couverture d'instructions, la couverture de branches et la couverture de chemins.

La couverture d'instructions exige l'exécution de chaque instruction dans le programme au moins une fois pendant le test. La couverture de branches exige que chaque branche du programme soit traversée au moins une fois. Une branche est un point dans le programme à partir duquel un ou plusieurs ensembles d'alternatives d'instructions de programme sont sélectionnés. La couverture de chemins est la méthode la plus rigoureuse et la plus efficace et consiste à parcourir un ensemble de chemins qui respecte un certain critère.

Ces méthodes peuvent être classées au moyen de la relation d'inclusion : un critère *X* est inclus dans le critère *Y* si tout jeu de test satisfaisant *Y* satisfait également X. On obtient

que la couverture d'instructions est incluse dans la couverture de branches, elle même incluse dans la couverture de chemins. Or, la couverture de 100% des chemins est théoriquement impossible à atteindre. En présence de boucles, le nombre de chemins d'un graphe de contrôle est potentiellement infini. Pour cette raison, plusieurs autres critères ont été définis dans le but de combler le vide entre la couverture des branches et celle des chemins.

Dans la section suivante, nous présentons un critère de test performant basé sur la couverture de chemins que nous désirons appliquer aux programmes VHDL.

3.3 Les techniques de McCabe : Le test structuré

McCabe a développé depuis 20 ans des solutions dédiées à la qualité, la maintenance et le test de logiciels fondées sur l'analyse structurelle du code. Ces techniques, appartenant à la famille du test structurel, font office de référence en matière de qualimétrie et de certification des tests. Plus de 25 milliards de lignes de code ont été analysées avec les outils McCabe. Ces techniques s'appliquent aux langages suivants : Cobol, C, C++, Java, VB, Ada et Fortran, indépendamment des plates-formes.

De nombreuses métriques et techniques reconnues depuis comme des standards de l'industrie ont été publiées. Tous ces travaux ont été standardisés par le NIST (*National Institute of Standards and Technology*). Dans cette sous-section, nous présentons en détails :

- la complexité cyclomatique notée v(G) ;
- le test cyclomatique, dit aussi test structuré ou test des chemins de base ;

qui sont basés sur la structure du code sous test. L'analyse de ce code nécessite une représentation graphique appelée Graphe Flot de Contrôle (GFC). Nous commencerons naturellement cette section par la définition de ce graphe.

3.3.1 Le graphe flot de contrôle

Le flot de contrôle représente le séquencement d'exécution des instructions d'un programme. Les Graphes Flot de Contrôle (GFC) fournissent une représentation graphique de la structure de commande dans un module. La définition d'un module est dépendante du langage ; en général c'est une unité de code avec un point d'entrée et un point de sortie. Chaque GFC est constitué de nœuds et d'arcs. Les nœuds symbolisent des déclarations ou des instructions ; ils peuvent représenter plus d'une ligne de code. Les arcs symbolisent le

transfert de commande entre les nœuds. Cette description graphique aide à la compréhension d'algorithmes complexes.

Formellement, on peut définir un GFC comme :

- un graphe orienté : $G=(N,E)$;
- l'ensemble N des nœuds représente les instructions du programme ;
- l'ensemble E des arcs orientés (s,s') représentent le passage du contrôle de l'instruction s, source de l'arc, à l'instruction s', destination de l'arc. Un arc sera appelé branche si s est une condition ;
- chaque nœud est caractérisé par :
    - le degré interne : le nombre d'arcs aboutissant au nœud ou le nombre de prédécesseurs ;
    - le degré externe : le nombre d'arcs sortant du nœud ou le nombre de successeurs ;
- un chemin $(a_1,a_2,\ldots,a_n)$ est une suite d'arcs telle que la destination de a_i est la source de a_{i+1}.

La Figure 3-7 illustre les différents types de nœuds pouvant composer un GFC :

- le nœud de début possède un degré interne égal à 0 : ce nœud représente le point d'entrée du programme ;
- le nœud simple possède un degré interne et externe égal à 1 : ces nœuds représentent les exécutions d'instructions arithmétiques ou logiques qui n'affectent pas le flot de contrôle du programme (par exemple les instructions d'affectation). Ils possèdent seulement un arc entrant et un arc sortant ;
- le nœud sélectif ou nœud de décision : une décision est un point du programme où le flot de commande peut diverger. Ces nœuds ont un arc entrant et plusieurs arcs sortants dépendant du type de la décision (IF, CASE...) ;
- le nœud de jonction : une jonction est un point dans le programme où le flot de commande peut converger. Par exemple une jonction peut être un END IF, ou un END CASE ;
- le nœud de fin possède un degré externe égal à 0 : ce nœud représente le point de terminaison du programme.

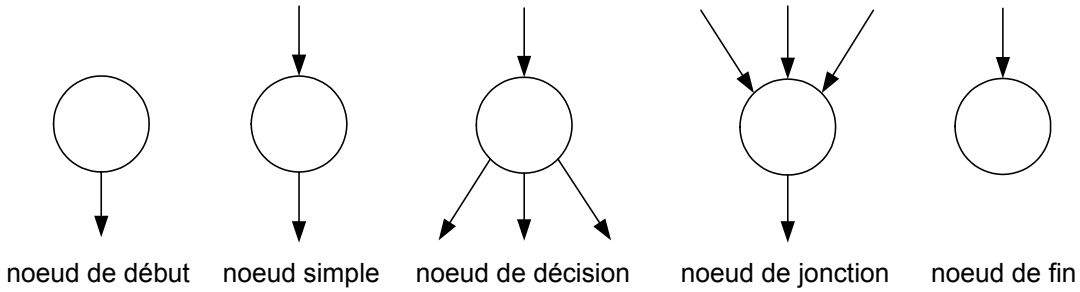


Figure 3-7. Les différents types de nœud d'un GFC.

Dans ce contexte, un chemin est une suite d'arcs représentant une séquence d'instructions réellement parcourues dans un programme lors d'une de ses exécutions, parmi la quantité de branchements divers qu'il contient.

On appellera *chemin d'exécution* (ou *chemin complet*), un chemin pour lequel il existe un ensemble de valeurs pour les variables d'entrée qui permet son exécution. Une *variable d'entrée* est une variable du programme apparaissant dans une instruction de lecture. Par opposition, on définit un *chemin infaisable* comme un chemin qui ne pourra être exercé par aucune valeur possible. Un chemin infaisable est un chemin qui traverse du code mort c'est-à-dire du code qui ne peut être exécuté ou qui est devenu inutile dans un programme. La présence de code mort révèle une erreur logique dans un programme. En fait, le programme a été mal conçu et peut être simplifié de sorte qu'il n'y ait plus de chemin infaisable.

Un chemin d'exécution d'un module de logiciel peut toujours être représenté par un chemin à travers son GFC qui commence au point d'entrée du programme (nœud de début) et qui se termine au point de sortie du programme (nœud de fin). Cette correspondance est une définition de base pour les techniques de McCabe.

3.3.2 La complexité cyclomatique : v(G)

Les métriques de logiciel décrivent les propriétés du code du logiciel. Leur objectif est d'évaluer si les exigences de qualité sont satisfaites durant toutes les phases du cycle de vie du logiciel. Les métriques sont des mesures qui permettent d'évaluer la complexité d'un logiciel offrant une aide pour le test.

La complexité cyclomatique est une métrique de logiciel développée par McCabe [McCabe 1976]. La complexité cyclomatique est aussi appelée v(G) où v fait référence au

nombre cyclomatique dans la théorie des graphes [Berge 1973] et où G indique que la complexité est dépendante du graphe.

La méthode de calcul est la suivante. Soit un module donné dont le Graphe Flot de Contrôle G contient e arcs et n nœuds, alors v(G) peut être calculée avec la formule :

$$v(G) = e - n + 2$$

Cette complexité correspond au nombre de chemins indépendants à travers un graphe orienté tel que le GFC. Une certaine familiarité avec l'algèbre linéaire est recommandée pour suivre les explications suivantes, mais le point important est que v(G) est précisément le nombre minimum de chemins qui peut, par combinaison linéaire, générer tous les chemins possibles à travers un module.

On considère qu'un ensemble de plusieurs chemins donne une matrice dans laquelle les lignes sont les chemins et les valeurs des colonnes sont le nombre de fois que les arcs sont traversés par ces chemins. D'après l'algèbre linéaire, chaque matrice possède un rang unique (nombre de lignes linéairement indépendantes) qui est inférieur ou égal au nombre de colonnes. Cela signifie que peu importe le nombre de chemins possibles qui lui est ajouté, son rang ne pourra jamais dépasser le nombre d'arcs du GFC. En fait, la valeur maximum de ce rang est égale exactement à v(G). Un ensemble minimal de vecteurs indépendants (ou chemins) avec un rang maximum est appelé base. Une base peut aussi être décrite comme un ensemble de vecteurs linéairement indépendants qui génèrent tous les vecteurs dans l'espace par combinaison linéaire. Cela signifie que v(G) est le nombre de chemins dans tout ensemble de chemins indépendants qui génèrent tous les chemins possibles par combinaison linéaire. Pour tout ensemble de chemins, il est possible de déterminer le rang en appliquant la méthode d'élimination de Gauss sur la matrice associée. Le rang est le nombre de ligne non égale à zéro une fois l'élimination accomplie. Si aucune ligne n'est égale à zéro pendant l'élimination, les chemins sont linéairement indépendants. Si le rang est égal à v(G), l'ensemble d'origine des chemins génère tous les chemins par combinaison linéaire. Si les deux conditions sont exactes, l'ensemble d'origine des chemins forme une base pour le GFC.

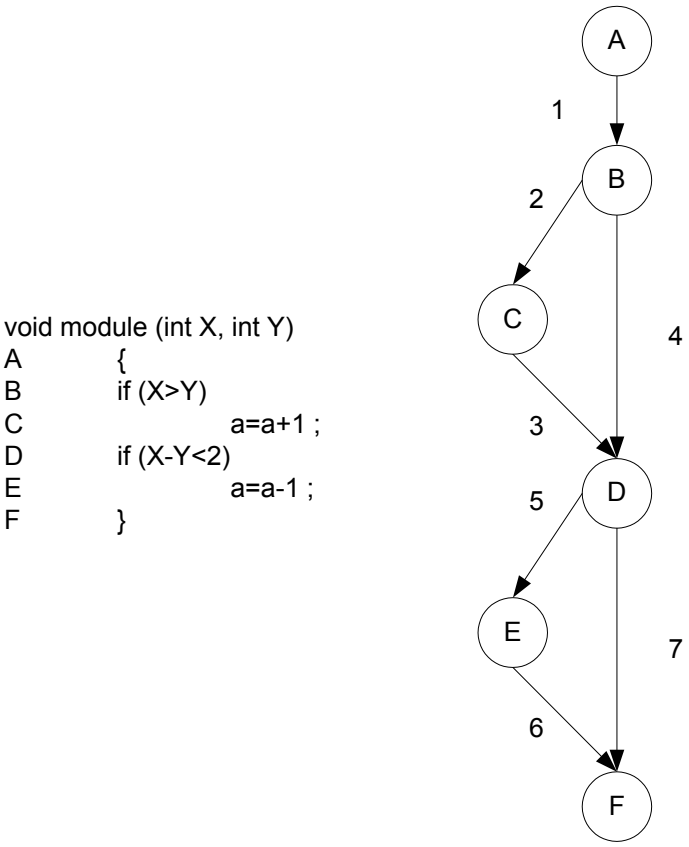


Figure 3-8. Un programme et son GFC.

Soit le programme la Figure 3-8 et son GFC qui possède 7 arcs numérotés de 1 à 7 et 6 nœuds notés de A à F ; sa complexité cyclomatique est égale à 3 (7-6+2). Le chemin {1,2,3,5,6} peut être représenté par le vecteur [1 1 1 0 1 1 0]. Les chemins sont combinés en ajoutant ou en soustrayant les représentations des vecteurs de chemin. Un ensemble de base pour ce CFG est ({1,2,3,5,6}, {1,2,3,7}, {1,4,5,6}). Le chemin {1,4,7} peut être construit par la combinaison {1,4,5,6} + {1,2,3,7} – {1,2,3,5,6} comme représenté sur le Tableau 3-1 (chemin 4).

Chemins\Arcs	1	2	3	4	5	6	7
Chemin 1	**1**	**1**	**1**	**0**	**1**	**1**	**0**
Chemin 2	**1**	**1**	**1**	**0**	**0**	**0**	**1**
Chemin 3	**1**	**0**	**0**	**1**	**1**	**1**	**0**
Chemin 4	**1**	**0**	**0**	**1**	**0**	**0**	**1**

Tableau 3-1. Matrice de construction des chemins.

L'ensemble ({1,2,3,5,6}, {1,4,7}) n'est pas un ensemble de base, parce qu'il n'y a aucune façon de construire le chemin {1,2,3,7}. L'ensemble ({1,2,3,5,6}, {1,2,3,7}, {1,2,4,7}) est également un ensemble de base.

Les ensembles de base ne sont pas uniques, ainsi un GFC peut avoir plus d'un ensemble de base. Bien sûr chaque base possède le même nombre v(G) de chemins.

3.3.3 Le critère du test structuré

Le test structuré [McCabe 1982] [Watson 1996a] est une méthodologie basée sur le concept de la complexité cyclomatique. Il est théoriquement plus rigoureux et plus efficace que les autres critères de couverture comme la couverture d'instructions ou la couverture de branches [Watson 1996b]. La Figure 3-9 illustre la relation d'inclusion entre les critères de couverture présentés dans la section 3.2.3 et le test structuré.

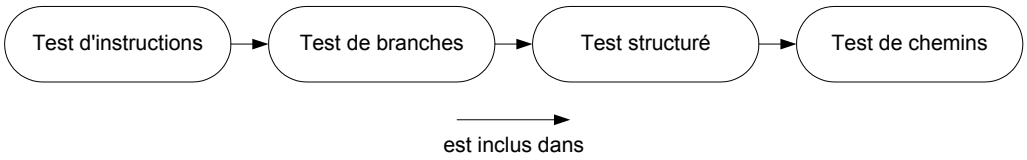


Figure 3-9. Relation d'inclusion entre les critères.

La philosophie générale est d'accepter v(G) comme un indicateur du nombre de chemins à tester dans un programme. Le critère de test structuré peut s'énoncer ainsi : « Tester une base de chemins à travers le graphe de flot de commande de chaque module ». Cela signifie que tout chemin supplémentaire à travers le graphe de flot de contrôle du module peut être exprimé comme une combinaison linéaire de chemins déjà testés. Ce critère établit un nombre, v(G), de chemins test qui a les propriétés suivantes :

- un ensemble de v(G) chemins indépendants peut être réalisé ;
- tester plus que v(G) chemins indépendants est redondant.

Le nombre minimum de tests demandé pour satisfaire le critère de test structuré est exactement la complexité cyclomatique : v(G). Le critère de test structuré mesure la qualité du test, fournissant un moyen pour déterminer si le test est complet, il n'a pas la fonction d'identifier les données de test. Ce problème sera abordé dans le paragraphe 3.4.

Quelque fois, il n'est pas possible de tester complètement un ensemble de base de chemins à travers le GFC d'un module. Cela se produit lorsqu'un chemin de la base est infaisable. Par exemple, si un module contient deux décisions identiques à la suite l'une de l'autre, aucune donnée d'entrée ne pourra faire varier la sortie de la première décision sans faire varier la sortie de la seconde et vis versa. Le chemin {1,2,3,7} de la Figure 3-10 est un exemple de chemin infaisable.

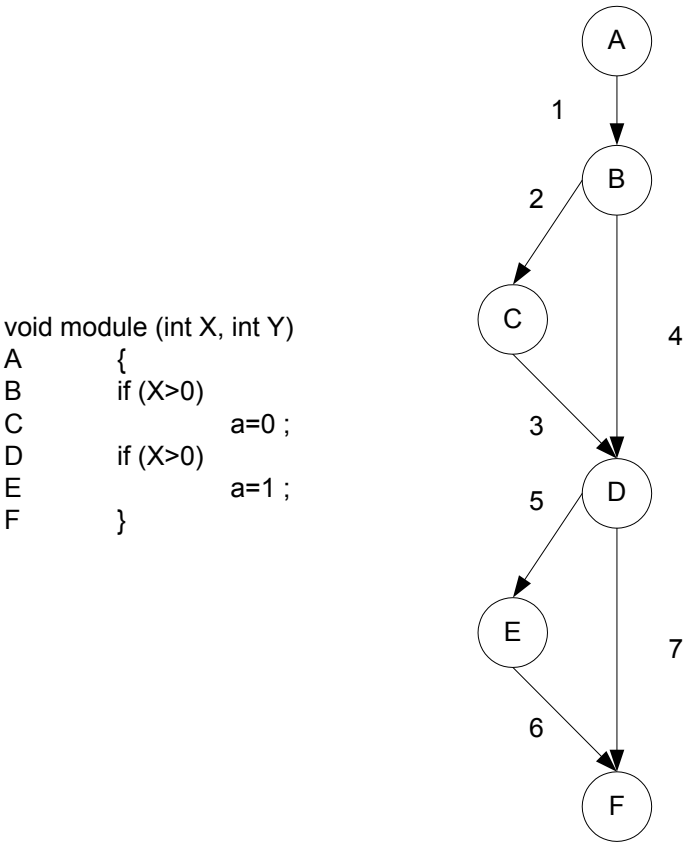


Figure 3-10. Un programme et son GFC avec deux décisions identiques.

Le critère de test structuré a fait l'objet de critiques dans [Evangelist 1984]. Les problèmes suivants ont été soulevés :

1. Comment générer une base de chemins indépendants ?
2. La génération des données de test à partir des v(G) chemins est un problème indécidable.

Nous présentons dans la section suivante un algorithme puissant : l'algorithme de Poole, permettant de générer une base de chemins à partir d'un GFC et ainsi de répondre à la première critique ; et dans le paragraphe 3.4 nous présentons une méthodologie capable d'extraire des données de test à partir d'un chemin.

3.3.4 L'algorithme de Poole

L'algorithme défini par Poole [Poole 1995] génère à partir du GFC d'un module une base de chemins. C'est un algorithme de recherche « en profondeur d'abord » (les nœuds sont numérotés dans l'ordre de leur exploration). La recherche commence au nœud de début et traverse récursivement tous les nœuds possibles du GFC. Si le nœud n'a jamais été visité, un arc est sélectionné par défaut en sortie, alors le chemin courant est divisé en nouveaux

chemins qui traversent chaque arc sortant, traversant l'arc de défaut d'abord. L'arc de défaut peut être n'importe quel arc sauf un arc de retour (arc qui, pour un chemin donné, mène à un nœud qui a été déjà visité) ou tout arc qui plus tard causera au nœud d'avoir deux arcs entrants. Par exemple, dans la condition de test d'une boucle, l'arc de défaut sera l'arc de sortie de la boucle. Si l'arc qui traverse le corps de la boucle était choisi, alors l'arc retour du dernier nœud dans le corps de la boucle vers le nœud de la condition de test devrait être traversé plus tard. Si le nœud visité est un nœud de fin (aucun arc de sortie), alors un chemin dans l'ensemble de base a été trouvé. Autrement, le chemin continue en traversant l'arc de défaut. L'algorithme de Poole en pseudo-code est donné sur la Figure 3-11.

```
TrouverBase (noeud)
si ce nœud est le nœud fin alors
      écrire ce chemin comme une solution
sinon
      si nœud n'a pas été visité avant
            marquer le nœud comme visité
            désigner un arc de défaut
            TrouverBase (destination de l'arc de défaut)
            pour tous les autres arcs sortants
                  TrouverBase (destination de l'arc)
      sinon
            TrouverBase (destination de l'arc de défaut)
```

Figure 3-11. Algorithme de Poole.

Sur la Figure 3-12 nous reproduisons le code et le GFC de la section 3.3.2 pour exposer cette méthode. L'algorithme commence au nœud de début : A dont l'arc de défaut est l'arc 1. Le nœud A est traversé et le nœud B est visité. L'arc de défaut de B peut être l'arc 2 ou 4, sélectionnons l'arc 2. Le nœud B est traversé et les nœuds C puis D sont visités. L'arc 5 est sélectionné comme arc de défaut pour le nœud D. Le nœud E est traversé et le nœud de fin : F est visité. Il n'y a aucun arc sortant, ainsi le chemin {1,2,3,5,6} est ajouté à l'ensemble de base. L'algorithme a traversé l'arc de défaut du nœud D, donc maintenant l'arc 7 est traversé. La destination est encore un nœud de fin, ainsi le chemin {1,2,3,7} est ajouté à l'ensemble de base. Tous les arcs sortants du nœud D ont été traversés, donc l'arc 4 du nœud B est traversé. La destination de l'arc 4 est le nœud D qui a déjà été visité. Par conséquent l'arc de défaut 5 menant au nœud E est traversé, ainsi que l'arc de défaut 6 car le nœud E a lui aussi été déjà visité. La destination est encore le nœud de fin : F, ainsi le chemin {1,4,5,6} est ajouté à l'ensemble de base. Tous les arcs sortants du nœud B ont été traversés, l'algorithme se termine.

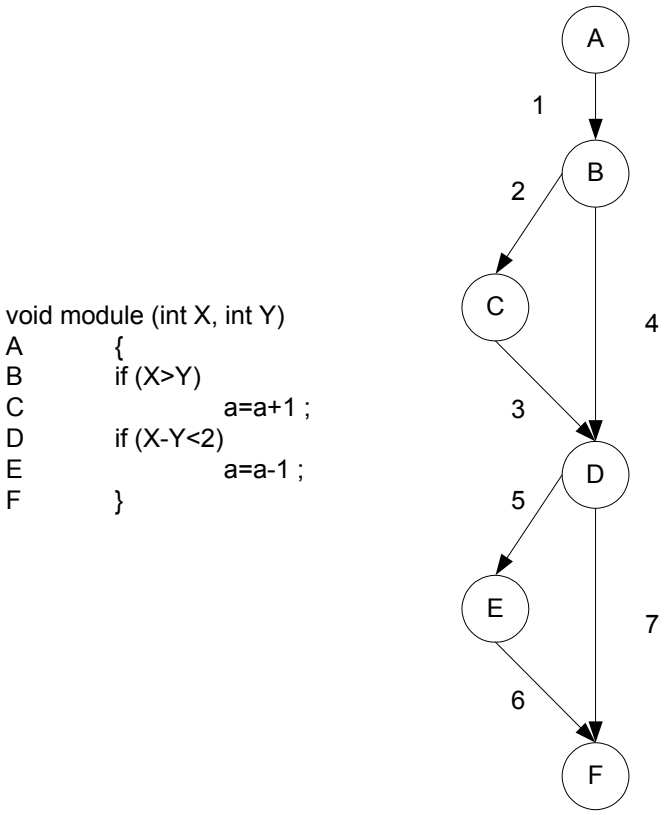


Figure 3-12. Application de l'algorithme de Poole.

Une trace du programme exécuté sur cet exemple est présentée sur la Figure 3-13.

```
TrouverBase(A)
        Arc1 :arc de défaut de A
        TrouverBase(destination de l'arc 1) : TrouverBase(B)
        Arc2 :arc de défaut de B
        TrouverBase(destination de l'arc 2) : TrouverBase(C)
        Arc3 :arc de défaut de C
        TrouverBase(destination de l'arc 3) : TrouverBase(D)
        Arc5 :arc de défaut de D
        TrouverBase(destination de l'arc 5) : TrouverBase(E)
        Arc6 :arc de défaut de E
        TrouverBase(destination de l'arc 6) : TrouverBase(F)
                Ecrire le chemin {1,2,3,5,6}
                TrouverBase(destination de l'arc 7) : TrouverBase(F)
                        Ecrire le chemin {1,2,3,7}
                        TrouverBase(destination de l'arc 4) : TrouverBase(D)
                        TrouverBase(destination arc de défaut de D) : TrouverBase(destination de
                        l'arc 5) : TrouverBase(E)
                        TrouverBase(destination arc de défaut de E) : TrouverBase(destination de
                        l'arc 6) : TrouverBase(F)
                                Ecrire le chemin {1,4,5,6}
```

Figure 3-13. Trace de l'algorithme de Poole.

L'algorithme de Poole répond donc à la première critique formulée par Evangelist [Evangelist 1984]. Cependant la connaissance de l'ensemble des chemins respectant le critère du test structuré ne donne pas une indication sur les valeurs d'entrée du programme à appliquer pour les traverser. La section suivante est consacrée à ce problème.

3.4 La génération automatique des données de test

L'automatisation de la génération de données de test est un des problèmes le plus difficile à résoudre et un des plus coûteux concernant le test de logiciels. On définit cette tâche, qui est traditionnellement réalisée à la main, comme le processus de création des données de test à appliquer aux entrées du programme qui satisfont un certain critère de test. Le problème général est indécidable, ainsi le monde de la recherche s'est intéressé à des heuristiques et à des méthodes donnant des solutions partielles. Les générateurs de données de test peuvent être divisés en trois groupes :

- les générateurs orientés structure qui essaient de couvrir certains éléments structurels du programme ;
- les générateurs basés sur une description formelle du domaine d'entrée du programme ;
- les générateurs aléatoires qui créent des données de test sans satisfaire a priori aucun critère de test.

Étant donné que la méthodologie choisie est basée sur un critère structurel, à savoir la couverture de v(G) chemins indépendants, on s'intéressera uniquement aux générateurs orientés structure.

Typiquement les générateurs orientés structure utilisent une représentation abstraite du programme (comme un GFC) et une certaine forme d'évaluation symbolique. L'évaluation symbolique exécute un programme en utilisant des valeurs symboliques pour les variables plutôt que des valeurs réelles. L'évaluation symbolique génère une condition de chemin, aussi appelée contrainte de chemin.

Une contrainte est une expression algébrique qui restreint le domaine des variables du programme. Ainsi les chemins peuvent être représentés par des systèmes de contraintes ; une contrainte pour chaque prédicat traversé par le chemin. Un prédicat est une expression booléenne associée à un nœud de décision qui détermine quel arc sera traversé à partir du nœud. Les prédicats sont d'abord exprimés en terme de variables de programme ; puis comme chacune de ces variables de programme peut finalement être exprimée en terme de variables d'entrée (en utilisant les instructions d'affectation le long du chemin), il est possible d'exprimer à nouveau les prédicats comme des contraintes en terme de variables d'entrée

uniquement. Si les données d'entrée qui satisfont la contrainte de chemin existent, le chemin est un chemin d'exécution et peut être utilisé pour tester le programme. Si la contrainte de chemin ne peut être satisfaite, le chemin est un chemin infaisable.

Si le chemin est faisable l'ensemble des données d'entrée forme alors un jeu d'essais qui est utilisé pour évaluer le logiciel sous test. La génération de données de test est définie alors comme le processus d'identification d'un jeu d'essais qui satisfait un critère de test.

DeMillo a présenté dans [DeMillo *et al.* 1991] une approche pour la génération de données de test qui utilise l'analyse du GFC, l'évaluation symbolique et un critère de couverture de mutants. Cette approche appelée *test basé sur les contraintes* (CBT en anglais pour Constraint-Based Testing) utilise une technique de satisfaction de contraintes appelée *réduction de domaine*. L'approche CBT rencontre de sérieuses difficultés lors de la manipulation de tableaux, d'expressions complexes et de boucles. Afin de résoudre ces problèmes, Offutt a présenté dans [Offutt *et al.* 1997] une nouvelle approche nommée *procédure de réduction de domaine dynamique* (DDR en anglais pour Dynamic Domain Reduction). Cette méthode combine l'approche CBT, l'approche de génération de donnée de test dynamique de Korel [Korel 1990] qui considère l'exécution d'un chemin spécifique, et l'évaluation symbolique.

Ce processus peut être présenté comme suit où le domaine de la variable est défini par l'ensemble de ses valeurs possibles :

Soit n_f le nœud à atteindre dans le GFC d'un programme P ayant un domaine d'entrée D. Le problème de génération de données de test est : trouver une entrée de programme $t \in D$ tel que quand P est exécuté avec t, n_f est atteint. Pour un chemin $c = <n_1, n_2, .., n_f>$, t doit causer l'exécution du chemin. Par commodité, on reproduit l'exemple très simple du module de la section 3.3.2 sur la Figure 3-14.

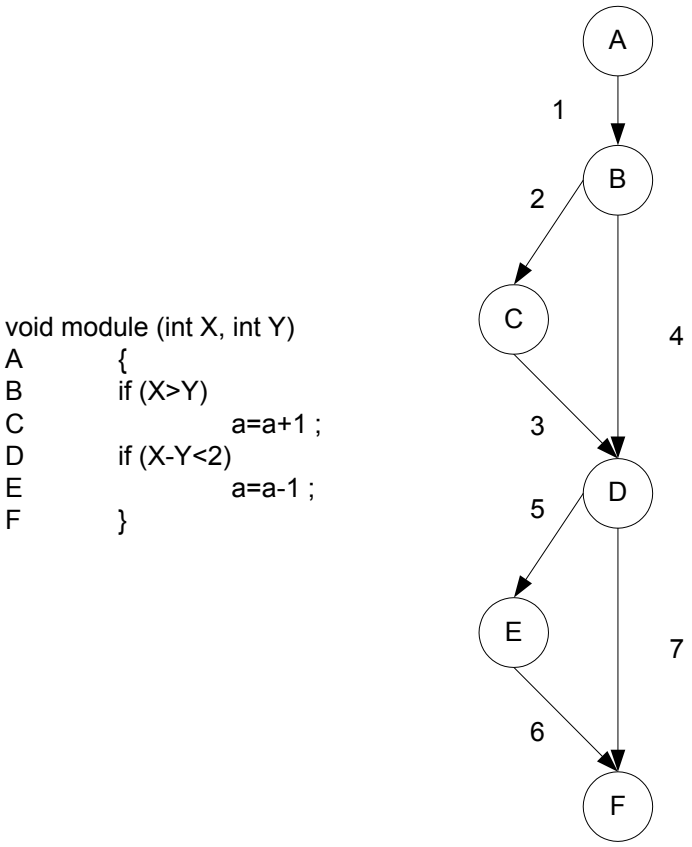


Figure 3-14. Un programme et son GFC.

On suppose que ce module a pour fonction de laisser la valeur a inchangée sous n'importe quelle condition, ce qui est donc clairement incorrect. Comme v(G) est égal à 3 (7 arcs moins 6 nœuds plus 2), on doit générer trois chemins test indépendants pour valider ce module. L'algorithme de Poole présenté précédemment donne :

Chemin 1 : {1,2,3,5,6},

Chemin 2 : {1,2,3,7},

Chemin 3 : {1,4,5,6}.

C'est le second chemin qui permet de découvrir l'erreur dans le module. En effet le chemin 1 et le chemin 3 laissent la valeur a inchangée, alors que le chemin 2 augmente la valeur a de 1. Donc le module ne réalise pas la tâche demandée. Puisque c'est le chemin 2 qui détecte l'erreur, prenons ce chemin comme exemple pour illustrer l'extraction des données de test qui permettront de traverser ce chemin.

Pour simplifier un peu plus le problème on suppose que les domaines de départ des variables d'entrée sont :

$X \in [0..5]$

$Y \in [0..5]$

Soit *F* le nœud cible, deux prédicats sont traversés et sont représentés par les couples d'arcs : 1-2 et 3-7. Le premier prédicat sur le couple d'arc 1-2 : X > Y doit être vrai. Il est utilisé pour réduire les domaines de X et Y.

Offutt [Offutt *et al.* 1997] propose alors de couper les domaines de X et Y en un point appelé point de coupure (*split point* en anglais). Initialement, ce point est choisi de sorte que chaque domaine « perde » approximativement le même nombre de valeurs. Les valeurs des domaines réduits peuvent ne pas satisfaire les prédicats traversés par le chemin. L'approche proposée par Offutt prévoit alors de modifier le choix du point de coupure.

Dans notre exemple, si on choisit X=3 comme point de coupure, les domaines des variables d'entrée deviennent :

$X \in [3..5]$

$Y \in [0..2]$

Ces domaines représentent une partie du domaine d'entrée que peut prendre une branche. On remarque qu'une partie du domaine d'entrée solution est éliminée. La Figure 3-15 montre les valeurs possibles pour les domaines originaux de X et Y et la contrainte X > Y à gauche (triangle blanc hachuré) et la région du domaine avec le point de coupure X=3 à droite (carré noir hachuré).

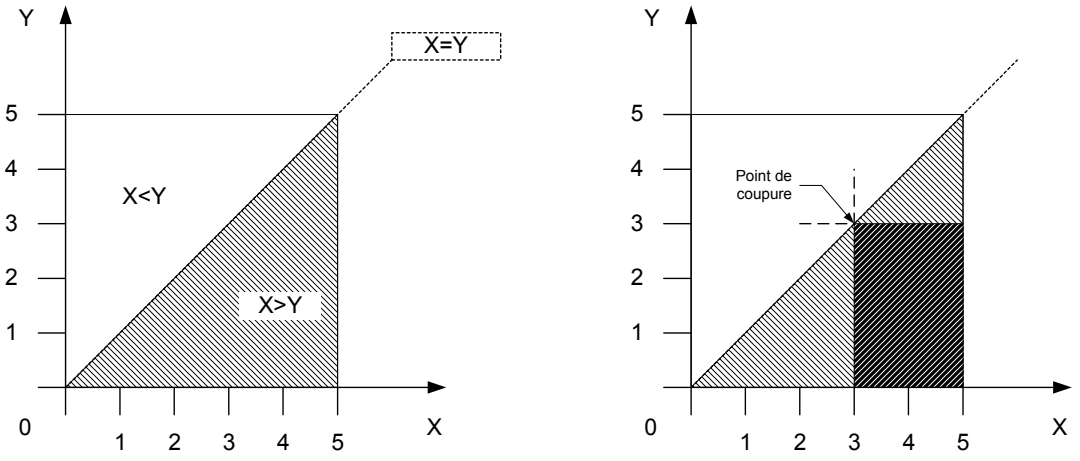


Figure 3-15. Vue du domaine avec contraintes et un point de coupure.

Après avoir traversé les arcs 1-2, on doit traverser le prédicat : X-Y < 2 qui doit être faux. Le point de coupure X=4 est utilisé pour réduire de nouveau les domaines de X et Y qui deviennent :

$X \in [3..4]$

$Y \in [0..1]$

Le Tableau 3-2 récapitule les domaines de chaque variable d'entrée après leur modification par les contraintes.

	X	Y
1 :début	[0..5]	[0..5]
2 :X > Y	[3..5]	[0..2]
3 :X-Y < 2	[3..4]	[0..1]

Tableau 3-2. Domaines de chaque variable d'entrée.

Une fois l'ensemble des contraintes utilisé, un jeu d'essais est choisi arbitrairement à partir des domaines des entrées. Par exemple, un jeu de tests satisfaisant est {X=4, Y=0} ou encore {X=3, Y=0}. Il est évident que pour respecter le critère de test structuré il faut répéter l'opération pour tous les autres chemins appartenant à la base. Ainsi v(G) chemins seront traversés et le critère de couverture sera respecté. Cette méthodologie constitue donc une alternative à la deuxième critique d'Evangelist [Evangelist 1984].

3.5 Conclusion

Dans ce chapitre, une vue générale de l'ensemble des techniques de test de logiciels a été présentée. Le test structurel basé sur un critère de couverture a été décrit de manière assez détaillée car cette méthode est adaptée pour la génération de tests en utilisant le code VHDL. Cette approche est basée sur une représentation graphique du programme appelée graphe flot de contrôle (GFC).

Plus précisément, notre choix s'est porté sur un critère de couverture de chemins qui est le critère le plus rigoureux parmi ceux appartenant au test structurel. Or la couverture à 100% des chemins d'exécution est impossible dès qu'il y a des boucles. Cependant, nous avons vu que le critère de test structuré établit qu'un ensemble de chemins indépendants est suffisant pour tester un programme puisque tout chemin supplémentaire est combinaison linéaire de la base formée par cet ensemble. Les inconvénients de cette méthode concernent la génération de la base de chemins et la génération des données à partir de ces chemins.

Pour remédier à ces problèmes nous avons présenté l'algorithme de Poole qui génère un ensemble de chemins linéairement indépendants à partir d'un GFC, et la *procédure de réduction de domaine dynamique* d'Offutt. Cette procédure utilise une technique de satisfaction de contraintes pour extraire les données d'entrée correspondant à un chemin particulier.

Ces concepts qui permettent de valider un logiciel ont été présentés dans le but de les appliquer pour détecter des erreurs de conception dans des programmes VHDL. Le prochain chapitre est donc consacré à l'adaptation de ces techniques pour des programmes écrits en VHDL. Des adaptations sont nécessaires du fait de certaines caractéristiques de ce langage que l'on ne retrouve pas dans les langages de programmation traditionnels, en particulier la notion de temps.

CHAPITRE

4

APPROCHE DE RESOLUTION DU PROBLEME

Au cours du chapitre précédent, nous avons pu voir les différentes techniques utilisées dans le domaine du test de logiciels notamment celles basées sur un critère de couverture. Nous avons vu en détail le critère du test structuré qui requiert l'utilisation d'un graphe de flot de contrôle. Pour pallier aux inconvénients de ce critère [Evangelist 1984], nous avons opté pour l'utilisation de l'algorithme de Poole et d'un générateur de données de test orienté structure. À partir de ces choix, nous allons à présent proposer une approche de résolution pour le problème de génération de vecteurs de test à partir d'une description VHDL comportementale.

Pour cela, nous définissons un graphe de flot de contrôle adapté aux programmes VHDL qui possèdent des caractéristiques que l'on ne retrouve pas dans les langages de programmation traditionnels : notion de temps, interconnexion de *process* s'exécutant en parallèle, mécanisme de retard delta pour les affectations des signaux. Concernant la

génération des données de test à partir des chemins donnés par l'algorithme de Poole, nous nous inspirons de travaux présentés précédemment [Cho *et al.*1991], [Kalyanaraman 1993], [Vemuri *et al.* 1995]. Nous présentons également le sous-ensemble VHDL que nous avons spécifié.

Ce chapitre est organisé de la façon suivante. Dans la première partie nous donnons un aperçu du langage VHDL. Nous présentons les restrictions effectuées sur les descriptions de type comportemental pouvant être prises en compte par notre approche. La deuxième partie présente notre modèle de graphe de flot de contrôle adapté à des programmes écrits en langage VHDL. La troisième partie est consacrée à la génération et à l'analyse des chemins à partir du GFC. On y donne la définition de deux types de chemins particuliers qui nécessitent une analyse : les chemins à ordonnancer et les chemins à modifier. La quatrième partie est dédiée à la traduction de notre problème d'identification des données de test en un problème de satisfaction de contraintes. Les concepts de la programmation logique par contraintes y sont rapidement présentés, ainsi que le modèle de contraintes de Kalyanaraman et Vemuri tenant compte de la spécificité des objets VHDL.

4.1 Concepts de base du langage VHDL

Le langage VHDL a été développé, dans les années 80, par IBM, Intermetrics, ATT sous l'impulsion du DOD (*Departement Of Defense, USA*). Le terme VHDL signifie VHSIC (*Very High Speed Integrated Circuit*) *Hardware Description Language*. En 1987, une première version du langage est standardisée par l'IEEE (*Institute of Electrical and Electronics Enginneers*).

L'application initiale du langage VHDL est la spécification et la simulation. Il a cependant été utilisé, dans un deuxième temps, pour la description en vue de la synthèse. De ce fait le VHDL est un langage unique permettant de faire :

- de la spécification : le langage VHDL est très bien adapté à la modélisation de systèmes numériques complexes. Il permet de concevoir un circuit à différents niveaux d'abstraction et de le décomposer en modules hiérarchiques ;
- de la simulation : la notion de temps, présente dans le langage, permet son utilisation pour décrire des fichiers de simulation (*test-bench* en anglais). Le modèle avec les fichiers de simulation peuvent constituer, ensemble, un cahier des charges. Les

fichiers de simulation peuvent également être utilisés avec un banc de tests de production (*benchmarks* en anglais) ;

- de la synthèse logique : les logiciels de synthèse permettent de traduire la description VHDL en une description au niveau porte logique. Il est ainsi possible d'intégrer la description dans un composant programmable (CPLD, FPGA) ou dans un circuit ASIC.

En VHDL, un circuit électronique est décrit à l'aide d'une entité et d'une architecture de la façon suivante :

- l'entité (*entity* en VHDL) définit les signaux d'entrée et de sortie du circuit, leur type ainsi que leur mode (lecture seule, écriture seule, lecture-écriture) ;
- l'architecture décrit le comportement de l'entité.

L'architecture spécifie la relation entre les entrées et les sorties de l'entité d'une description. Une architecture n'est associée qu'à une seule entité. VHDL accepte trois grands types de descriptions d'architecture :

- les descriptions de type comportemental : elles décrivent, sous forme d'un programme informatique, les instructions séquentielles constituant un *process* et traduisant un algorithme ou un comportement ;
- les descriptions de type flot de données : elles décrivent à partir du flot de données, le comportement des données transitant à l'intérieur de la boîte noire (entité) ;
- les descriptions de type structurel : elles décrivent l'assemblage de différents circuits électroniques, chacun d'eux étant une boîte noire (entité).

VHDL permet également de mixer ces trois types de description à l'intérieur d'une même description.

Une description de circuit au niveau algorithmique telle qu'elle a été présentée dans le chapitre 2 correspond à une description VHDL de type comportemental. Notre étude se limite donc aux descriptions VHDL de ce type.

La Figure 4-1 montre un exemple de description VHDL de type comportemental d'un système digital. La description possède deux entrées, in_1 et in_2 et une sortie out_1 appelées *ports*. On distingue les ports d'entrée des ports de sortie par un mode qui leur est

associé : respectivement *in* (écriture seule) et *out* (lecture seule). On ne prend pas en compte le mode *inout* (lecture-écriture) car il n'a de sens que pour les descriptions structurelles. Le comportement de la description est décrit grâce à des *process* à l'intérieur d'une *architecture* couplée à l'*entité*. Un *process* est un groupe d'instructions séquentielles. Plusieurs *process* se comportent, d'un point de vue externe, comme des groupes d'instructions concurrentes. Les *process* communiquent entre eux par des signaux internes (par exemple a sur la Figure 4-1).

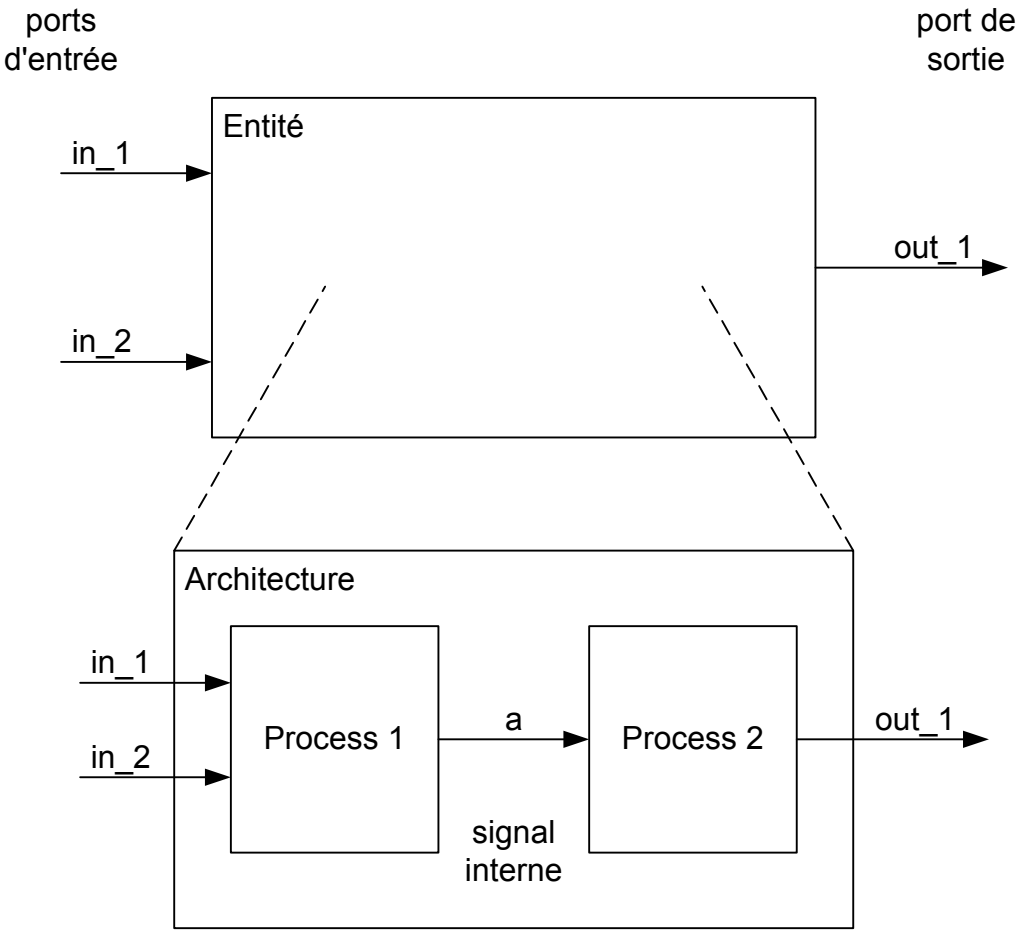


Figure 4-1. Illustration d'une description VHDL de type comportementale.

Nous présentons dans la sous section suivante les principaux éléments du langage VHDL inclus dans l'architecture d'une description de type comportemental.

4.1.1 Les principaux éléments de l'architecture

4.1.1.1 Le *process*

Le *process* est l'objet fondamental manipulé en VHDL. La description du comportement d'un circuit digital est composée d'un ou de plusieurs *process* qui s'exécutent

en parallèle. Un *process* est dit cyclique ; lors de son exécution, lorsqu'on atteint son mot clé final, la simulation recommence à partir de son mot clé initial. Seule l'instruction wait peut suspendre un *process*. Dans le corps d'un *process* déclaré, il ne peut y avoir que des instructions séquentielles (cf. section 4.2.3) c'est à dire exécutées dans l'ordre d'écriture, l'une après l'autre. On retrouve ici la programmation classique.

4.1.1.2 L'instruction wait

Un *process* a deux états : il est soit actif, soit suspendu. L'instruction wait permet de gérer le *process*.

wait {**on** liste_de_signaux} {**until** condition_booléenne} {**for** durée};

Une telle syntaxe signifie que le bloc d'instructions qui suit sera exécuté si une des conditions suivantes est vérifiée :

- Un événement survient sur un des signaux spécifiés dans liste_de_signaux et le résultat de l'évaluation de la condition_booléenne associée à until est vrai. Les signaux sont désignés comme étant les signaux sensibles du *process*. Nous appellerons événement tout changement de valeur survenant sur un signal entre deux cycles de simulation (cf. section 4.1.2.3).
- La durée spécifiée après l'instruction for est écoulée depuis l'instant où le *process* a été suspendu par l'instruction wait.

4.1.1.3 Les différentes classes d'objets

Un objet VHDL est un élément mémorisant qui possède une valeur d'un type donné. Il existe trois classes d'objets :

- **les constantes :** une constante est un objet dont la valeur ne peut être modifiée ;
- **les variables :** une variable est dite locale à un *process*, sa valeur peut être modifiée par affectation. On note A:=B l'affectation de la valeur de la variable B à la variable A. L'affectation d'une variable est effective immédiatement comme dans les autres langage de programmation tel que C ou PASCAL ;
- **les signaux :** un signal est un objet dit global, sa valeur peut être modifiée par affectation. Pour différencier l'affectation d'un signal de celle d'une variable on utilise le symbole « <= ».

Exemples :

```
A<='1' after 50 ns ;
B<=A  ;
```

Dans la première affectation, after 50 ns signifie que le signal A prendra sa nouvelle valeur 50 ns après le temps présent de simulation. La seconde affectation ne comporte pas d'instruction after, ceci signifie after 0 ns.

Parmi les signaux on distingue les ports des signaux internes :

- Un port décrit un signal externe et est déclaré dans l'entité. Les ports permettent la communication avec l'architecture. On peut leur affecter une valeur quand leur mode associé est *in* (c'est là que l'on applique les vecteurs de test) et sont observables quand leur mode associé est *out* (c'est là que l'on observe les sorties de la description VHDL).
- Un signal interne est déclaré dans l'architecture et permet la communication entre les états concurrents à l'intérieur de celle-ci, notamment entre les *process*. Contrairement aux ports d'entrée on ne peut pas affecter une valeur aux signaux internes depuis un *test-bench*.

La principale différence entre une affectation de signaux et celle de variables réside dans le fait que pour le signal, le changement de valeur est effectif seulement à la fin du cycle de simulation (cf. section 4.1.2). L'affectation du signal est différée à cause de son pilote. Lors de l'affectation, le ou les couples *valeur_future : heure_de_simulation* sont placés dans le pilote. La valeur sera effectivement passée au signal au moment de la suspension du *process* par une instruction wait. C'est le pilote du signal qui est affecté et non le signal lui-même. Le signal sera affecté plus tard grâce au mécanisme de retard delta.

La Figure 4-2 montre un exemple simple d'une instruction d'affectation de signal décrivant un inverseur. L'instruction est équivalente à la déclaration d'un *process* possédant un signal d'entrée a (source) et un signal de sortie q (cible). Le *process* est actif si un événement (un changement de valeur du signal) se produit sur le signal a. Chaque fois qu'un événement se produit sur a, l'instruction s'exécute et une nouvelle valeur est programmée pour la cible q. la nouvelle valeur de q, qui est le résultat de l'évaluation de l'expression not a, n'apparaît pas immédiatement sur q, mais après un retard d'un delta. Ce retard est

équivalent à une période de temps de zéro seconde, il est utilisé pour permettre au simulateur de synchroniser l'exécution en parallèle de plusieurs *process*.

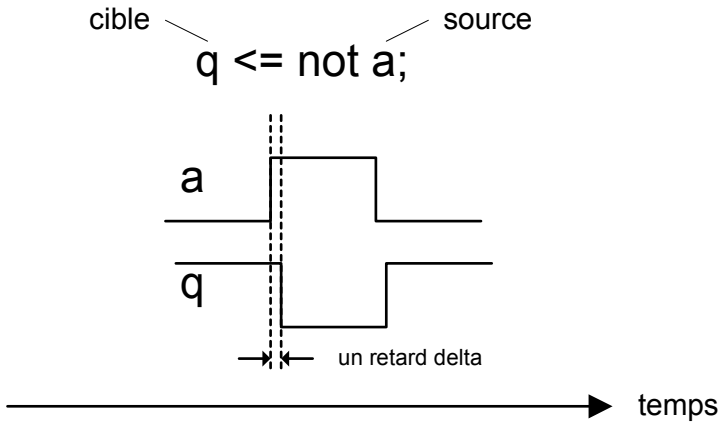


Figure 4-2. L'affectation des signaux.

S'il était spécifié de décrire un inverseur possédant un retard de 10 ns, l'expression not a aurait été suivie par un délai qui aurai eu pour effet de retarder de 10 ns l'affectation de la nouvelle valeur à q :

```
q <= not a after 10 ns ;
```

4.1.2 La simulation des descriptions VHDL

Le langage VHDL est indissociable de la simulation dirigée par événement (à propagation d'événement ou à pilotage événementiel). Elle est basée sur l'occurrence d'un événement. Un événement initial déclenche la simulation et peut causer un nouvel événement ou plusieurs qui peuvent à leur tour causer d'autres événements. La simulation continue jusqu'à ce qu'aucun événement ne survienne.

Un événement est caractérisé par trois paramètres : un identificateur de signal, une nouvelle valeur à imposer au signal et le temps où cette valeur doit être imposée au signal.

La simulation d'une description VHDL se décompose en deux phases : une phase d'initialisation suivie par l'exécution répétitive des *process* dans la description de ce modèle. Chacune de ces répétitions est appelée cycle de simulation. À chaque cycle, les valeurs de tous les signaux dans la description sont calculées. Si suite à ce calcul, un événement se produit sur un signal donné, les *process* qui sont sensibles à ce signal reprennent et sont exécutés en tant qu'éléments du cycle de simulation.

4.1.2.1 La phase d'initialisation

Le simulateur VHDL effectue un cycle de simulation durant lequel tous les *process* de la description sont actifs. Cette phase permet d'établir l'état initial du système. Les valeurs de départ des signaux et variables sont déterminées en se reportant aux règles suivantes :

- La valeur initiale d'un signal (ou d'une variable) est spécifiée dans la déclaration (exemple : `signal a : bit <= '0'`).
- La valeur du signal n'est pas spécifiée, auquel cas cette valeur est implicite : la première valeur du domaine de l'ensemble de définition.

À la fin de ce cycle de simulation toutes les données de la description VHDL ont été initialisées. La phase d'exécution peut alors débuter.

4.1.2.2 La phase d'exécution

C'est au début de cette phase qu'est introduit le vecteur d'entrée. La simulation de la phase d'exécution se fait par scrutation des signaux sensibles, déterminant quels *process* sont actifs et lesquels sont suspendus. La phase d'exécution peut comporter plusieurs cycles de simulation. Elle arrive à son terme lorsque tous les *process* sont suspendus.

4.1.2.3 Le cycle de simulation

Un cycle de simulation peut être un cycle de simulation delta ou un cycle de simulation temps. Le premier cycle de simulation se produit après l'initialisation. Un cycle delta est un cycle de simulation dans lequel le temps de simulation au début du cycle est le même qu'à la fin du cycle. C'est-à-dire que le temps de simulation n'est pas avancé dans un cycle delta.

Un delta (ou bien cycle delta) est une unité de temps infinitésimale, mais quantifiée. Le mécanisme de retard delta est employé pour fournir un retard minimum dans une instruction d'affectation de signal.

Sur la Figure 4-3, on peut voir un exemple de simulation d'un programme VHDL. La description possède un port d'entrée in_1, un port de sortie out_1 et deux signaux internes a,b. La table de simulation correspond aux vecteurs d'entrée in_1 = 0 à t = 0 ns et in_1 = 1 à t = 80 ns. La table de simulation montre que lorsque le temps de retard n'est pas spécifié dans

une affectation de signal (b <= NOT in_1), le changement de valeur sur le signal est effectif seulement après un retard delta.

```
entity essai is
port ( in_1  :in bit;
       out_1:out bit);
 end essai;
architecture behavior of essai is
signal a,b:bit;
begin
   a <= not in_1 after 50 ns;
   b <= in_1;
   out_1<=b;
end behavior;
```

No	temps(ns)	delta	in_1	out_1	a	b
1	0	+0	0	0	0	0
2	50	+0	0	0	1	0
3	80	+0	1	0	1	0
4	80	+1	1	0	1	1
5	80	+2	1	1	1	1

Figure 4-3. Programme VHDL et table de simulation.

La première ligne de la table correspond au temps zéro, tous les signaux sont initialisés à la valeur '0' (la première valeur du domaine de l'ensemble de définition du type bit). La seconde ligne correspond au temps 50 ns +0 delta, le signal a passe à la valeur '1'. Delta est égal à +0 puisque le retard était spécifié. La troisième ligne correspond au forçage à la valeur '1' du port d'entrée in_1 créant ainsi un événement (cases grisées). Le signal b prend sa nouvelle valeur à 80ns +1 delta puisque le retard n'était pas spécifié. Le signal out_1 prend sa nouvelle valeur à 80ns +2.

4.1.3 Un sous ensemble de VHDL pour des spécifications comportementales

Nous venons de voir que le VHDL est un langage de description de matériel permettant de concevoir un circuit à différents niveaux d'abstraction (niveau algorithmique et niveau RTL) et de le décomposer en modules hiérarchiques (description de type comportemental et structurel). La richesse de ce langage dépasse largement nos besoins puisque nous désirons prendre en compte des descriptions VHDL décrites au niveau algorithmique. Il semble donc raisonnable de nous restreindre à un sous-ensemble VHDL qui inclue toutes les constructions utilisées pour une description d'architecture de type comportementale. Contrairement au niveau RTL [IEEE1076.6 1998], on peut noté l'absence de standard qui établit complètement le sous-ensemble VHDL que tout outil de synthèse comportementale doit accepter.

Afin de définir notre sous-ensemble, nous nous somme inspirés des travaux consacrés au problème de la synthèse comportementale (cf. chapitre 2 section 2.2). Ce domaine, qui est

un champ de recherche très actif, nécessite la définition de sous-ensembles VHDL. La synthèse à partir d'une description VHDL est un véritable problème car ce langage possède une très forte sémantique de simulation. Aussi, afin d'éviter de prendre en compte cette sémantique complexe lors de la phase d'analyse de la compilation, tous les outils de synthèse imposent l'utilisation de motifs syntaxiques particuliers pour identifier facilement les éléments matériels qui sont modélisés dans une description VHDL. Le lecteur intéressé peut se référer à [Gajski 1988b], [Roy *et al.* 1992] et [Govindarajan *et al.* 1999].

La grammaire de notre sous-ensemble VHDL est présentée en détail en annexe 1 dans un format similaire au *Backus-Naur Form*. Ses caractéristiques principales sont présentées dans les paragraphes suivants :

4.1.3.1 Type de données

VHDL est un langage permettant la définition de type de données. Nous prenons en compte les types de base : *Boolean*, *bit*, *bit-vector*, *integer* et *enumerated* et les tableaux de simple dimension de ces types.

4.1.3.2 Les opérateurs

Les opérateurs booléens acceptés sont les fonctions logiques préexistantes de VHDL : nand, and, or, nor, xor, not, = (égal), /= (différent) et la fonction & de concaténation. Ces opérateurs sont utilisés lors d'assignation de signaux. Toutes les formes d'assignation sont permises : assignation simple, assignation conditionnelle (de type if_then_else ou de type case) ou assignation itérative (de type for_loop de type while_loop).

4.1.3.3 Les références au temps

Au niveau algorithmique, le temps est représenté par l'ordre dans lequel les affectations des variables son affectées. Notre sous-ensemble n'autorise donc pas d'informations temporelles quantitatives. Chaque déclaration, clause ou attribut (*attribute* en VHDL), avec une référence explicite au temps comme wait for, after, ou comme les attributs 'delayed, 'stable ou 'quiet ne sont pas autorisés.

Les attributs customisés (ceux qui n'ont pas été définis dans les spécifications IEEE [IEEE1076 1993]) et prédéfinis ne sont pas autorisés sauf l'attribut 'event. Associé à un

signal, il renvoie la valeur booléenne VRAI si un événement vient de se produire sur le signal. Sinon l'attribut 'event renvoie la valeur booléenne FAUX.

4.1.3.4 Les process

Les descriptions VHDL contenant plusieurs *process* sont autorisées ainsi que celles contenant des instructions d'affectation de signaux en parallèle. Le mécanisme d'élaboration défini par la norme [IEEE1076 1993] assure en particulier la transformation de ces instructions concurrentes en *process* équivalents. Les signaux sur la partie droite de l'affectation de signal en parallèle forme la liste sensible du *process* équivalent.

C'est pourquoi nous ne nous intéressons par la suite qu'aux *process* VHDL. De cette façon nous considérons toutes spécifications VHDL algorithmiques comme une interconnexion de *process*. En suivant la règle précédente le code VHDL :

```
architecture example1 of example is
signal A,B,C : std_logic_vector(7 downto 0);
      begin
      C <= A and B;
      A <= B and C;
end example1;
```

sera remplacé par :

```
architecture example1 of example is
signal A,B,C : std_logic_vector(7 downto 0);
begin
process(A,B)
      begin
      C <= A and B;
end process;
process(B,C)
      begin
      A <= B and C;
end process;
end example1;
```

Comme une liste sensible peut être utilisée pour remplacer une instruction wait et que les deux expressions sont équivalentes, nous avons choisi, dans notre sous-ensemble VHDL, de ne pas supporter l'utilisation de l'instruction wait. Tous les *process* doivent donc contenir une liste sensible explicite. Les instructions wait on sont remplacées par la liste sensible équivalente. Ainsi le code VHDL suivant :

```
process
begin
      …
      wait on X
end process
```

sera remplacé par :

```
process(X)
begin
      …
end process;
```

De plus, on suppose qu'un signal interne donné ne peut pas être affecté dans deux *process* différents de la même architecture. En effet, dans le cadre de cette étude, on s'est attaché à respecter les constructions présentes dans les descriptions VHDL des *benchmarks* ITC'99 [ITC'Benchmarks 1999]. Ces descriptions ne comportent pas de fonctions de résolution de bus qui permettent au simulateur de résoudre une affectation multiple d'un signal. Puisque nous n'autorisons pas les fonctions de résolution de bus, nous n'autorisons pas les signaux multi-sources.

4.1.3.5 Les fonctions et les procédures

Les fonctions et les procédures, qui sont connues comme des sous-programmes sont autorisées. Les sous-programmes permettent de regrouper une suite d'instructions séquentielles pour décrire une seule fois un calcul particulier devant (ou non) se répéter. Ils peuvent être soit définis localement (dans l'architecture) soit placés dans une unité de compilation appelée *package* et être utilisés globalement dans toutes descriptions.

Nous traitons le sous-programme défini globalement comme un autre programme, c'est-à-dire comme une description VHDL indépendante. Dans l'autre cas (sous-programme localement défini), nous considérons les instructions du sous-programme comme les autres instructions du programme VHDL auquel elles appartiennent.

Maintenant que nous avons défini les constructions VHDL prises en compte dans notre approche, nous présentons dans la section suivante les différents composants du GFC afin de représenter la structure de contrôle d'un programme VHDL.

4.2 Graphe de flot de contrôle de programme VHDL

Nous abordons ici une des parties principales de ce chapitre, à savoir la présentation de notre modèle interne pour le GFC de programme VHDL [Paoli 1998ab, 1999abc]. Considérant que VHDL est un langage de programmation partageant des concepts communs avec des langages conventionnels tels que C ou ADA, il est évident que toutes les descriptions

VHDL peuvent être décrites avec un GFC. Le GFC représente l'ordonnancement des opérations impliquées dans un module de logiciel. Un module de logiciel est défini comme un programme qui peut s'exécuter indépendamment et pouvant faire partie d'un ou plusieurs programmes plus grands. Nous avons fait l'hypothèse qu'une description VHDL de type comportemental est un module de logiciel avec un point d'entrée et un point de sortie. Le GFC correspondant possède donc un nœud d'entrée et un nœud de sortie.

Notre modélisation de GFC pour les programmes VHDL est basée sur les composants décrits pour les GFCs de logiciel exposés dans le chapitre 3 : nœud de début, nœud de fin, nœud de décision, nœud de jonction et nœud simple. Néanmoins certains aménagements sont nécessaires à cause des caractéristiques du langage VHDL : notion de temps, interconnexion de *process* s'exécutant en parallèle, mécanisme de retard delta pour les affectations des signaux.

Dans cette section nous donnons la correspondance entre les instructions VHDL et leur structure graphique en terme de nœuds et d'arcs. On associe à chaque arc une valeur et à chaque nœud une instruction. La valeur de l'arc dépend du type de nœud et peut être nulle.

4.2.1 L'architecture

Un module VHDL est déclaré avec le mot clé architecture, il correspond au nœud de début du graphe. Il ne possède aucun arc entrant et un seul arc sortant. La fin de l'architecture est déclarée avec le mot clé end, suivie optionnellement du nom de l'architecture. Ce mot clé correspond au nœud de fin du GFC. Ce nœud possède un arc entrant qui est l'arc sortant du nœud représentant la dernière instruction présente dans le corps de l'architecture, et aucun arc sortant.

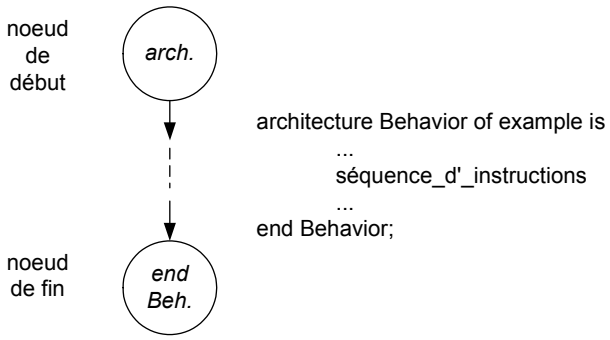


Figure 4-4. Un modèle pour l'instruction architecture.

La Figure 4-4 illustre la modélisation de l'architecture nommée Behavior.

4.2.2 Le process

Le premier begin rencontré après le mot clé architecture repère la zone concurrente. Cette déclaration correspond au nœud *process*. Ce nœud modélise le parallélisme présent dans les descriptions VHDL à travers l'exécution des instructions *process*. Il possède autant d'arc sortant qu'il y a de *process,* plus un arc qui modélise la suspension de tous les *process*. À chaque arc sortant de ce nœud une valeur est associée qui modélise la liste sensible du *process* correspondant. La valeur portée par l'arc symbolisant la suspension de tous les *process*, représente le complément de tous les signaux de la liste sensible de tous les *process*. Cet arc se dirige vers le nœud modélisant l'instruction déclarant le point de sortie du programme, c'est-à-dire le nœud de fin.

L'instruction end process correspond à un nœud de jonction appelé le nœud de fin de *process*. Il représente la fin de l'exécution des *process* actifs de la description. Le nombre d'arcs entrants de ce nœud est égal au nombre de *process* dans la description. L'arc sortant de ce nœud rejoint toujours le nœud *process*. Cet arc sortant modélise le fait que les *process* attendent un événement sur la liste sensible afin de s'exécuter.
L'arc qui joint le nœud *end process* au nœud *process* peut être vu comme la boucle de mise à jour des signaux en fin de cycle de simulation.

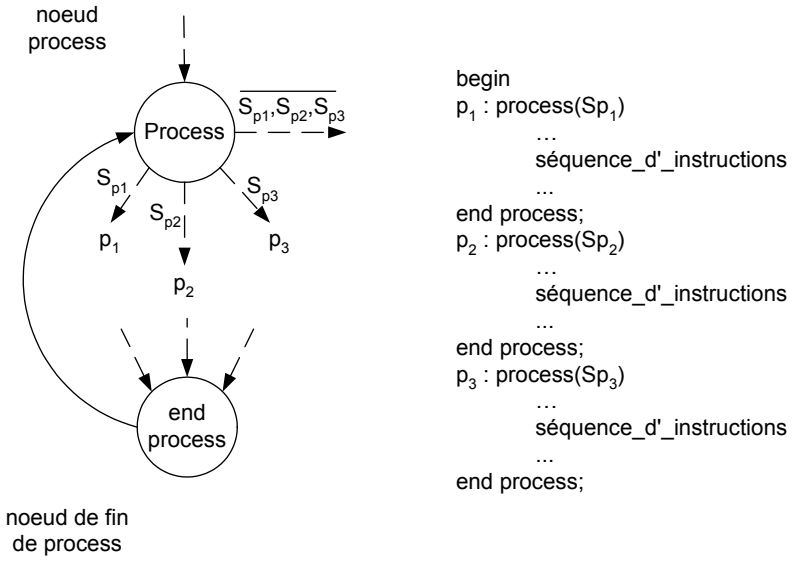


Figure 4-5. Un modèle pour l'instruction process.

La Figure 4-5 illustre la modélisation de trois *process* p_1, p_2, p_3 dont la liste sensible contient respectivement les signaux Sp_1, Sp_2 et Sp_3.

4.2.3 Les instructions séquentielles

Les instructions décrites dans cette sous-section sont des instructions séquentielles que l'on retrouve dans le corps des *process*.

4.2.3.1 Les affectations

Les affectations de variable ou de signal se composent de deux parties distinctes séparées par l'opérateur d'affectation : **=** pour une variable, et **<=** pour un signal. Le signal ou la variable qui doit être assigné est sur la partie gauche de l'expression et la valeur assignée est sur la partie droite. Pour chaque instruction d'affectation un nœud simple doit être créé. Il possède :

- l'expression d'affectation associée ;
- un arc sortant et un arc entrant, les deux sans aucune valeur associée.

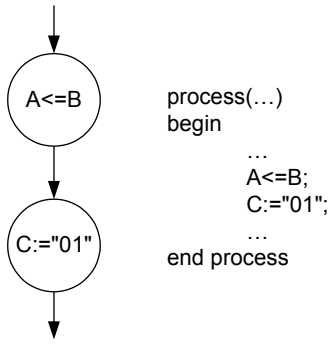


Figure 4-6. Les instructions d'affectations.

La Figure 4-6 illustre la modélisation de deux affectations séquentielles qui propagent pour la première la valeur de B sur le signal A, puis la valeur "01" sur la variable C.

4.2.3.2 Les instructions conditionnelles

Les instructions conditionnelles if et case permettent de sélectionner différents chemins d'exécution en fonction d'une ou plusieurs conditions booléennes.

Les instructions if sont des nœuds de décision bidirectionnel (ou binaire). Elles sont représentées par un nœud qui possède un arc entrant et exactement deux arcs sortants. On associe au nœud l'expression booléenne incluse dans l'expression if_then_else. Chacun des arcs sortants porte une valeur qui représente le résultat de l'évaluation de la condition : VRAI ou FAUX. La Figure 4-7 illustre la modélisation de l'instruction if.

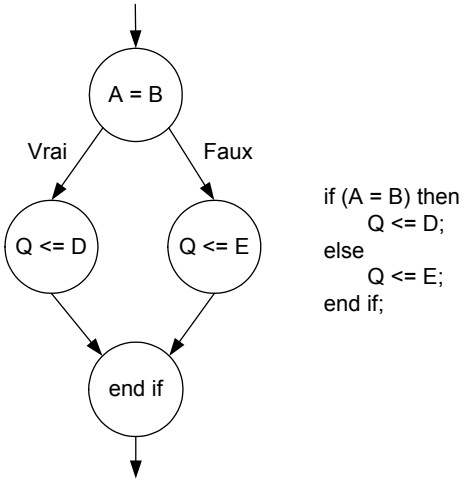


Figure 4-7: Un modèle pour l'instruction *if_then_else*.

Une instruction case est un nœud de décision multidirectionnel. Il existe deux possibilités pour la modéliser.

La première consiste à la représenter comme un ensemble de déclarations if_then_else. Par exemple, une instruction case à trois branches peut être écrite comme :

```
if case = 0 then
      A1
else
      if case = 1 then
            A2
      else
            A3
      endif
endif
```

Nous ne choisissons pas cette représentation pour les raisons suivantes :

- la traduction n'est pas unique. Il y a beaucoup de façon de créer un arbre pour l'enchaînement d'instructions if_then_else qui modélise les branches multiples ;
- ce genre de représentation implique l'ajout de concept d'ordre d'évaluation des décisions, alors que pour une instruction case aucune priorité n'est indiquée pour les conditions impliquées.

Nous avons décidé d'utiliser la seconde possibilité qui consiste à créer un arc pour chacune des branches de l'instruction case. Une instruction case sera représentée comme un seul nœud qui possède un arc entrant et autant d'arcs sortants qu'il y a de conditions indiquées. En VHDL, ces conditions sont représentées par le mot clé when.

On associe au nœud case l'expression à évaluer. Chacun des arcs sortants possède une valeur, qui représente le résultat de l'évaluation de l'expression.

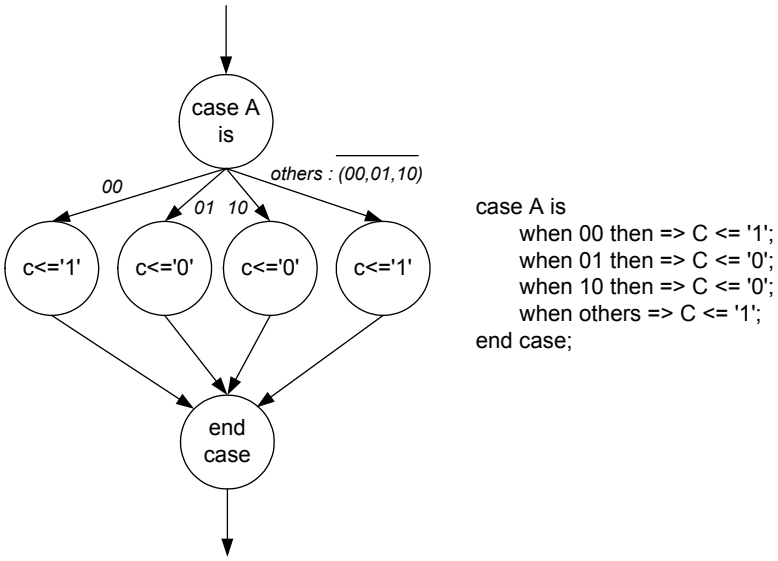


Figure 4-8: Un modèle pour l'instruction case.

La Figure 4-8 illustre la modélisation de l'instruction case. Un arc est construit à chaque spécification d'une branche when. Pour le mot clé others, la valeur portée par l'arc représente le complément de toutes les valeurs qui ont été spécifiées sur les autres arcs.

4.2.3.3 Les instructions itératives

Une instruction itérative comporte une séquence d'instructions VHDL qui doit être exécutée à plusieurs reprises ou pas du tout. D'après notre sous-ensemble VHDL on considère deux types d'instructions itératives : for loop et while loop.

Dans le cas de l'instruction while loop, le mot réservé while avec une condition précède le mot réservé loop. La séquence d'instructions à l'intérieur de la boucle est exécutée si la condition de l'itération est vraie. Si la condition est fausse le contrôle passe à l'instruction end loop.

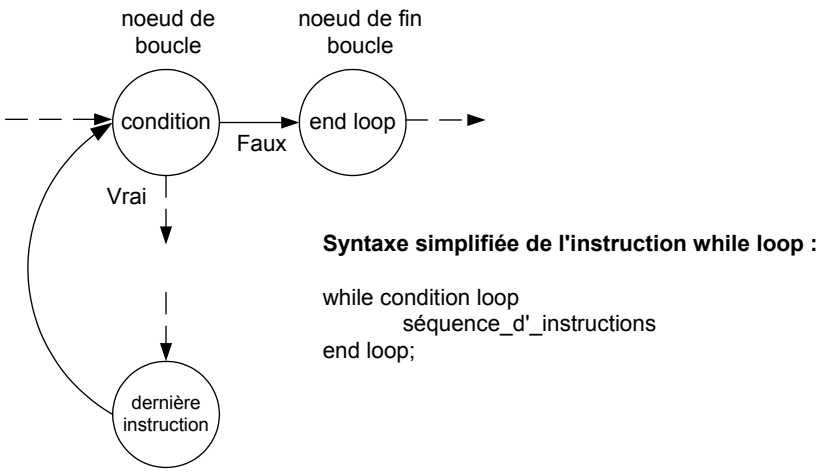


Figure 4-9: Un modèle pour la structure de boucle while.

Les structures while loop sont modélisées par un ensemble de deux nœuds (Figure 4-9) : le nœud de boucle, le nœud de fin de boucle :

- Le nœud de boucle est à la fois un nœud de décision et un nœud de jonction. C'est le nœud d'entrée de la structure de répétition. Il possède deux arcs entrants et deux arcs sortants. Le premier arc entrant vient de l'instruction qui précède la structure itérative, le second vient de la dernière instruction du corps de la boucle. La valeur associée au nœud est l'expression de la condition de la structure while loop. Chaque arc sortant porte une valeur, qui représente le résultat de l'évaluation de la condition de boucle : soit VRAI, soit FAUX.
- Le nœud de fin de boucle est un nœud simple, il marque la fin de la liste des instructions séquentielles qui sont incluses dans le corps de la structure répétitive. Ce nœud possède un arc entrant avec une valeur FAUX qui vient du nœud de boucle et un arc sortant sans valeur, vers l'instruction située après la structure répétitive.

Un autre schéma itératif utile est l'instruction for loop. Dans ce cas le mot clé for avec le paramètre de boucle précède le mot clé loop. L'intervalle discret pour le paramètre de boucle est aussi spécifié. À chaque itération le paramètre de boucle est incrémenté, la valeur initiale du paramètre de boucle est donnée par la valeur la plus à gauche de l'intervalle (Figure 4-10).

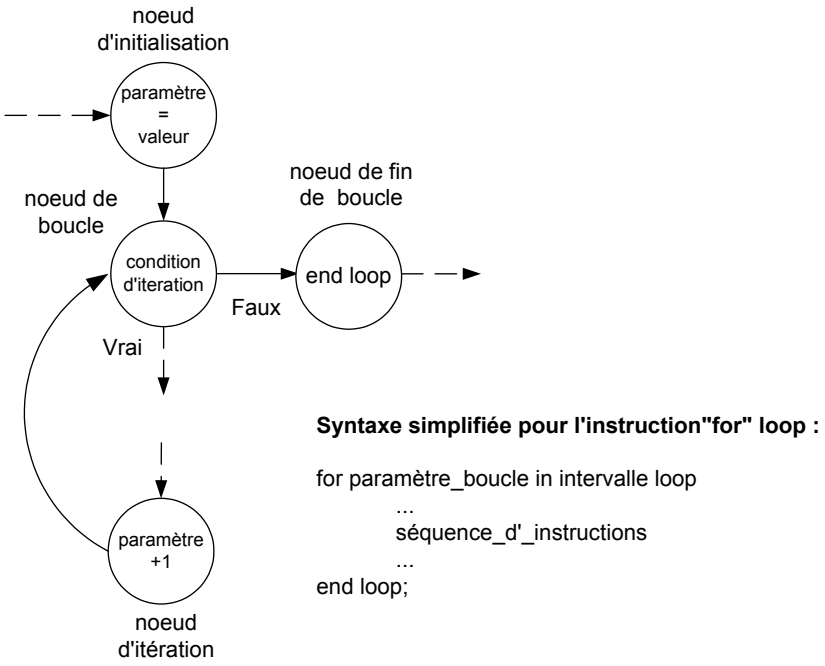


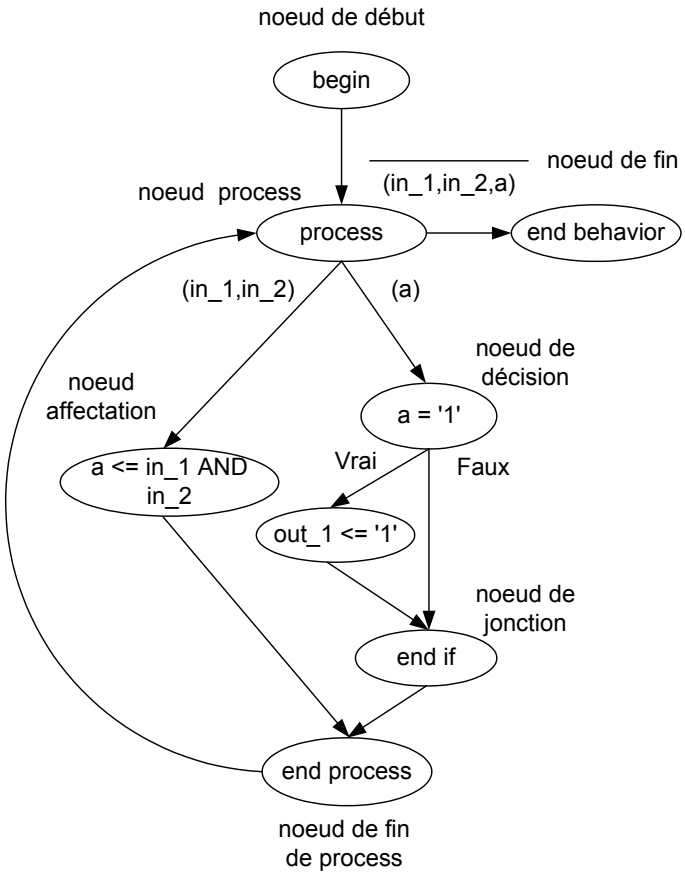
Figure 4-10: Un modèle pour la structure de boucle for.

Les structures for loop sont modélisées par un ensemble de quatre nœuds (Figure 4-10) : le nœud initialisation, le nœud de boucle, le nœud itération et le nœud de fin de boucle :

- Le nœud d'initialisation modélise l'affectation du paramètre de boucle à la valeur la plus à gauche de l'intervalle. Il possède un arc entrant venant de l'instruction précédent la structure itérative et un arc sortant allant vers le nœud de boucle.
- Le nœud de boucle est à la fois un nœud de décision et de jonction. C'est le nœud d'entrée de la structure de répétition. Il a deux arcs entrants et deux sortants. La valeur du nœud est la condition de fin d'itération. Chaque arc sortant possède une valeur, qui représente le résultat de l'évaluation de la condition de fin d'itération : soit VRAI, soit FAUX.
- Le nœud d'itération marque la fin de la liste d'instructions contenues dans le corps de la structure répétitive. Ce nœud possède un arc entrant sans valeur qui vient de la dernière instruction du corps de la boucle, et un arc sortant qui retourne au nœud de boucle.
- Le nœud de fin de boucle est le même que celui décrit pour la structure while loop.

Pour conclure cette section consacrée à la modélisation du flot de contrôle dans un programme VHDL, nous présentons sur la Figure 4-11 un GFC complet et son code VHDL

correspondant. Les valeurs des nœuds (instructions) sont écrites dans les cercles. Les valeurs quand elles existent sont écrites sur les arcs.




```
entity example is
port ( in_1,in_2: in bit;
       out_1: out bit);
end example;
architecture behavior of example is
signal a:bit;
begin
process1: process(in_1, in_2)
begin
          a <= in_1 AND in_2;
end process;
process2: process(a)
begin
if a = '1' then
          out_1 <= '1';
end if;
end process;
end behavior;
```

Figure 4-11. GFC d'un programme VHDL.

On remarque que la description VHDL comporte deux *process* notés process1 et process2. Chacun des *process* est modélisé par les deux branches du nœud *process* dont les deux arcs initiaux portent les listes sensibles respectives. Les nœuds suivants représentent les instructions séquentielles qui se trouvent à l'intérieur du corps des *process*.

La section suivante est consacrée à la génération des chemins à partir du GFC et à leur analyse.

4.3 Génération et analyse des chemins

L'algorithme de Poole génère à partir du GFC une base de chemins indépendants. Cet algorithme, tel qu'il a été présenté au chapitre 3, peut s'appliquer sans aucune modification sur le GFC défini dans la section précédente, puisque ce graphe comporte un nœud de début et un nœud de fin.

Cependant, les chemins générés par cet algorithme ne sont pas, a priori, des chemins d'exécution, mais des sous-chemins de chemins d'exécution. La génération automatique des données de test à partir de ces chemins pose des problèmes liés aux deux caractéristiques suivantes :

- Les variables et les signaux internes sont des objets VHDL qui conservent leur valeur à travers le temps et qui ne sont pas directement contrôlables depuis le *test-bench* comme les ports d'entrée.
- Les descriptions VHDL de type comportemental peuvent comporter plusieurs *process* interconnectés par des signaux internes.

La première caractéristique implique qu'un chemin traversant un nœud de décision associé à une variable ou un signal interne nécessite qu'une séquence de test soit exécutée avant le vecteur de test correspondant au chemin proprement dit. Dans le paragraphe 4.3.2 nous donnons la définition de ce type de chemin appelé **chemin à ordonnancer** ainsi que notre solution pour la résolution de ce problème.

La deuxième caractéristique est propre au langage VHDL. Il en découle que certains chemins de la base ne commencent pas au point d'entrée du programme. Ils doivent être combinés à un chemin de la base qui commence au point d'entrée du programme. Dans le paragraphe 4.3.3 nous donnons la définition de ce type de chemin appelé **chemin à modifier** ainsi que notre solution pour la résolution de ce problème.

Auparavant, nous donnons dans le paragraphe 4.3.1 la définition d'un chemin d'exécution sur un GFC de programme VHDL.

4.3.1 La définition d'un chemin d'exécution

Un chemin complet ou chemin d'exécution est une suite d'instructions réellement parcourues pendant l'exécution d'un programme. Un chemin d'exécution d'un module de logiciel peut toujours être représenté par un chemin à travers son GFC qui commence au point d'entrée du programme (nœud de début) et qui se termine au point de sortie du programme (nœud de fin). Concernant les programmes VHDL, le point d'entrée correspond à l'injection des vecteurs de test sur les ports d'entrée à un temps t, alors que le point de sortie correspond à la suspension de tous les *process* à un temps $t + n.\delta$, où n correspond au nombre de cycle δ

(delta). Le *process* est modélisé par un nœud de décision qui contrôle si un événement survient sur un signal sensible. Tant qu'un événement se produit sur un signal appartenant à la liste sensible associée, le *process* sera exécuté. Ce type particulier de décision est appelé « boucle infinie » en référence aux programmes qui s'exécutent à plusieurs reprises sans s'arrêter.

De ce fait, un module VHDL est appelé module « boucle infinie ». La particularité de ce type de module est que l'on ne peut pas toujours définir un chemin d'exécution comme un chemin à travers le GFC qui débute au point d'entrée du programme et se termine au point de sortie du programme. En effet, un chemin donné peut traverser une instruction qui provoque un événement sur un signal interne. Les *process* qui sont sensibles à ce signal reprendront et seront exécutés impliquant la traversée d'un ou de plusieurs sous-chemins (suite d'instructions traversées appartenant à un chemin) supplémentaires non prévus dans le chemin donné, jusqu'à ce que tous les *process* soient suspendus.

L'algorithme de Poole génère donc un ensemble de chemins qui ne sont pas a priori des chemins d'exécution, mais des sous-chemins de chemins d'exécution. Nous choisissons de les considérer comme des chemins d'exécution et de ne pas s'occuper de savoir si un où plusieurs sous-chemins seront traversés ensuite. Ce choix est guidé par les deux points suivants :

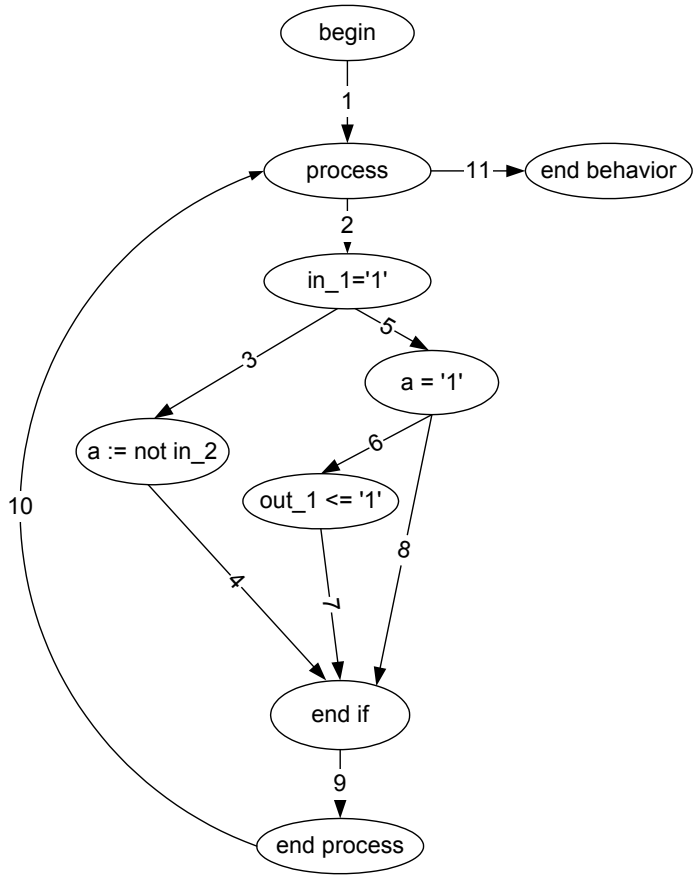
- le critère du test structuré requiert la couverture de tous les chemins de la base ;
- les boucles infinies ne sont pas réellement infinies, elles s'exécutent tant qu'un événement les active.

De ce fait, si lors de son exécution un chemin couvre plus de code qu'il était prévu, le critère de couverture du test structuré est quand même respecté. De plus, l'exécution du chemin s'arrêtera forcement à un moment donné si plus aucun événement n'est déclenché.

Néanmoins un certain nombre de chemin de la base nécessitent un traitement supplémentaire avant que la génération des données de test permettant leur exécution soit possible. Ces chemins font l'objet des deux sections suivantes.

4.3.2 Les chemins à ordonnancer

Il s'agit dans cette section de donner une définition pour les **chemins à ordonnancer**. Sur la Figure 4-12 nous avons représenté un exemple de programme VHDL et son GFC simplifié. La complexité cyclomatique est égale à 4 (11 arcs – 9 nœuds + 2), donc la base, donnée par l'algorithme de Poole, compte quatre chemins indépendants. On note B = (Ch1, Ch2, Ch3, Ch4) l'ensemble des chemins de la base.




```
entity example is
port (in_1, in_2 :in bit;
        out_1:out bit);
end example;

architecture behavior of example is

begin

process1: process(in_1)
variable a:bit;

begin

if (in_1='1') then
        a:=not in_2;
elsif (a='1') then
        out_1 <= '1';
end if;
end process;

end behavior;
```

Ch1:1.11
Ch2:1.2.3.4.9.10.11
Ch3:1.2.5.6.7.9.10.11
Ch4:1.2.5.8.9.10.11

Figure 4-12. Exemple de GFC avec des chemins à ordonnancer.

Essayons de trouver « manuellement » les données d'entrée qui permettent l'exécution du chemin Ch3. Le chemin traverse l'arc 2 donc un événement doit se produire sur le port d'entrée in_1. Puis le chemin traverse l'arc 5 donc après l'événement le port d'entrée in_1 doit être différent de '1' donc égal à '0' (en accord avec le type bit). Ensuite le chemin traverse l'arc 6, ce qui implique que la variable a doit être égale à '1'. La table suivante présente les vecteurs de test solution qui permettent de traverser le chemin Ch3 :

temps(ns)	in_1	a
t1	1	-
t2	0	1

Tableau 3. Solution pour le chemin Ch3.

Deux vecteurs de test sont nécessaires au déclenchement de l'événement sur le port d'entrée in_1, respectivement au temps t1 et t2 avec t1 < t2. Cependant on ne peut pas simuler cette séquence de test car les variables (par exemple a) ne sont pas, comme les ports d'entrée, directement contrôlables depuis le *test-bench*. Un vecteur de test ou une séquence de vecteurs de test devra être exécuté avant les vecteurs de test permettant de traverser le chemin Ch3. Cette séquence devra affecter la valeur '1' à la variable a.

La table de simulation suivante montre le résultat de l'exécution d'un triplet de vecteurs : {v1,v2,v3} sur notre description VHDL (cases grisées des lignes 1 à 3 du Tableau 4) qui permet de traverser le chemin Ch3.

No	temps(ns)	delta	in_1	in_2	out_1	a	vecteur
1	0	+0	0	0	0	0	v1
2	10	+0	1	0	0	1	v2
3	30	+0	0	0	0	1	v3
4	30	+1	0	0	1	1	-

Tableau 4. Table de simulation.

On remarque que la variable a est affectée à '1' grâce au couple de vecteurs {v1,v2} alors que le couple de vecteurs {v2,v3} correspond aux données d'entrée solutions pour traverser le chemin Ch3. On appellera par la suite un chemin tel que Ch3 : un **chemin à ordonnancer**. Si on se réfère au GFC de la Figure 4-12, le premier couple de vecteur correspond à l'exécution du chemin Ch2 et le second au chemin Ch3. Comme les variables conservent leur valeur à travers le temps, les chemins sont traversés à des temps de simulation différents. Dans notre exemple, le chemin Ch3 n'est pas le seul chemin à ordonnancer, c'est aussi le cas du chemin Ch4 qui traverse le nœud de décision : if a='1' par l'arc 8.

En conséquence, deux problèmes se posent :

i. Comment identifier les chemins à ordonnancer parmi les chemins de la base ?
ii. Comment produire la séquence de test précédant l'exécution du chemin à ordonnancer ?

À partir des observations antérieures et afin de répondre à ces deux questions, on propose d'utiliser un graphe issu des techniques de compilation : le graphe de dépendance.

4.3.2.1 Le graphe de dépendance

Le Graphe de Dépendance (GdD) est un graphe orienté représentant les dépendances entre les diverses instructions d'un programme. Le graphe est basé sur les mêmes nœuds que le GFC, c'est-à-dire les instructions VHDL, mais les arcs représentent le lien de dépendance entre deux nœuds.

On utilise la définition suivante : un nœud Q est dépendant d'un autre nœud P si l'objet (variable ou signal) affecté en P apparaît dans l'instruction conditionnelle associée à Q (c'est-à-dire un nœud de décision). La Figure 4-13 illustre la dépendance (arc en pointillé) entre le nœud de décision Q et un nœud d'affectation P.

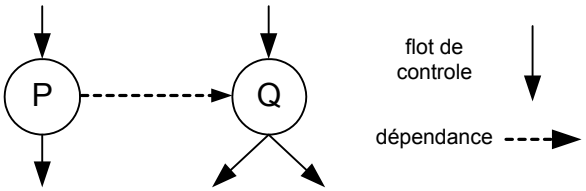


Figure 4-13. Flot de dépendance.

Dans notre définition du GdD, nous considérons que la dépendance concerne seulement les nœuds de décision. En effet, c'est à partir de l'expression associée au nœud de décision que l'on détermine quel arc, à partir de ce nœud, est traversé.

La Figure 4-14 reprend l'exemple précédent de description VHDL où la dépendance entre le nœud affectation `a := not in_2` et le nœud de décision `if a='1'` est illustrée par un arc en pointillé. Les chemins qui traversent le nœud de décision `if a='1'` sont liés aux chemins passant par le nœud `a := not in_2`. Les chemins qui traversent le nœud de décision `if a='1'` sont donc des chemins à ordonnancer. Une séquence de test doit précéder les vecteurs de test correspondant au chemin à ordonnancer. La séquence de test représente les valeurs d'entrée permettant l'exécution du chemin qui traversent le nœud d'affectation `a := not in_2`, c'est à dire le chemin Ch2.

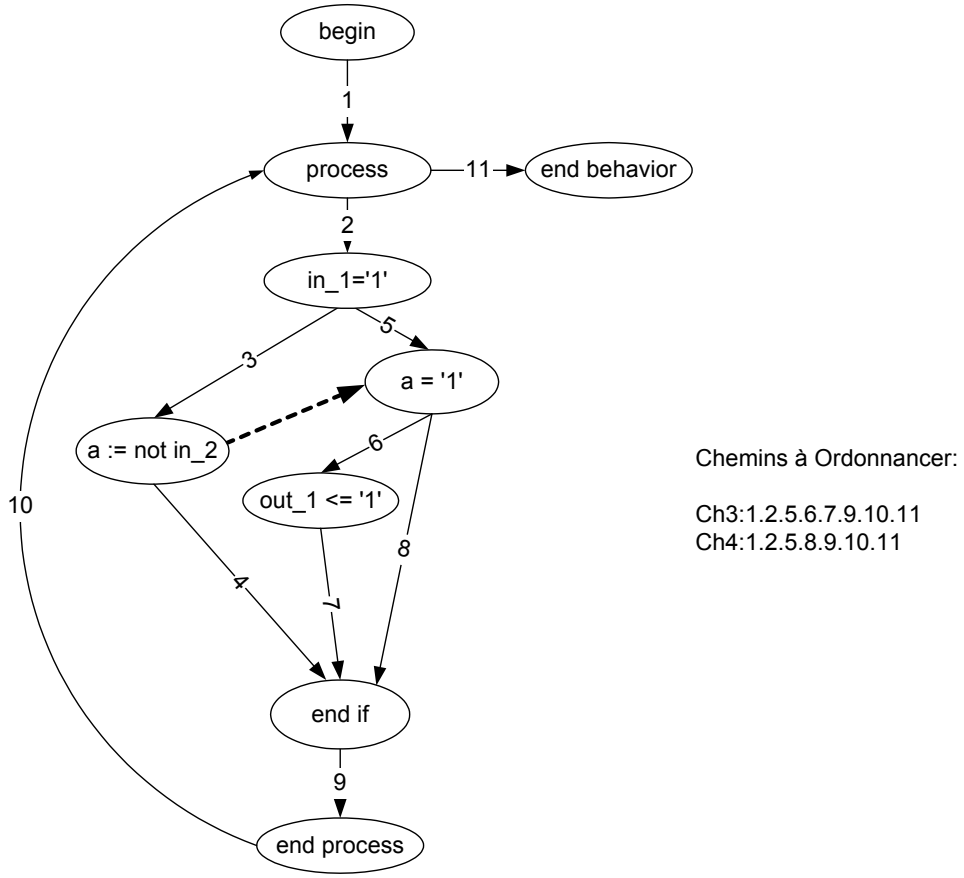


Figure 4-14. GFC et GdD d'un programme VHDL.

Cet exemple est un cas simple de chemin à ordonnancer. En effet, un seul nœud de décision impliquant une variable ou un signal interne doit être traversé. La séquence de test qui précède le vecteur de test correspondant au chemin à ordonnancer comporte au plus deux vecteurs de test (nécessaires à la création d'un événement sur la port d'entrée in_1). De plus, seulement un chemin de la base traverse le nœud d'affectation a := not in_2. Aucun choix ne se pose donc puisqu'une seule solution est possible.

La section suivante présente la notion de **liste d'ordonnancement** qui est utilisée dans cas de chemin à ordonnancer plus complexe.

4.3.2.2 La liste d'ordonnancement

Les chemins à ordonnancer sont plus compliqués que ceux étudiés sur l'exemple de la section précédente. On observe les deux cas suivants :

- plusieurs chemins solution peuvent traverser le nœud d'affectation de la variable ou du signal interne ;

- le chemin à ordonnancer peut traverser plusieurs nœuds de décision associés à un signal interne ou à une variable.

Concernant le premier point, nous avons vu dans la section 4.3.1 que les chemins de la base peuvent, durant leur exécution, traverser plus de code que ce qui est spécifié par le chemin lui même. Les signaux internes et variables peuvent alors être affectés à une valeur non désirée. Cela entraîne un conflit entre la valeur du signal interne ou de la variable affectée dans le chemin a priori solution et la valeur de ce même signal interne ou variable impliqué dans le chemin à ordonnancer. On choisit donc de conserver tous les chemins solution sous la forme d'une liste de chemin. Si après le processus de génération de données de test un conflit survient, on choisira un autre chemin à partir de cette liste.

Concernant le second point, la séquence de test comportera alors plusieurs vecteurs de test, au maximum deux par condition traversée. En effet, deux vecteurs sont nécessaires pour créer un événement sur un port d'entrée.

On propose de trouver ces vecteurs de test à partir d'une **liste d'ordonnancement**. Pour chaque condition à traverser on aura une liste de chemins solution. La liste d'ordonnancement se présente donc sous la forme d'une liste de liste de chemins.

On peut maintenant répondre aux deux questions posées à la fin de la section 4.3.2 :

i. Comment identifier les chemins à ordonnancer parmi les chemins de la base ?
   Un chemin de la base qui traverse des nœuds de décision impliquant des signaux internes ou des variables est un **chemin à ordonnancer**. À un chemin à ordonnancer correspond une **liste d'ordonnancement**.
ii. Comment produire la séquence de test précédant l'exécution du chemin à ordonnancer ?
   La séquence de test précédant l'exécution du chemin à ordonnancer est produite à partir de sa liste d'ordonnancement.

La liste d'ordonnancement est composée d'autant de sous-listes qu'il y a de nœuds de décision sur un signal interne ou une variable à travers le chemin à ordonnancer. Chaque sous-liste représente l'ensemble des chemins solution susceptibles de produire les vecteurs de test de la séquence complète correspondant à la liste d'ordonnancement. Le Tableau 5 illustre

un cas de chemin à ordonnancer qui traverse 3 nœuds de décision : condition 1 à condition 3. À chaque condition correspond une liste de chemins solution. Ces chemins permettent de traverser le nœud d'affectation du signal interne ou de la variable impliqué dans la condition.

Conditions traversées par le chemin	Liste des chemins solutions
Condition 1	**Ch1, Ch2**
Condition 2	**Ch3, Ch4, Ch5**
Condition 3	**Ch6**

Tableau 5. Création d'une liste d'ordonnancement.

La liste d'ordonnancement associée au chemin à ordonnancer sera la suivante : ((Ch1, Ch2) (Ch3, Ch4, Ch5) (Ch6)). La séquence de test à exécuter avant le vecteur de test du chemin à ordonnancer doit être formée à partir d'une des permutations de la liste d'ordonnancement. On considère les permutations de la liste d'ordonnancement car à chaque sous-liste correspondent des vecteurs de test et on ne sait pas dans quel ordre ils devront être exécutés.

Cependant il se peut que le chemin à ordonnancer soit lui même présent à l'intérieur de la liste d'ordonnancement associée. On distingue alors deux cas :

1. Le nœud affectation est avant le nœud de décision (voir schéma gauche de la Figure 4-15) : Dans ce cas, le chemin à choisir est lui même, il n'aura pas besoin d'ordonnancement. En effet la variable impliquée dans le nœud contrôle du chemin est affectée dans ce même chemin mais ce nœud affectation est exécuté avant le nœud contrôle.
2. Le nœud affectation est après le nœud de décision (voir schéma droit de la Figure 4-15) : Dans ce cas, ce choix n'est pas obligatoirement le bon, on conservera ce chemin dans la liste d'ordonnancement.

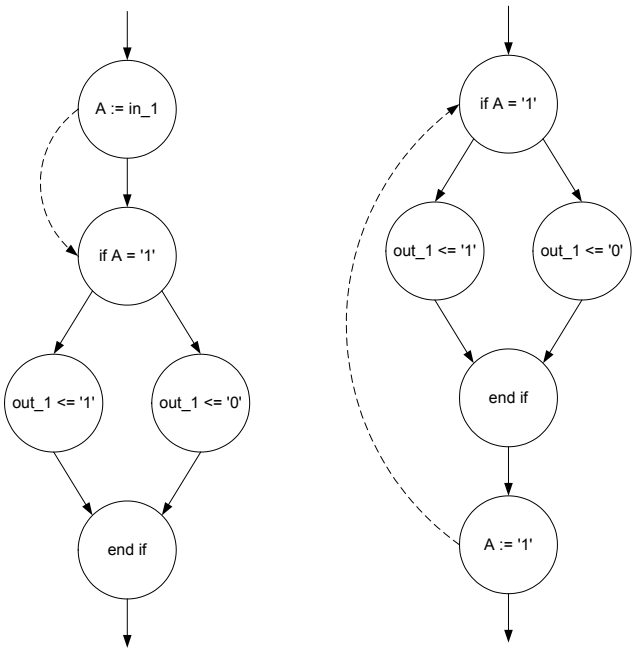


Figure 4-15. Chemin à ordonnancer dans la liste d'ordonnancement.

Notons que ce problème ne se pose que dans le cas d'une variable. En effet un signal permet de connecter deux *process* et d'après nos restrictions il ne peut être affecté que dans un seul *process*. On ne peut donc pas théoriquement, pour une condition sur un signal, trouver le chemin à ordonnancer à l'intérieur de la liste d'ordonnancement associée.

4.3.3 Les chemins à modifier

Il s'agit dans cette section de donner une définition des **chemins à modifier**. Pour cela on s'appuie sur le Graphe de Modélisation de Process (PMG en anglais pour *Process Model Graph*) défini par Cho et Armstrong [Cho *et al.*1991] qui représente la connectivité entre plusieurs *process* interconnectés par des signaux internes. Cho et Armstrong ont défini la notation suivante pour la représentation formelle des interconnections de *process* :

P : un ensemble d'instructions *process* dans une description VHDL ;

p_i : le $i^{ème}$ *process* appartenant à P ;

$(I_S)p_i$: un ensemble de signaux qui sont des entrées de p_i et appartiennent à sa liste sensible ;

$(I_{NS})p_i$: un ensemble de signaux qui sont des entrées de p_i et qui n'appartiennent pas à sa liste sensible ;

$(O)p_i$: un ensemble de signaux qui sont des sorties de p_i.

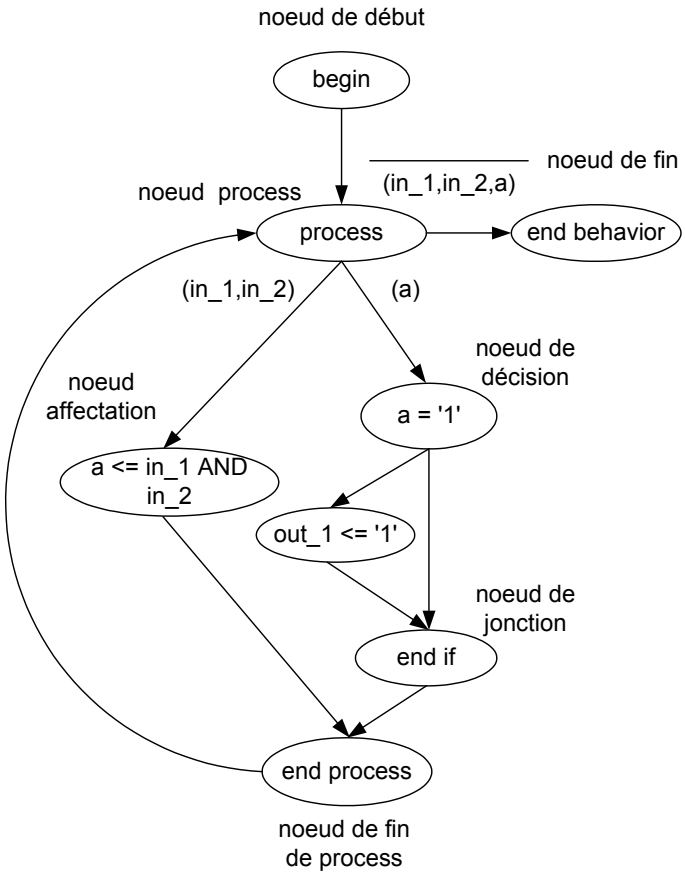
En utilisant la notation définie ci-dessus, un *process* p peut être représenté comme :

$p = ((I_S)p, (I_{NS})p, (O)p)$

La définition suivante peut être utilisée pour définir la connectivité à travers des *process* :

Deux *process* p_i et p_j sont fortement connectés si $(O)_{Pi}$ et $(I_S)_{Pj}$ possèdent un élément en commun, c'est-à-dire si un signal de sortie d'un *process* appartient à la liste sensible d'un autre *process*.

Par commodité on reproduit l'exemple de la description VHDL présenté à la fin de la section 4.1.3 sur la Figure 4-16.




```
entity example is
port ( in_1,in_2: in bit;
       out_1: out bit);
end example;
architecture behavior of example is
signal a:bit;
begin
process1: process(in_1, in_2)
begin
        a <= in_1 AND in_2;
end process;
process2: process(a)
begin
if a = '1' then
        out_1 <= '1';
end if;
end process;
end behavior;
```

Figure 4-16. GFC d'un programme VHDL.

Les *process* process1 et process2 peuvent être représentés respectivement comme $p_1 = (\{in1, in2\}, \{\}, \{a\})$ et $p_2 = (\{a\}, \{\}, \{out1\})$.

D'après la définition précédente, p_1 et p_2 sont fortement connectés puisque $(O)_{P1}$ et $(I_S)_{P2}$ possèdent {a} comme élément en commun. La Figure 4-17 illustre cette forte

connectivité sous la forme d'un PMG. Un cercle représente un *process* et un arc représente soit une connexion entre deux *process* c'est à dire un signal interne, soit une connexion vers l'extérieur c'est à dire un port.

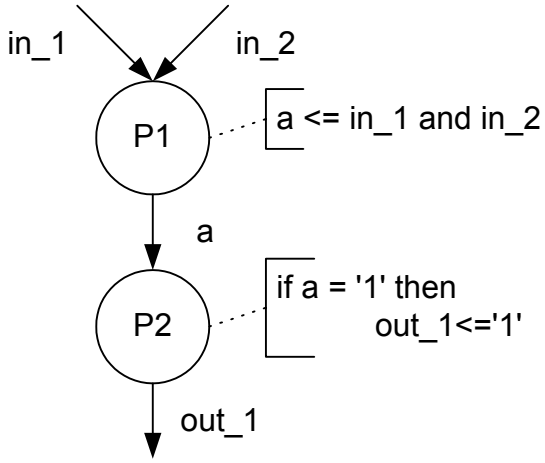


Figure 4-17. Graphe de modélisation de process.

La forte connectivité entre deux *process* implique que certains chemins de la base ne débutent pas au point d'entrée du programme. Chercher des valeurs d'entrée pour ce type de chemins (par exemple les chemins de p_2) n'aurait pas de sens. En effet, les valeurs trouvées ne correspondraient pas à des valeurs à appliquer sur les ports d'entrée et ne permettraient pas l'exécution du code correspondant au chemin.

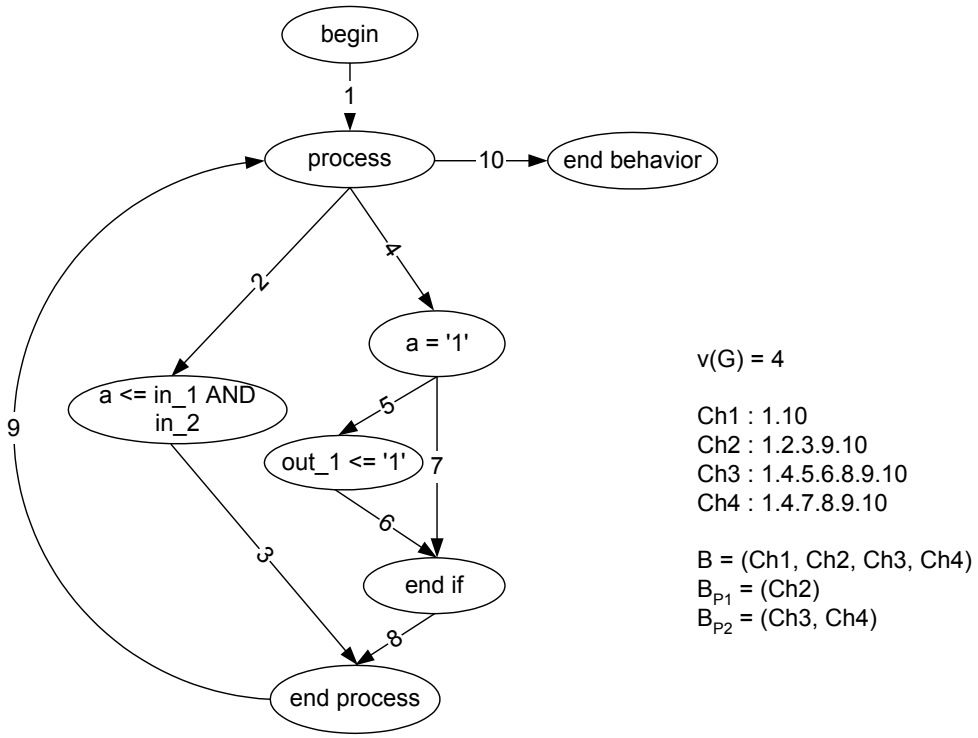


Figure 4-18. Un CFG simplifié et sa base de chemins.

La Figure 4-18 montre un GFC simplifié du programme VHDL précédent ainsi que le résultat de l'application de l'algorithme de Poole. Les arcs sont numérotés de 1 à 10 alors que les nœuds sont au nombre de 8.

On trouve v(G) égal à 4 (10-8+2). La base compte 4 chemins indépendants notés de Ch1 à Ch4. L'algorithme de Poole génère un chemin indépendant pour chaque branche de chaque nœud de décision rencontré à travers le GFC. On note B = (Ch1, Ch2, Ch3, Ch4) l'ensemble des chemins de la base. Les sous-ensembles de B notés B_{p1} = (Ch2) et B_{p2} = (Ch3, Ch4) ne traversent respectivement que les *process* p_1 et p_2.

Essayons de trouver « manuellement » les données d'entrée qui permettent l'exécution du chemin Ch3. Le chemin traverse l'arc 4, un événement doit donc se produire sur le signal interne a. Puis le chemin traverse l'arc 5, donc après l'événement, le signal interne a doit être égal à '1'. Or nous savons qu'il n'est pas possible de déclencher depuis le *test-bench* un événement sur un signal interne. En fait, le chemin Ch3, comme tous les chemins de B_{P2}, ne débute pas au point d'entrée du programme.

D'après la Figure 4-17, il est évident que le *process* p_1 doit être exécuté avant le *process* p_2. En effet, le *process* p_2 pour être activé réclame un événement sur le signal interne a qui est affecté dans le corps du *process* p_1. La table de simulation du Tableau 6 montre le résultat de l'exécution d'un couple de vecteurs {v1,v2} sur notre description VHDL (cases grisées des lignes 1 et 2) qui permet de traverser Ch3.

No	temps(ns)	delta (δ)	in_1	in_2	out_1	a	vecteur
1	0	+0	0	0	0	0	v1
2	10	+0	1	1	0	0	v2
3	10	+1	1	1	0	1	-
4	10	+2	1	1	1	1	-

Tableau 6. Table de simulation.

On remarque que l'événement sur le signal a est provoqué par l'intermédiaire d'un événement sur les ports d'entrée in_1 et in_2. On traverse en fait le chemin Ch2 qui affecte le signal a avant de traverser le nœud de décision impliquant a via le chemin Ch3. Cependant, contrairement au cas des chemins à ordonnancer, les chemins Ch2 et Ch3 sont traversés au même temps de simulation (10 ns) mais à des retards delta (δ) différents. En fait, si on se réfère au GFC de la Figure 4-18, les deux chemins sont combinés en un seul qui traverse les

arcs : 1.2.3.9.4.5.6.8.9.10. La solution est donc de considérer le chemin Ch3 comme un sous-chemin et de le remplacer dans la base par un nouveau chemin. Ce nouveau chemin est une combinaison de lui même et d'un chemin traversant l'affectation de a et débutant au point d'entrée du programme. La séquence de test qui correspond à ce nouveau chemin est le couple de vecteurs (v1,v2). On appellera par la suite un chemin tel que Ch3 : un **chemin à modifier**.

Nous avons donc la définition suivante : « On appellera **chemin à modifier** tout chemin de la base appartenant à un *process* fortement connecté qui nécessite un événement sur un signal interne ».

On remarque que ce problème ne se pose pas si la description VHDL comporte un seul *process*. Dans ce cas la liste sensible ne comporte que des ports d'entrée puisque les signaux internes servent à la communication entre plusieurs *process*.

De façon plus pratique, un chemin peut impliquer un événement sur un signal interne suivant deux conditions, l'une explicite, l'autre implicite :

- Condition explicite : le chemin traverse un nœud de décision du type signal'event.
- Condition implicite : le chemin appartient à un *process* fortement connecté.

Dans les deux cas, le signal appartient à la liste sensible du *process* traversé par le chemin. Pour illustrer ces conditions, prenons comme exemple une description VHDL contenant deux instructions *process*. Trois cas de figure peuvent se présenter et sont illustrés sur la Figure 4-19 qui reprend la notation de Cho et Armstrong.

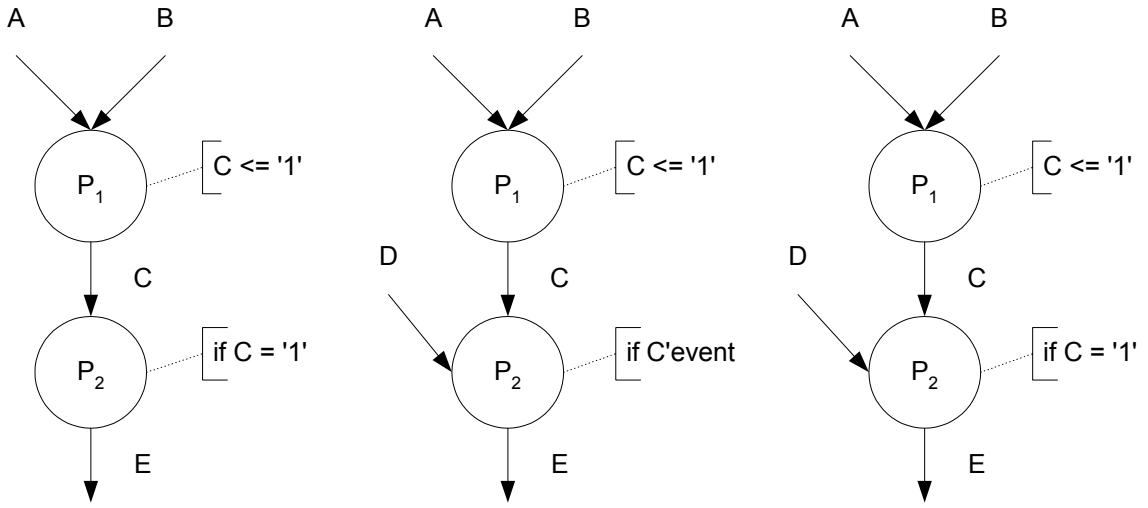


Figure 4-19. Cas de figure pour les chemins à modifier.

Dans le premier et deuxième cas, les chemins qui appartiennent à p_2 doivent être modifiés. En effet, p_2 ne peut être activé que si un événement se produit sur C qui est affecté dans p_1. Dans le premier cas, C est le seul signal de la liste sensible de p_2, et dans le deuxième, une condition explicite d'événement sur le signal C est demandée par le chemin à travers la condition : if C'event.

Dans le troisième cas, deux solutions nous sont offertes :

i. les chemins qui appartiennent à p_2 sont modifiés avec un chemin de p_1. En effet, p_2 peut être activé si un événement se produit sur C qui est affecté dans p_1 ;
ii. les chemins qui appartiennent à p_2 sont ordonnancés avec un chemin de p_1. En effet, p_2 peut être activé si un événement se produit sur D alors que C est affecté dans p_1.

Dans cette configuration, on aura deux possibilités pour traverser le chemin. Soit on l'ordonnance, soit on le modifie, les deux solutions sont possibles. Notre but étant de traverser le chemin, on ne s'est pas préoccupé dans cette recherche de savoir quelle était la meilleure des deux solutions concernant le temps d'exécution ou la place mémoire. On fait donc un choix a priori en gardant la possibilité de revenir à un autre choix de chemin au cas où il y aurait un conflit de valeur lors du processus de génération de données de test. Néanmoins, la modification d'un chemin de la base soulève deux questions :

i. Le critère du test structuré est il toujours respecté ?
ii. Le nouvel ensemble de chemins forme t-il lui aussi une base ?

En ce qui concerne le premier point, le critère du test structuré est toujours respecté puisque une fois exécuté, le chemin modifié traverse plus de code que prévu. Le deuxième point est clarifié par le fait que le chemin à modifier est remplacé dans la base par une combinaison linéaire de lui même avec un chemin de la base. Le nouvel ensemble de chemins est donc lui aussi un ensemble de chemins indépendants, donc une base.

4.3.3.1 Généralisation

Si on généralise pour un programme VHDL de type comportemental avec n *process* $p_1, p_2, .. , p_n$ où $n \geq 1$. Soit C l'ensemble des v(G) chemins de la base $C = (C_1, C_2, .. , C_{v(G)})$. Pour chaque *process* p_i il existe un sous-ensemble de C noté C_{pi} qui est l'ensemble des chemins traversant p_i. Soient deux *process* p_i et p_j fortement connectés et leur ensemble de chemins indépendants associés $C_{pi} = (C_1, C_2, .. , C_q)$ et $C_{pj} = (C_r, C_{r+1}, .. , C_s)$, avec $q < r < s <$ v(G), alors chaque chemin de C_{pj} est un chemin à modifier : il doit être remplacé par une combinaison linéaire d'un chemin de C_{pi} et de lui même. Il y a $q * (s - r + 1) << v(G)^2$ chemins d'exécution potentiels dans le programme qui traversent p_i et p_j. Pour chaque chemin de C_{pj} on choisira une combinaison linéaire parmi les q possibilités offertes par C_{pi}.

On sait donc que parmi les q possibilités il y a au moins un chemin valable. Se pose alors le problème du choix des chemins parmi les q possibilités car si on choisit une mauvaise possibilité alors le chemin est infaisable et aucune donnée d'entrée ne permettra l'exécution du chemin modifié. On présente dans la section suivante une approche permettant de limiter le nombre de ces chemins infaisables.

4.3.3.2 La liste des chemins solution

On désigne par **liste des chemins solution**, la liste des chemins pouvant se combiner à un chemin à modifier. D'après la définition précédente le nombre de chemins de cette liste est inférieur ou égal au nombre de chemins associés au process connecté (noté q).

Le nombre de ces chemins peut être réduit grâce au GdD utilisé dans le cas des chemins à ordonnancer. La Figure 4-20 illustre cette situation.

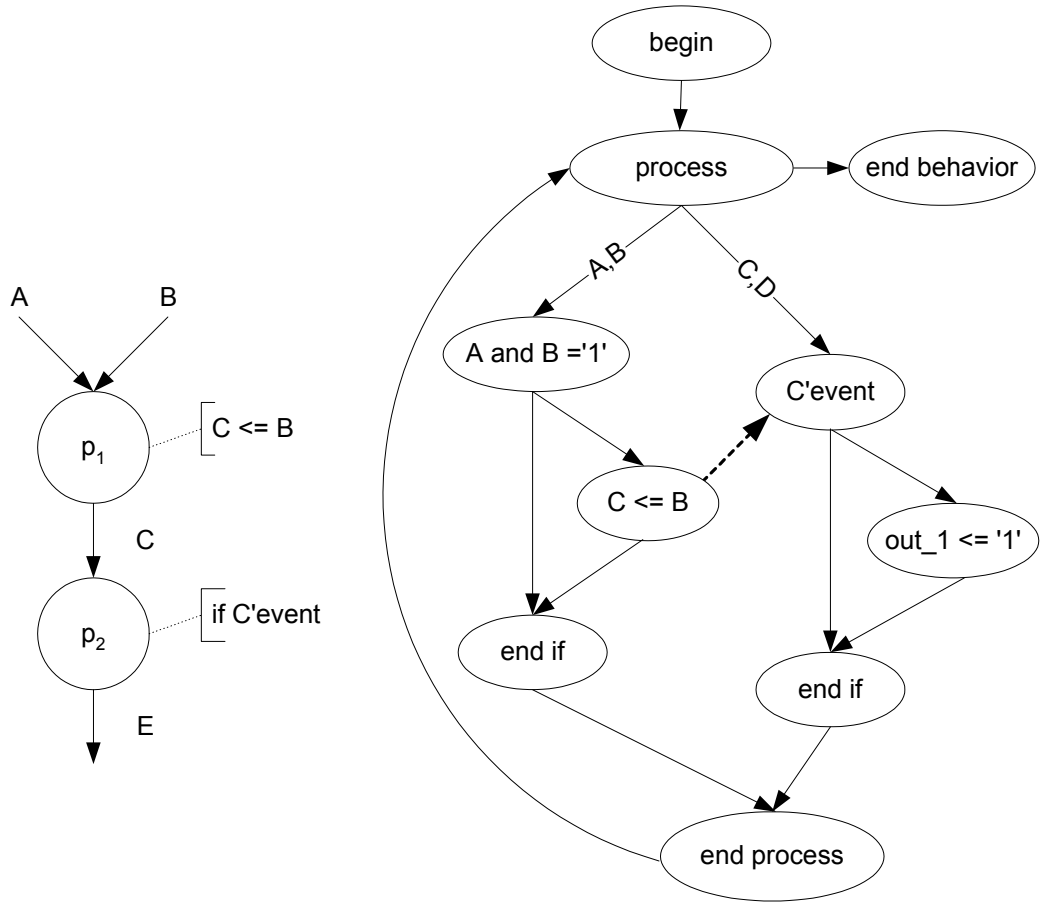


Figure 4-20. Cas d'utilisation du GdD pour un chemin à modifier.

À gauche on a représenté le PMG de la description. Les process p_1 et p_2 sont fortement connectés. L'ensemble des chemins de p_1 constitue la liste des chemins solution. Si on considère maintenant le lien de dépendance (à droite sur le GFC) qui lie le nœud de décision if C'event au nœud d'affectation c <= B, on obtient que seuls les chemins traversant ce nœud affectation sont solutions. Ce qui réduit la liste des chemins solution de deux chemins à un chemin.

La section suivante est consacrée au problème de la génération des données de test à partir de chemins.

4.4 Traduction de notre problème de génération de données vers un CSP

Nous avons vu dans le chapitre 3 que les générateurs de données orientés structure sont basés sur le principe de l'évaluation symbolique. Les chemins du GFC sont représentés par des systèmes de contraintes. Une contrainte est utilisée pour chaque expression associée à un nœud de décision traversée par le chemin.

En ce qui concerne les descriptions VHDL, le problème de la génération de données consiste à trouver les valeurs à appliquer sur les ports d'entrée (stimuli) qui exercent un chemin donné de la base. On choisit pour un chemin donné appartenant à la base, d'exprimer toutes les instructions traversées en terme de contraintes [Paoli 1999c, 2000abd, 2002]. Le résultat de la résolution de ces contraintes représente le domaine des données d'entrée qui permettent de traverser le chemin.

Nous allons donc dans un premier temps rappeler les principes généraux des problèmes de satisfaction de contraintes, et la correspondance pour notre problème. Dans un deuxième temps nous présenterons le modèle de contrainte, défini par Kalynaraman [Kalyanaraman 1993] et Vemuri [Vemuri *et al.* 1995], pour les objets VHDL (signaux et variables) impliqués dans un chemin que nous souhaitons utiliser.

4.4.1 Le problème de satisfaction de contraintes

Un problème de satisfaction de contraintes (CSP en anglais pour *Constraints Solving Problem*) est la donnée :

- d'un ensemble de variables $X = \{x_1, x_2, .. , x_n\}$;
- pour chaque variable x_i, d'un domaine de variation D_i qui est l'ensemble fini des valeurs que x_i peut prendre ;
- d'un ensemble de contraintes $\{C_1, C_2, .. , C_m\}$ portant sur les x_i, chaque C_j décrivant implicitement un sous-ensemble de $D_1 \times D_2 \times .. \times D_n$.

Résoudre un tel problème revient à déterminer les affectations des variables qui satisfont les contraintes, ou, autrement dit, déterminer l'intersection des sous-ensembles de $D_1 \times D_2 \times .. \times D_n$ que décrivent les contraintes. On peut vouloir trouver :

- toutes les solutions ;
- une solution, sans aucune préférence ;
- une solution optimale.

Pour notre part, nous désirons trouver toutes les solutions. En effet nous avons vu dans la section 4.3.2.2 qu'il peut apparaître un conflit de valeur entre signaux internes ou variables impliqués dans le chemin à ordonnancer et sa liste d'ordonnancement. Ceci est dû au fait que

les chemins de la base peuvent, durant leur exécution, traverser plus de code que ce qui est spécifié par le chemin lui même.

Certains CSPs bien délimités peuvent être résolus par des algorithmes spécialisés. Ces algorithmes tirent parti de leur connaissance sur la forme des contraintes ou sur le domaine des variables (CSP numériques, CSP booléens). L'objet de cette étude n'est pas d'étudier ces algorithmes spécialisés.

Les contraintes peuvent être résolues en utilisant un langage de Programmation Logique avec Contraintes (PLC) incluant un résolveur de contraintes. Si tous les domaines des variables sont finis et peuvent être énumérés, alors le résolveur de contraintes trouvera une solution si elle existe. En fait, si le résolveur de contrainte utilise le concept de retour arrière (*backtracking* en anglais) alors toutes les solutions peuvent être obtenues. On dit qu'un système est sous contraint lorsque l'ensemble des contraintes n'est pas suffisant pour définir une solution unique au CSP. Notons que c'est ce qui fait la force d'un environnement à base de contraintes, on ne spécifie que partiellement le problème laissant la tâche de calculer la valeur des variables à l'environnement.

Dans le cadre de la génération de données d'entrée à partir des chemins nous avons deux éléments à modéliser :

- la définition des objets VHDL (variables et signaux) en variables de contrainte, et la définition de leur domaine ;
- la traduction des instructions VHDL en contraintes.

Vemuri et Kalynaraman ont défini une méthode de génération de vecteurs de test pour des programmes VHDL à partir d'un modèle de contraintes dont nous nous inspirons largement. La section suivante est dédiée à la présentation de ce modèle.

4.4.2 Le modèle de contraintes de Vemuri et Kalynaraman

Dans un programme VHDL, les variables et les signaux peuvent être affectés plusieurs fois. Ainsi dans le fragment de code VHDL suivant :

```
variable m,a : bit := 0 ;
variable b : bit := 1 ;
m := a and b;
m := not a;
```

la variable `m` est affectée à deux endroits différents. Sa valeur peut être différente dans les deux cas. En effet si on traduit directement ces instructions en contraintes, on obtient l'ensemble de contraintes suivant :

```
a = 0
b = 1
m = a and b
m = not a
```

Cet ensemble de contraintes n'a pas de solution car les contraintes sont contradictoires. Cette situation provient du fait que les signaux et les variables VHDL sont traités directement comme des variables de contraintes. Les contraintes sont des relations liant entre elles les variables de contrainte qui représentent la valeur des objets VHDL plutôt que les objets VHDL eux-mêmes. Les contraintes sont des formules logiques ; l'ordre dans lequel les contraintes sont écrites n'a pas d'incidence. Si on renomme chaque instance de l'objet VHDL chaque fois qu'il est affecté, il n'y aura aucune conséquence pour les contraintes produites. Le processus de génération de contraintes doit tenir compte des nouvelles incarnations des objets affectés. Un objet VHDL prend sa première incarnation (sa première valeur) lors de sa déclaration et de son initialisation. Chaque fois que l'objet VHDL est affecté, une nouvelle incarnation est crée représentant sa nouvelle valeur. Chaque incarnation doit correspondre à une variable de contrainte unique.

Vemuri et Kalynaraman ont défini l'idée de l'incarnation spatiale d'un objet VHDL. Chaque objet VHDL possède plusieurs incarnations spatiales. Quand l'objet VHDL est déclaré, le nombre d'incarnation spatiale est initialisé à zéro. Chaque fois que l'objet VHDL est affecté le nombre d'incarnation spatiale est incrémenté.

Si on reprend l'exemple précédent, on obtient l'ensemble de contraintes suivant :

$a_0 = 0$
$b_0 = 1$
$m_0 = a_0$ and b_0
$m_1 = \text{not } a_0$

où le nombre d'incarnation spatiale est représenté par un indice.

Pour les signaux, un nombre d'incarnation supplémentaire est nécessaire. En effet si on considère l'instruction d'affectation de signal suivante :

```
signal m : bit :='0' ;
m <= '1' ;
```

la valeur '1' sera affectée au signal m au cycle de simulation suivant, d'où la nécessité d'un nombre d'incarnation temporelle pour les signaux. Si on reprend l'exemple précédent, on obtient l'ensemble de contraintes suivant :

$$m_0^0 = 0$$
$$m_1^0 = m_0^0$$
$$m_1^1 = 1$$

où le nombre d'incarnation temporelle est noté en exposant. L'incarnation temporelle correspond au cycle de simulation (cf. section 4.1.2.3). Le générateur de contrainte devra prendre en compte l'incrémentation de ce cycle de simulation, chaque fois qu'un *process* est traversé une nouvelle fois par un chemin.

4.5 Conclusion

Dans cette partie, nous avons présenté un modèle interne basé sur des structures graphiques pour représenter le flot de contrôle des programmes VHDL. Cette représentation graphique est nécessaire pour appliquer les techniques de test de logiciels que nous avons choisi. Nous avons vu en détail la correspondance entre les instructions de programme VHDL et leur représentation graphique en tant qu'éléments du GFC. Nous avons détaillé pour chaque instruction leur structure en terme de nœud et d'arc.

À partir de ce modèle interne, nous avons également défini la notion de chemin d'exécution ainsi que les notions de chemin à modifier, de liste de chemins solution, de chemin à ordonnancer et de liste d'ordonnancement. Ces chemins de la base requièrent une étape de traitement supplémentaire. Afin de résoudre les problèmes liés à ces chemins, on s'appuie sur deux nouvelles structures de type graphe : le Graphe de Modélisation de Process (PMG) défini par Cho et Armstrong qui modélise la connectivité entre les *process*, et le Graphe de Dépendance (GdD) qui modélise l'interaction entre les instructions d'affectation et les instructions de contrôle où elles sont utilisées.

Concernant le problème de génération de données d'entrée à partir d'un chemin, nous avons choisi de traduire notre problème vers un CSP en utilisant le modèle de contrainte défini par Vemuri et Kalynaraman. Nous avons vu que ces contraintes peuvent être résolues en utilisant un langage de PLC incluant un résolveur de contraintes.

À partir des structures et des concepts définis dans ce chapitre, nous pouvons maintenant présenter notre approche pour la génération de vecteurs de test. On construit d'abord le GFC de la description VHDL auquel nous allons appliquer l'algorithme de Poole. La base ainsi générée sera analysée, grâce au PMG et au GdD, afin de détecter les chemins qui demandent un traitement supplémentaire. Les chemins à ordonnancer réclament une liste d'ordonnancement pour former la séquence de test associée et les chemins à modifier nécessitent qu'on les combine avec un autre chemin appartenant à la liste des chemins solution. Une fois cette analyse accomplie, les chemins sont traduits en un CSP en utilisant un langage de PLC incluant un résolveur de contraintes. Le résolveur génère alors les stimuli à appliquer sur les ports d'entrée.

Les différents algorithmes, qui permettent notamment l'analyse des chemins, leur modification, leur ordonnancement sont présentés en détail dans le chapitre suivant. Il contient également les algorithmes nécessaires à la génération des contraintes à partir des chemins et à l'extraction des stimuli.

CHAPITRE

5

LA GENERATION DES VECTEURS DE TEST

Dans le chapitre précédent nous avons défini un Graphe de Flot de Contrôle (GFC) adapté au description VHDL [Paoli 1998ab, 1999abc]. À partir de ce graphe nous avons constaté que l'algorithme de Poole génère une base de chemins qui requiert une analyse. Nous avons identifié deux types de chemins particuliers qui demande un traitement supplémentaire : les chemins à ordonnancer qui nécessitent une liste d'ordonnancement et les chemins à modifier qui réclament qu'on les combine avec un autre chemin de la base à partir de la liste des chemins solution. Nous avons établi que ce traitement peut être réalisé grâce au Graphe de Modélisation de Process (PMG) et à un Graphe de Dépendance (GdD) [Paoli 1999c, 2000abd].

Nous avons également présenté le problème de génération de données d'entrée (stimuli) comme un problème de satisfaction de contraintes. Nous avons montré le modèle de contrainte de Vemuri et Kalynaraman que nous désirons utilisé pour notre approche de

génération de contraintes. Nous avons vu que ces contraintes peuvent être résolues en utilisant un langage de PLC incluant un résolveur [Paoli 2000abd, 2002].

Dans ce chapitre, nous allons détailler étape par étape notre méthodologie pour la génération des séquences de test. La première partie présente une vue d'ensemble de notre méthodologie séparée en quatre phases distinctes. Les différents algorithmes nécessaires à la réalisation de ces quatre étapes sont décrits dans les parties suivantes illustrées par de nombreux exemples pédagogiques.

5.1 Vue d'ensemble

La Figure 5-1 illustre notre approche dont le point de départ est un programme VHDL de type comportemental et dont le résultat est un ensemble de vecteurs de test. Ces séquences de test traversent l'ensemble de base des chemins test respectant ainsi le critère du test structuré. Quatre phases permettent d'obtenir ce résultat :

i. *Production de la base de chemins :*

La première phase consiste en la construction du GFC modélisant le flot de contrôle du programme VHDL sous test et en la production de la base de chemins indépendants assurée par l'application de l'algorithme de Poole.

ii. *Analyse des chemins :*

La seconde phase concerne l'analyse des chemins. En effet, nous avons vu dans le chapitre précédent que les chemins à modifier ne peuvent pas être traduits directement en terme de contraintes. Cette étape consiste donc en leur détection et en leur modification.

iii. *Extraction des vecteurs de test :*

La troisième phase est consacrée à la génération des contraintes correspondant aux instructions traversées par chaque chemin de la base modifiée à l'étape précédente et en leur résolution afin d'extraire les stimuli qui formeront les vecteurs de test. À chaque chemin correspond un vecteur de test.

iv. *Production des séquences de test :*

La dernière phase concerne les chemins à ordonnancer. Ces chemins nécessitent qu'une séquence de test soit exécutée avant le vecteur de test correspondant au chemin. Cette phase consiste donc en leur détection et en la génération de cette séquence.

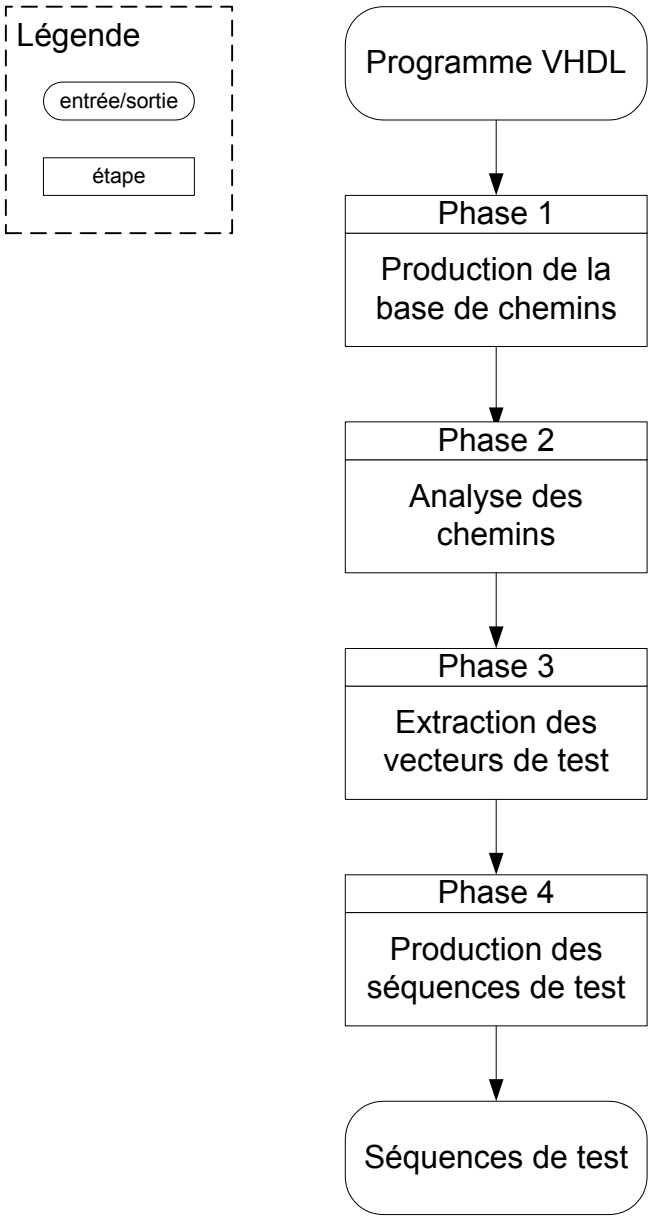


Figure 5-1. Méthodologie pour la génération des séquences de test.

Sur la Figure 5-1 nous n'avons pas représenté les interactions possibles entre les différentes phases par souci de clarté. Néanmoins ces interactions sont présentées explicitement lorsque nous abordons chacune de ces étapes.

5.2 Production de la base de chemins

La Figure 5-2 illustre la première phase qui consiste en la génération de la base de chemins. Le programme VHDL est analysé par un lexeur-parseur[3] qui donne en sortie une

[3] Le principe de fonctionnement d'un lexeur-parseur est détaillé dans la partie du chapitre suivant consacrée à l'implémentation.

représentation intermédiaire du programme sous la forme d'un arbre syntaxique abstrait (AST). C'est à partir de cet AST que l'on construit le GFC ainsi que le GdD et le PMG (graphes nécessaires dans les sections suivantes).

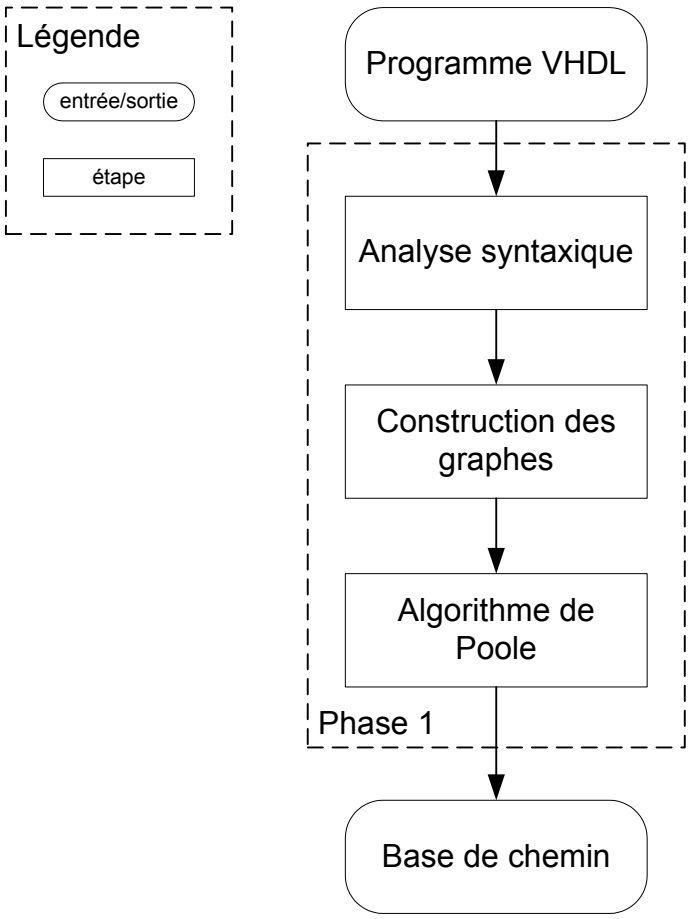


Figure 5-2. La production de la base.

Une fois le GFC construit, on applique l'algorithme de Poole (cf. chapitre 3). Le résultat de cette étape est la production d'une base de chemins indépendants. Cette base est l'entrée de la deuxième phase : l'analyse des chemins.

5.3 Analyse des chemins

Cette étape concerne uniquement les programmes VHDL contenant plusieurs instructions *process*. En effet dans le chapitre précédent nous avons vu que lorsqu'une description VHDL est composée de plusieurs *process* interconnectés par des signaux internes, certains chemins ne commencent pas au point d'entrée du programme. Ces chemins sont appelés **chemins à modifier** car ils doivent être combinés avec un chemin commençant au

point d'entrée du programme pour que l'on puisse les traduire en terme de contraintes exploitables pour la génération des données d'entrée.

La Figure 5-3 illustre cette phase. À partir de la base de chemins, on détecte les chemins à modifier. Pour chacun des chemins, on génère la liste des chemins solution. En effet, il est possible que plusieurs chemins commençant au point d'entrée soient solution. On choisit de conserver ces possibilités sous la forme d'une liste de chemin appelée liste des chemins solution. On propose ensuite une combinaison de chemin que l'on insère dans une nouvelle base. Évidemment, les chemins de la base initiale qui ne sont pas des chemins à modifier sont ajoutés à la nouvelle base sans aucune modification.

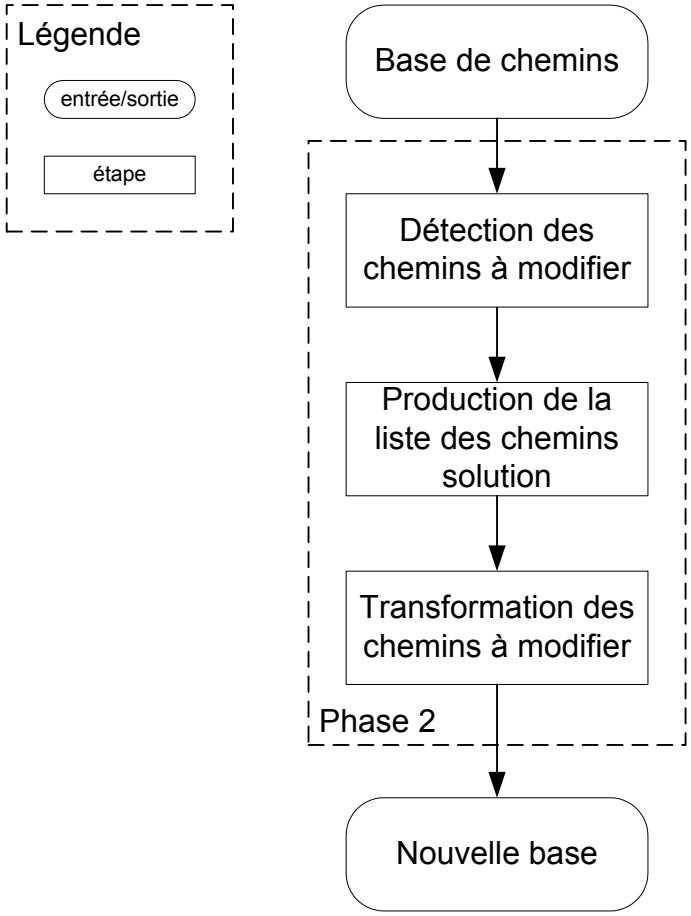


Figure 5-3. Analyse des chemins.

L'algorithme général en pseudo-langage, de la phase d'analyse des chemins, est le suivant, où B représente la base initiale composée des chemins C_i avec $0 < i \leq v(G)$, où C_i' est

le chemin modifié, où LS est la liste des chemins solution C_j et où B' représente la nouvelle base qui est initialement vide :

```
B' est vide
Pour tous les chemins Ci de B faire
      Si Ci est un chemin à modifier alors
            Produire la liste solution LS de Ci
            Répéter
                  Choisir un chemin Cj dans LS
                  Ôter Cj de LS
                  Ci' est une combinaison de Ci et de Cj
                  Générer et résoudre les contraintes sur Ci'
            Jusqu'à solution ou LS est vide
            Si LS est vide
                  Ci est un chemin infaisable
            Sinon
                  Ajouter Ci' à la base B'
            Fin Si
      Sinon
            Ajouter Ci à la base B'
      Fin Si
Fin pour
```

Nous allons détailler chacune des actions de cet algorithme. Pour tous les chemins de la base B, on désire détecter les chemins à modifier. Au départ B' est vide, puis la condition `Si Ci est un chemin à modifier` est évaluée. L'évaluation de cette condition est réalisée par une fonction dont l'algorithme est illustré par l'organigramme de la Figure 5-4.

Sur ce diagramme, les cercles représentent les entrées et les sorties de la fonction, les rectangles représentent les actions, et les losanges représentent les décisions (ces définitions sont valables pour tous les diagrammes présentés dans ce chapitre).

Trois entrées sont nécessaires : le chemin de la base à tester, le GFC et le PMG du programme. Par souci de clarté, le chemin est vu comme une suite de nœud, ce qui diffère de la définition d'un chemin utilisée dans les chapitres précédents.

Cette fonction permet de distinguer si un chemin de la base est un chemin à modifier ou non en renvoyant respectivement la valeur booléenne VRAI ou FAUX. Elle utilise la définition établie dans le chapitre 4 : si un chemin de la base appartenant à un *process* fortement connecté nécessite un événement sur un signal interne alors le chemin est un chemin à modifier, sinon le chemin n'est pas à modifier.

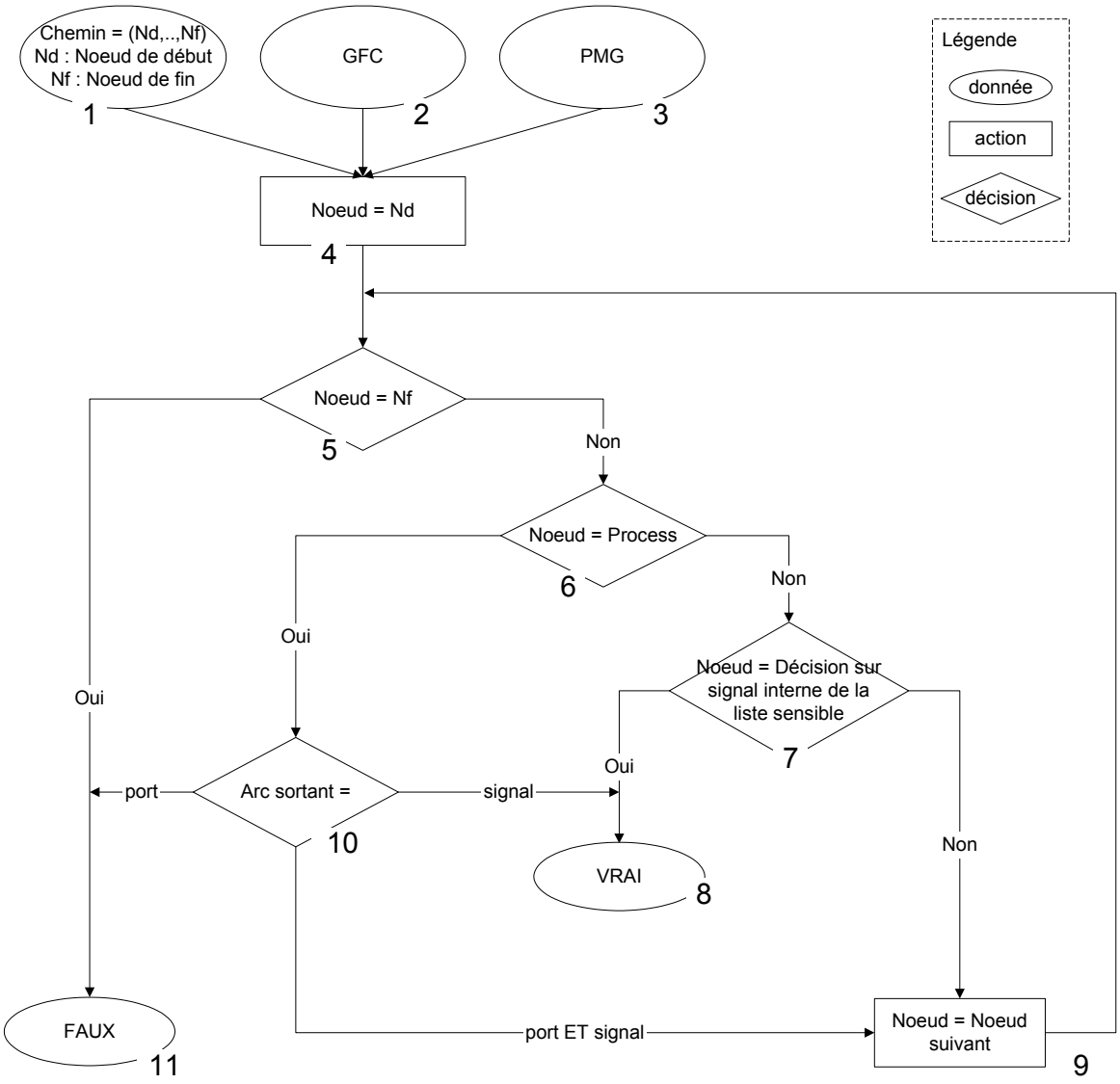


Figure 5-4. Organigramme pour la détection des chemins à modifier.

Le chemin de la base est parcouru du nœud de début (noté Nd) au nœud de fin (noté Nf). Si le nœud courant est un nœud *process* (décision 6 sur le diagramme), on observe sa liste sensible (modélisée sur le GFC par l'arc sortant du nœud *process*). Si la liste sensible ne contient que des ports (sortie gauche de la décision 10 sur le diagramme) le chemin n'est pas à modifier. En effet le chemin commence à un point d'entrée du programme, un événement sur l'un des ports permet d'exécuter ce chemin. Dans ce cas, la fonction renvoie la valeur FAUX (donnée 11 sur le diagramme) à l'algorithme général. Le chemin n'est pas un chemin à modifier, il est ajouté à la nouvelle base B' sans aucune modification.

Par contre, si la liste sensible ne contient que des signaux internes (sortie droite de la décision 10 sur le diagramme) c'est obligatoirement un chemin à modifier. En effet, le chemin

ne commence pas à un point d'entrée du programme, on ne peut pas directement déclencher un événement sur un signal interne. Dans ce cas, la fonction renvoie la valeur VRAI (donnée 8 sur le diagramme) à l'algorithme général.

Dans le dernier cas, si la liste sensible est composée à la fois de ports et de signaux internes (sortie du bas de la décision 10 sur le diagramme) il faut parcourir les nœuds restants du chemin. Si le chemin traverse un nœud de décision impliquant un signal interne appartenant à la liste sensible (décision 7 sur le diagramme), c'est un chemin à modifier. Dans ce cas la fonction renvoie la valeur VRAI. Si on arrive au nœud de fin (décision 5 sur le diagramme) sans traverser une telle décision, le chemin n'est pas un chemin à modifier. Dans ce cas la fonction renvoie la valeur FAUX (donnée 11 sur le digramme) à l'algorithme général. Le chemin n'est pas un chemin à modifier, il est ajouté à la nouvelle base B' sans aucune modification.

Dans le cas où un chemin est à modifier, il faut chercher LS, la liste des chemins solution. L'action `Produire la liste solution LS de` C_i est réalisée par une procédure dont l'algorithme est illustré par l'organigramme de la Figure 5-5. Cette procédure permet de produire LS pour un chemin à modifier. Trois entrées sont nécessaires : le chemin de la base à modifier, le GFC et le GdD du programme.

Au départ, la liste LS est vide (action 4 sur le diagramme). Afin de générer LS, on parcourt le chemin à modifier du nœud de début (noté Nd) au nœud de fin (noté Nf). Si un nœud de décision implique un signal interne appartenant à la liste sensible du *process* traversé par le chemin à modifier (décision 7 sur le diagramme), alors on conserve les chemins traversant le nœud lié par un arc de dépendance au nœud de décision (action 8 sur le diagramme) dans la liste LS. À chaque fois que l'on rencontre ce type de nœud on ajoutera à LS (action 9 sur le diagramme) les chemins correspondants. À la fin du chemin (condition 6 sur le diagramme), la procédure se termine. LS est la liste des chemins solutions.

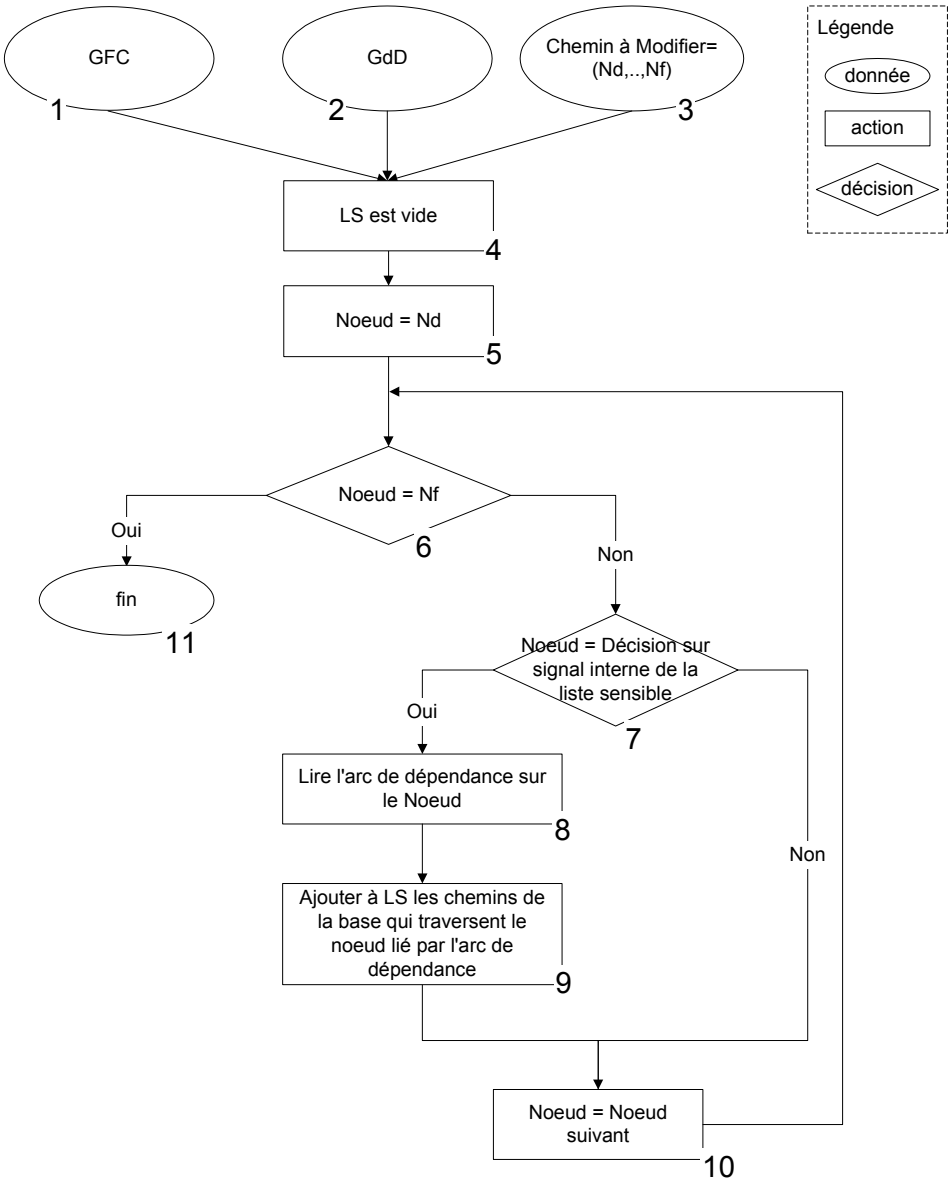


Figure 5-5. Génération de la liste des chemins

Si on revient à l'algorithme général, on doit maintenant combiner le chemin à modifier avec un chemin de la liste des chemins solution. Cependant, il se peut qu'au cours de la phase suivante (Extraction des vecteurs de test) aucune solution ne soit trouvée par le résolveur de contrainte, le chemin dans ce cas est infaisable. Dans ce cas, on doit choisir la combinaison de chemins suivante à partir de la liste des chemins solution, d'où la présence de la boucle : `Répéter Jusqu'à solution ou LS est vide`.

La première action dans la boucle est de choisir arbitrairement un chemin C_j parmi les chemins de LS puis on le combine avec C_i. Le nouveau chemin C_i' est formé par la

combinaison de C_j sans son dernier arc et de C_i sans son premier arc. L'action `Générer et résoudre les contraintes sur Ci'` est l'objet de la section suivante. Si cette action échoue (`solution` égale à FAUX) le chemin est infaisable. Un nouveau chemin C_j est choisie dans LS et le processus est répété jusqu'à ce que l'action `Générer et résoudre les contraintes sur Ci'` réussisse (`solution` égale à VRAI). Si l'action échoue et que la liste LS est vide, on est en présence d'un chemin infaisable. Sinon on a un chemin faisable. Dans ce cas C_i' est ajouté à la nouvelle base B'.

5.4 Extraction des vecteurs de test

La troisième phase est consacrée à l'extraction des vecteurs de test. La Figure 5-6 illustre cette étape qui consiste en la génération des contraintes et en leur résolution. L'idée est de traduire chaque chemin en un ensemble de contraintes, le résultat de la résolution de cet ensemble formant les vecteurs de test qui permettent d'exécuter les chemins.

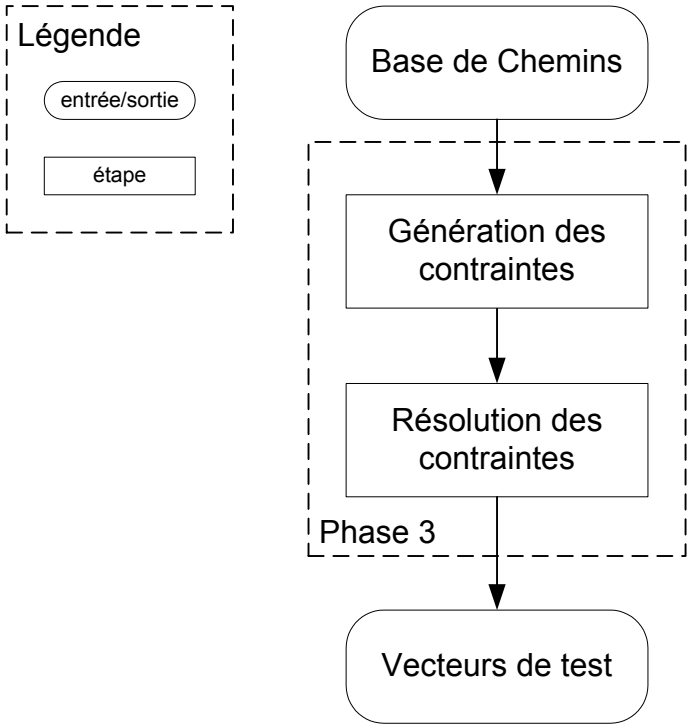


Figure 5-6. L'extraction des vecteurs de test.

L'entrée de cette phase est une base de chemins. Notons que si la description VHDL sous test comporte un seul *process*, cette base de chemins correspond à la base initiale issue

de la phase 1 (cf. section 5.2). Dans le cas d'une description VHDL avec plusieurs *process*, cette base correspond à la nouvelle base résultat de la phase 2 (cf. section 5.3).

5.4.1 La génération des contraintes

La première étape est de produire des variables de contraintes correspondant aux objets VHDL (signal et variable) du programme et les contraintes sur ces variables. On considère deux types de contraintes. Les contraintes de domaine spécifient le domaine à partir duquel les valeurs d'une variable de contrainte sont générées. Les contraintes relationnelles sont des égalités, des inégalités à travers les variables de contrainte. Elles correspondent aux diverses instructions d'affectation et autres instructions de contrôle présentes dans un programme VHDL.

5.4.1.1 Contraintes de domaine

Les contraintes de domaine sont générées à partir des déclarations des objets VHDL. On distingue les ports, les signaux internes, les variables et les constantes.

a. Contraintes de domaine sur les ports :

On trouve les informations nécessaires à la définition des contraintes de domaine des ports dans la partie déclarative de l'instruction entité. Par exemple pour l'entité `example` suivante :

```
entity example is
port ( in_1,in_2: in bit;
          out_1: out bit);
end example;
```

on génère les contraintes suivantes :

```
écrire « in_1 appartient au domaine fini [0,1] »
écrire « in_2 appartient au domaine fini [0,1] »
```

On ne génère pas de contraintes pour les ports de sortie car ces objets VHDL n'influencent pas le flot de contrôle du programme.

b. Contraintes de domaine sur les signaux internes :

On trouve les informations nécessaires à la définition des contraintes de domaine des signaux internes dans la partie déclarative de l'architecture.

Par exemple pour l'architecture suivante :

```
architecture behavior of example is
signal a:bit;
begin
```

on génère la contrainte suivante :

```
écrire « a appartient au domaine fini [0,1] »
```

c. Contraintes de domaine sur les variables :

On trouve les informations nécessaires à la définition des contraintes de domaine des variables dans le corps des *process*. Par exemple pour le *process* suivant :

```
process1: process(in_1)
variable b:bit;
begin
```

on génère la contrainte suivante:

```
écrire « b appartient au domaine fini [0,1] »
```

d. Contraintes de domaine sur les constantes :

On trouve les informations nécessaires à la définition des contraintes de domaine des constantes n'importe où on peut déclarer des objets. On peut les trouver aussi bien dans l'entité que dans l'architecture que dans un *process*. Par exemple pour la déclaration suivante :

```
Constant st0:integer:=1;
```

on génère les contraintes suivantes :

```
écrire « st0 appartient au domaine des entiers »
écrire « st0 = 1 »
```

5.4.1.2 Contraintes relationnelles

Les contraintes relationnelles dérivent des instructions d'affectation et des instructions de contrôle correspondant aux nœuds parcourus par un chemin sur le GFC du programme VHDL.

a. Les instructions d'affectation de variables :

Les variables peuvent être affectées plusieurs fois pendant l'exécution d'un programme VHDL. Nous avons vu qu'il est nécessaire de définir pour les variables une incarnation spatiale. À chaque variable V, on associe un nombre d'incarnation spatiale noté

s(V). Quand la variable est déclarée, son nombre d'incarnation spatiale est initialisé à '0'. Chaque fois qu'une instruction d'affectation de cette variable est traversée par le chemin, le nombre d'incarnation spatiale est incrémenté. Pour une instruction d'affectation de variable de la forme :

```
V := Expression
```

on génère les contraintes avec l'algorithme suivant :

```
s(V) = s(V) + 1
V_s(V) appartient au domaine de V
écrire « V_s(v) = Expression »
```

où les composants de `Expression` (variable ou signal) prendront la valeur de leur dernière incarnation. Il est évident que chaque incarnation de variable héritera de la contrainte de domaine du type correspondant.

b. Les instructions d'affectation de signaux :

Contrairement aux variables dont l'affectation est immédiate, l'affectation des signaux est retardée par des retards delta. Nous avons vu qu'en plus de l'incarnation spatiale il est nécessaire de définir pour les signaux une incarnation temporelle. À chaque signal U on associe en plus de son incarnation spatiale s(U), un nombre d'incarnation temporelle noté d(U) qui représente la valeur courante du retard delta associé au signal U.

Chaque fois qu'une instruction d'affectation de signal est traversée par le chemin, le nombre d'incarnation spatiale est incrémenté. Le signal possède alors plusieurs incarnations spatiales noté $U_{s(U)}$ qui possèdent plusieurs incarnations temporelles noté $U^{d}_{s(U)}$ où d représente la valeur du delta associée au signal U, avec $0 \leq d \leq d(U)$. Où d(U) est la dernière incarnation temporelle du signal U.

À chaque instruction d'affectation rencontrée, une nouvelle incarnation spatiale est crée. Cette incarnation spatiale hérite de toutes les incarnations temporelles du signal jusqu'au delta courant. Par contre, cette incarnation spatiale possède une nouvelle incarnation temporelle correspondant à l'affectation du signal. Ainsi, pour une instruction d'affectation de signal de la forme :

```
U <= Expression
```

on génère les contraintes avec l'algorithme suivant :

```
s(U) = s(U) + 1
écrire « U^{d(U)}_{s(U)} appartient au domaine de U »
      pour 0 ≤ d ≤ d(U)
            écrire « U^{d}_{S(U)} = U^{d}_{s(U)-1} »
      fin pour
écrire « U^{d(U)+1}_{s(U)} = Expression »
d(U)= d(U)+1
```

où les composants de `Expression` (variable ou signal) prendront la valeur de leur dernière incarnation. Le delta associé au signal est mis à jour à la fin. Évidemment, chaque incarnation du signal hérite de la contrainte de domaine correspondant au type du signal. De plus, on peut remarquer que seul les signaux internes et les ports de sortie peuvent être affectés.

Toutefois on ne s'occupera pas des ports de sortie car notre but est de trouver les valeurs à appliquer en entrée de la description VHDL. Concernant les ports d'entrée, ils ne possèderont qu'une seule incarnation spatiale puisqu'ils ne peuvent être affectés.

c. Les instructions *process* et *end process* :

Nous avons vu qu'une liste de signaux sensibles est associée à chaque instruction *process*. Les instructions présentes dans le corps du *process* ne s'exécutent que si un événement se produit sur au moins un des signaux de la liste.

Si un chemin traverse un *process* : P(a,b,c) où a, b, c sont les signaux sensibles de P, la condition suivante doit être vraie : événement sur a ou événement sur b ou événement sur c. Le modèle de contraintes défini par Vemuri et Kalynaraman, présenté dans le chapitre précédent, permet de définir cette décision sous forme de contraintes à travers l'incarnation temporelle.

Suivant que le signal de la liste sensible est un port d'entrée ou un signal interne, l'incarnation temporelle n'a pas la même signification. En effet, pour un port d'entrée l'incarnation temporelle représente le temps de simulation courant associé au simulateur. Pour un signal interne, l'incarnation temporelle représente un retard delta puisque l'affectation d'un signal est retardée par un delta. Contrairement à un signal interne, un port d'entrée ne peut être affecté à l'intérieur de la description VHDL. En conséquence, pour une incarnation temporelle de port d'entrée, on peut avoir plusieurs incarnations temporelles de signal interne.

On définit l'incarnation temporelle T, comme la représentation du temps d'un cycle de simulation associé à un port d'entrée alors que l'incarnation temporelle d(A) représente le delta associé à un signal interne A. De plus, on a besoin d'une variable qui représente la valeur courante du cycle delta. Cette variable est incrémentée à chaque cycle de simulation impliquant un évènement sur un signal sensible. On l'appellera Delta.

Pour tous les signaux (signal interne ou port d'entrée) de la liste sensible du *process* on génère les contraintes ci-dessous suivant la liste associée à l'arc de sortie du nœud process considéré. Auparavant, on aura incrémenté l'incarnation temporelle T : T = T +1, puisqu'un cycle de simulation temps (représenté par l'arc liant le nœud de début au nœud process) commence. On aura préalablement initialisé l'incarnation temporelle T et la variable Delta à zéro ainsi que tous les d(A) des signaux internes associés. Pour chaque `process` de la forme :

```
process(P, A)
```

où P représente un port d'entrée et A un signal interne de la liste des signaux sensibles associée on génère les contraintes avec l'algorithme suivant :

```
Pour tous les ports et les signaux
      écrire « P^T_s(P) ≠ P^T-1_s(P) ou A^d(A)_s(A) ≠ A^d(A)-1_s(A) »
fin pour
```

Lorsqu'on rencontre un nœud de fin de process, il faut mettre à jour les signaux internes qui n'ont pas été affectés ainsi que leur incarnation temporelle d(A). Ainsi pour chaque instruction de la forme :

```
end process
```

on génère les contraintes avec l'algorithme suivant :

```
Delta = Delta +1
Pour tous les signaux internes
      Si d(A)<Delta
            écrire « A^d(A)+1_s(A) = A^d(A)_s(A) »
            d(A) = d(A) +1
      fin si
fin pour
```

d. Les structures de contrôle :

D'après notre sous-ensemble VHDL, on considère trois types de structure de contrôle : IF, CASE, et LOOP. Pour chaque instruction IF de la forme :

```
IF (Boolean_Expression)
```

on génère les contraintes ci-dessous suivant que l'arc de sortie du nœud de décision `IF` est VRAI ou FAUX :

```
Boolean_Expression = Vrai
Boolean_Expression = Faux
```

Pour chaque instruction CASE de la forme :

```
CASE (Expression) IS WHEN Choices =>
```

suivant le `Choice` sur l'arc de sortie du nœud de décision `CASE`, on génère la contrainte ci-dessous :

```
Expression = Choice
```

Un `Choice` est une expression simple composée de variable, signal, port d'entrée avec des opérateurs d'addition tel que : + - &. Si un `Choice` est un `OTHERS` la contrainte suivante est générée :

```
((Expression = Choice_1) ou (Expression = Choice_2) .. ) = Faux
```

Selon notre sous ensemble VHDL on peut avoir deux structures itératives. Pour chaque instruction LOOP de la forme :

```
WHILE Boolean_Expression
```

on génère les contraintes ci-dessous suivant que l'arc de sortie du nœud de décision `WHILE` est VRAI ou FAUX :

```
Boolean_Expression = Vrai
Boolean_Expression = Faux
```

Pour chaque instruction LOOP de la forme :

```
FOR Indentifier IN Discrete_Range
```

on génère les contraintes ci-dessous suivant que l'arc de sortie du nœud de décision FOR est VRAI ou FAUX :

```
Indentifier IN Discrete_Range = Vrai
Indentifier IN Discrete_Range = Faux
```

e. Les expressions booléennes :

Les ports d'entrée, les signaux internes et les variables impliqués dans une expression booléenne : `Boolean_Expression` seront remplacés par la dernière incarnation de leur variable de contrainte respective.

f. Les attributs

D'après notre sous ensemble VHDL, un seul attribut est autorisé. Il s'agit de l'attribut : `'event`. Pour un signal A, `A'event` renvoie VRAI si un événement s'est produit sur A pendant le cycle de simulation courant. Un cycle de simulation peut être un cycle delta ou un cycle de temps. Chaque occurrence de cet attribut dans une expression booléenne impliquera les contraintes suivantes :

$A^{T}_{s(A)} \neq A^{T-1}_{s(A)}$ pour les ports d'entrée »

$A^{d(A)}_{s(A)} \neq A^{d(A)-1}_{s(A)}$ pour les signaux internes

5.4.1.3 Exemple de génération de contraintes

Pour illustrer la génération des contraintes, nous présentons un exemple complet. On considère que le code VHDL sur la gauche de la Figure 5-7, doit être traversé par un chemin qui traverse les deux *process*.

Les contraintes de domaine suivantes sont produites :

in_1, in_2, a appartiennent au domaine fini [0,1]

Les contraintes relationnelles générées grâce aux algorithmes précédents sont sur la droite de la Figure 5-7 dans un format intermédiaire.

```
entity essai is
port ( in_1,in_2 :in bit;
       out_1 :out bit);
end essai;
architecture behavior of essai is
signal a,b:bit;
Process(in_1,in_2)
         begin
         a <= in_1 and in_2;
end process
Process(a)
         begin
         out_1 <= not a;
end process
```

T=1, Delta=0, d(a)=0, s(a)=0

$in\_1^1_0 \neq in\_1^0_0$ ou $in\_2^1_0 \neq in\_2^0_0$

s(a)=1

$a_1^0 = a_0^0$

$a_1^1 = in\_1_1^1$ et $in\_2_0^1$

d(a)=1

Delta=1

$a_1^1 \neq a_1^0$

Delta=2

Figure 5-7. Exemple de génération de contraintes.

5.4.2 Résolution des contraintes

La résolution de contraintes est réalisée grâce à un résolveur de contraintes. Ce programme calcule à partir d'un ensemble de contraintes une solution (si elle existe) satisfaisant toutes les contraintes. Ainsi, à chaque chemin correspond un ensemble de contraintes de domaine et de contraintes relationnelles. À la fin de l'étape de résolution, trois types de valeur sont à considérer :

- les valeurs des différentes incarnations temporelles des ports d'entrée qui formeront les vecteurs de test. Ce vecteur appliqué en entrée de la description VHDL permettra de traverser le chemin considéré ;
- les valeurs des différentes incarnations temporelles des signaux internes qui sont nécessaires à la production de la séquence de test dans le cas des chemins à ordonnancer (voir section suivante) ;
- les valeurs des différentes incarnations spatiales des variables qui sont nécessaires à la production de la séquence de test dans le cas des chemins à ordonnancer (voir section suivante).

Ces valeurs forment les domaines d'entrée des chemins de la base. Chaque objet VHDL possède un domaine d'entrée de valeurs possible. Les vecteurs de test qui permettent de traverser chaque chemin sont formés à partir des valeurs des domaines des ports d'entrée.

Concernant les chemins à ordonnancer un traitement supplémentaire est à réaliser. Cette phase est l'objet de la section suivante.

5.5 Production de la séquence de test

Cette phase concerne uniquement les chemins à ordonnancer appartenant à la base de chemins. Les chemins à ordonnancer nécessitent qu'une séquence de test soit exécutée avant le vecteur de test généré lors de la phase précédente (section 5.4). Cette séquence de test est construite à partir d'une liste d'ordonnancement. Cette liste est composée de chemins qui doivent être traversés avant le chemin à ordonnancer. Ces chemins permettent l'affectation à la bonne valeur des objets VHDL (signaux internes et/ou variables) impliqués dans le flot de contrôle (nœuds de décision) du chemin à ordonnancer.

Nous avons vu dans le chapitre précédent que la séquence de test à exécuter avant le vecteur de test du chemin à ordonnancer doit être formée à partir d'une des permutations de la liste d'ordonnancement. On considère les permutations de la liste d'ordonnancement car à chaque sous-liste correspondent des vecteurs de test et on ne sait pas dans quel ordre ils devront être exécutés. Cette phase permet de produire une séquence de test valable.

La Figure 5-8 illustre notre démarche : on détecte les chemins à ordonnancer à partir de la base, puis pour chaque chemin à ordonnancer, on produit la liste d'ordonnancement. Enfin pour chaque chemin à ordonnancer, on extrait une séquence de test valable permettant de traverser le chemin à partir de la liste d'ordonnancement et des vecteurs de test résultat de la phase 3.

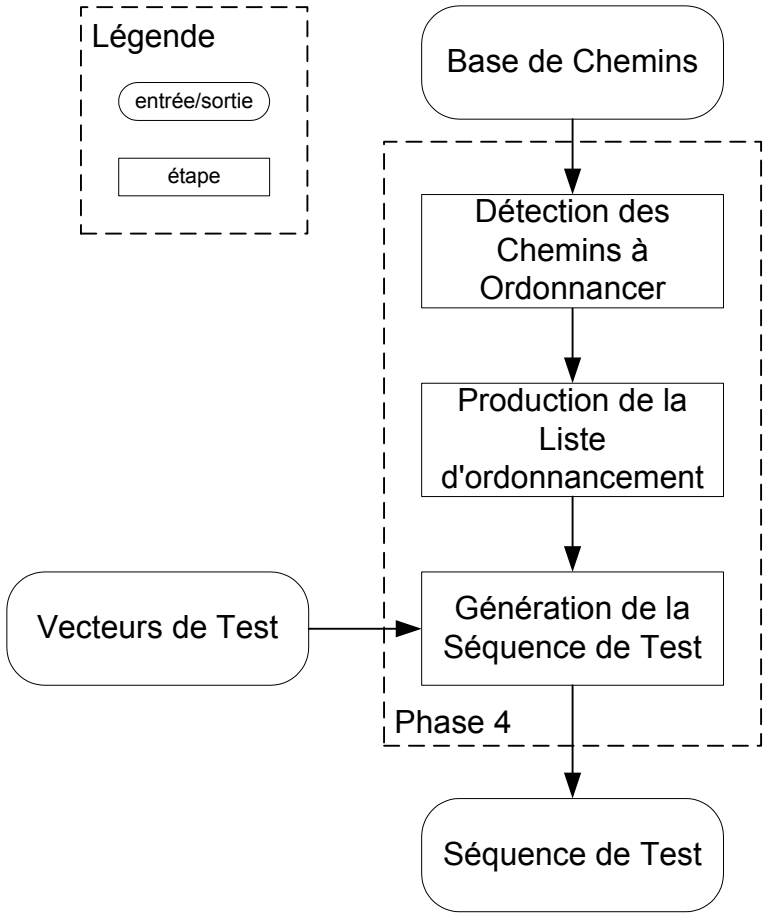


Figure 5-8. Production de la séquence de test.

L'entrée ce cette phase est une base de chemins. Si la description VHDL sous test comporte un seul *process*, cette base correspond à la base initiale issue de la phase 1. Dans le

cas d'une description VHDL avec plusieurs *process*, cette base correspond à la nouvelle base résultat de la phase 2.

L'algorithme général en pseudo-langage, de la phase de production de la séquence de test est le suivant, où B représente la base composée des chemins C_i avec $0 < i \leq v(G)$, où LO est la liste d'ordonnancement et où P est l'ensemble des permutations P_n de LO :

```
Pour tous les chemins Ci de B Faire
      Si Ci = chemin à ordonnancer alors
            Produire la liste P des permutations à partir de LO
            Répéter
                  Choisir une permutation Pn dans P
                  Ôter Pn de P
                  Lancer la méthode générer et tester
            Jusqu'à (solution ou P est vide)
            Si P est vide alors
                  On a un chemin infaisable
            Sinon
                  On a une séquence de test
            Fin si
      Fin si
Fin pour
```

Nous allons détailler chacune des actions de cet algorithme. Pour tous les chemins de la base B, on désir détecter les chemins à ordonnancer. Si la condition `Si Ci = chemin à ordonnancer` est évaluée à VRAI, on doit produire LO la liste d'ordonnancement associée au chemin à ordonnancer. L'évaluation de cette condition est réalisée par une fonction dont l'algorithme est illustré par l'organigramme de la Figure 5-9.

Cet algorithme détecte à partir de la base, les chemins à ordonnancer et renvoie la liste d'ordonnancement correspondante. Trois entrées sont nécessaires : le chemin de la base à tester, le GFC et le GdD du programme VHDL.

Au départ la liste LO est vide (action 4). Chaque chemin de la base est parcouru du nœud de début (noté Nd) au nœud de fin (noté Nf). Si un nœud de décision avec un arc de dépendance est traversé (décision 7), on suit l'arc de dépendance (action 8). On ajoute les chemins qui traversent le nœud lié dans une liste LO (action 9). On répète l'opération pour chaque nœud de décision possédant un arc de dépendance.

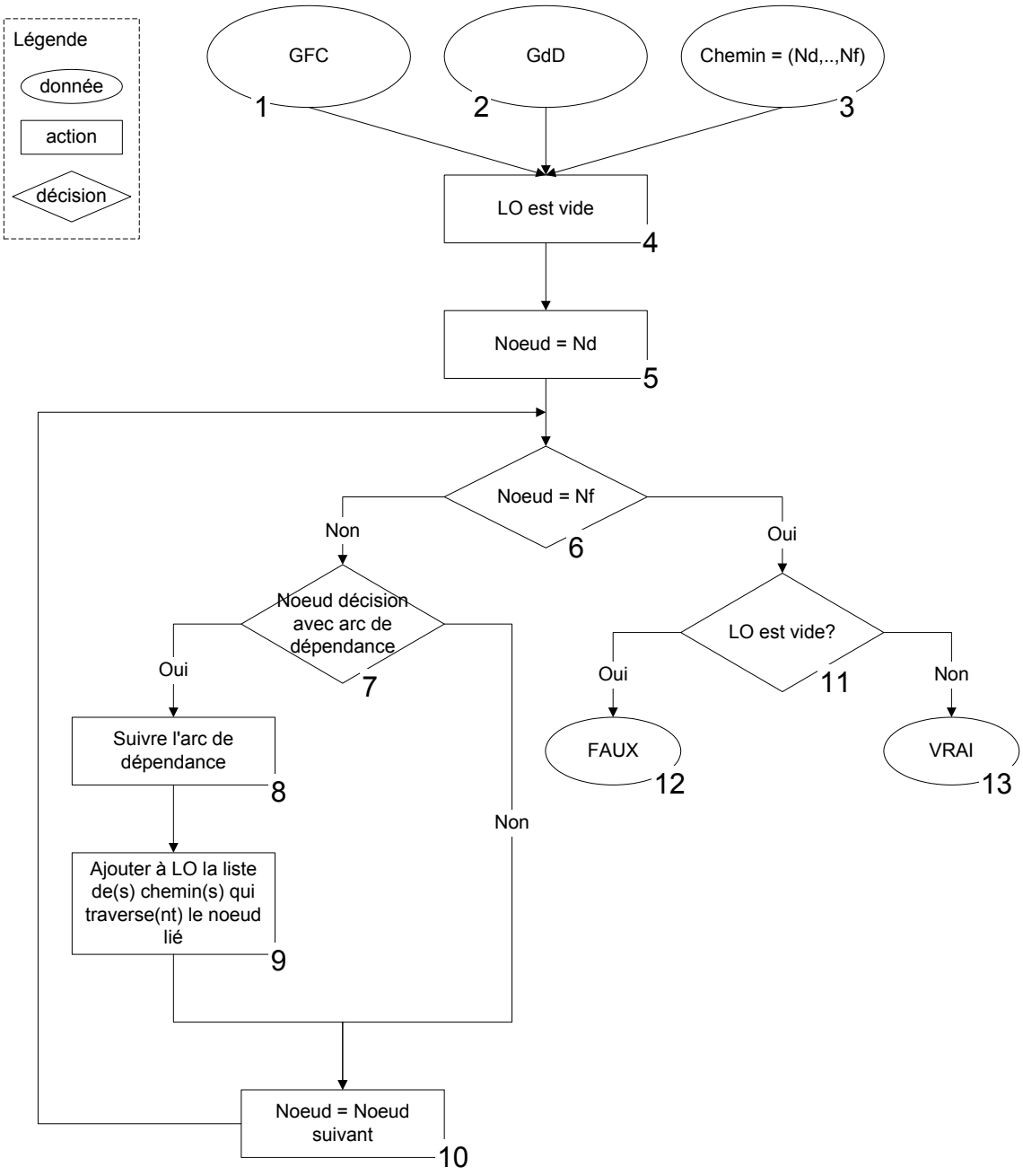


Figure 5-9. Organigramme pour la détection des chemins à ordonnancer et pour la génération de la liste d'ordonnancement.

À la fin du chemin, détecté par la condition 6, l'algorithme se termine. LO est la liste d'ordonnancement qui correspond au chemin à ordonnancer qui a été détecté (donnée 13). Si LO est vide (condition 11), le chemin n'est pas à ordonnancer (donnée 12).

Si on revient à notre algorithme général, on choisi maintenant arbitrairement une permutation P parmi l'ensemble des permutations P_i de LO. L'action `Lancer la méthode générer et tester` est illustrée par l'organigramme de la Figure 5-10. Cette action produit

la séquence de test complète correspondant à un chemin à ordonnancer et à sa liste d'ordonnancement associée. Cette action va énumérer les vecteurs de test pour chaque permutation de la liste d'ordonnancement et tester si la séquence formée est plausible, d'où la dénomination : « générer et tester ».

Trois entrées sont nécessaires : le domaine d'entrée du chemin à ordonnancer, une permutation de la LO associée, l'ensemble des vecteurs de test possibles pour chaque chemin de la permutation (obtenue lors de la phase 3).

Pour chaque chemin de la permutation donnée par l'algorithme général, on choisit un vecteur de test à partir des domaines des solutions du chemin (donnée 3 sur le diagramme). Ce domaine est le résultat de la résolution des contraintes donné par la phase 3. En effet, il est possible que la résolution des contraintes produise plusieurs solutions pour l'exécution d'un chemin. À partir de ces vecteurs de test on constitue une séquence de test (action 4 sur le diagramme). On simule le programme VHDL sous test avec cette séquence afin de recueillir les dernières valeurs des variables et des signaux internes (action 5 sur le diagramme).

On compare ces valeurs avec celles du domaine des solutions correspondant au chemin à ordonnancer (action 6 sur le diagramme). S'il y a une différence de valeur (décision 7 sur le diagramme), on choisit une autre séquence de test. Dans le cas où plus aucune nouvelle séquence n'est disponible (décision 14 sur le diagramme), la valeur `solution` égale à FAUX (donnée 15 sur le diagramme) est rendue à l'algorithme général qui demande une nouvelle permutation.

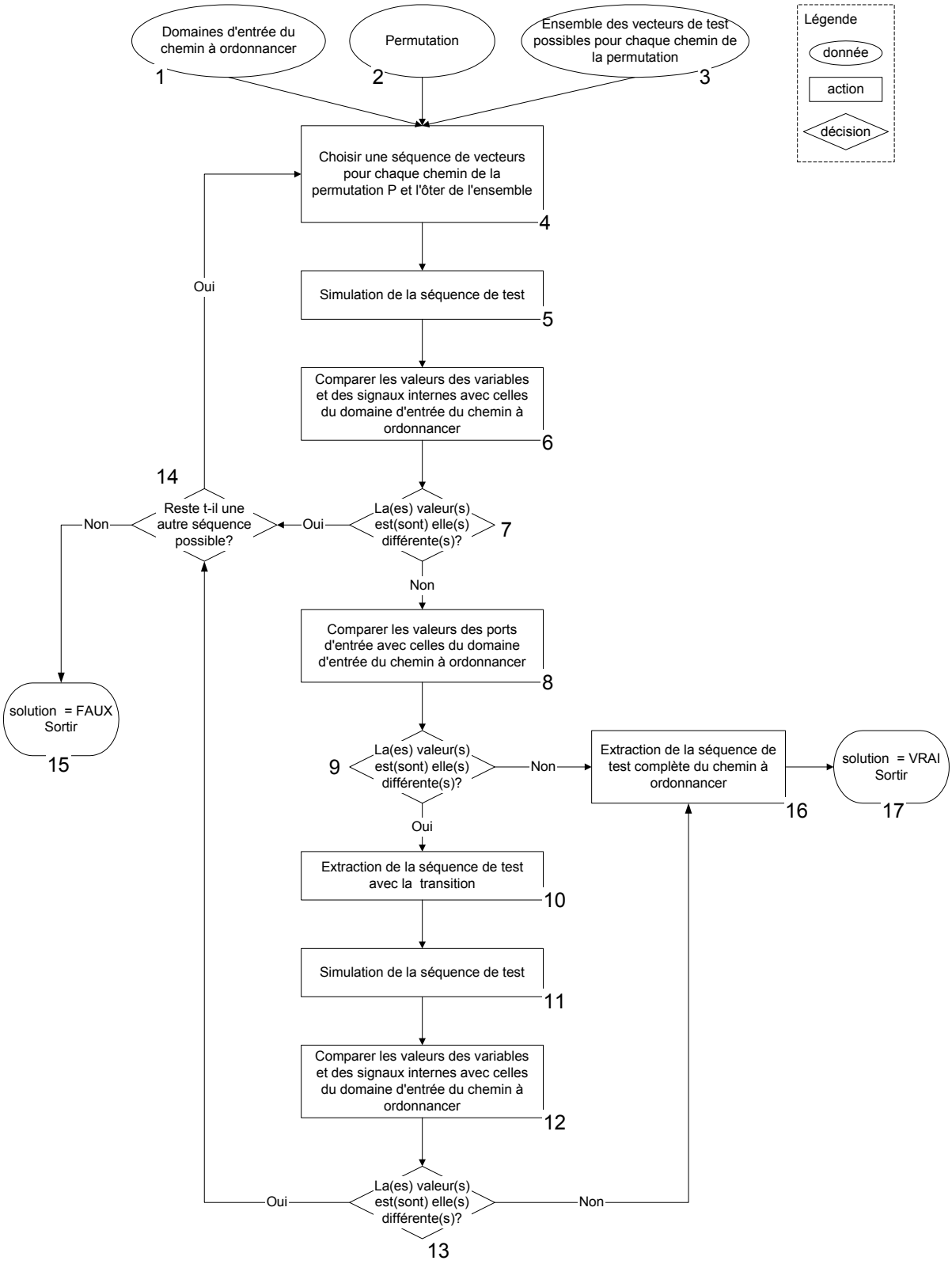


Figure 5-10. Méthode « générer et tester ».

S'il n'y a pas différence de valeur, on compare les dernières valeurs des ports d'entrée de la séquence avec les premières valeurs des ports d'entrée de la séquence de test du chemin

à ordonnancer (action 8 sur le diagramme). S'il n'y a pas de différence de valeur (décision 9 sur le diagramme), la séquence complète est extraite : elle est formée de la séquence de vecteurs de la permutation et des vecteurs de test du chemin à ordonnancer (action 16 sur le diagramme). La valeur `solution` égale à VRAI est rendue à l'algorithme général qui demande le chemin suivant de la base.

S'il y a une différence de valeur, on doit simuler la séquence de test ajoutée de la transition due à la différence de valeurs des ports d'entrée (action 10 sur le diagramme). On compare à nouveau les dernières valeurs des variables et des signaux internes avec celles du domaine des solutions du chemin à ordonnancer (action 12 sur le diagramme). S'il y a une différence de valeur (décision 13 sur le diagramme), on choisit une autre séquence de test pour la permutation. S'il n'y a pas de différence de valeur, la séquence complète est extraite comme précédemment (action 16 sur le diagramme).

5.6 Conclusion

Au cours de ce chapitre nous avons détaillé, étape par étape, notre méthodologie pour la génération des vecteurs de test. Nous avons présenté une vue d'ensemble de notre méthodologie séparée en quatre phases distinctes :

- la production de la base de chemins qui consiste en la construction du GFC modélisant le flot de contrôle du programme VHDL et en la production de la base de chemins indépendants grâce à l'algorithme de Poole ;
- l'analyse des chemins qui permet la détection et la modification des chemins à modifier à partir du PMG et du GdD ;
- l'extraction des vecteurs de test qui est basée sur la génération et la résolution des contraintes ;
- la production des séquences de test qui détecte les chemins à ordonnancer et génère la séquence de test associé.

Nous avons présenté, pour chacune de ces quatre étapes, les différents algorithmes nécessaires à leur réalisation.

Grâce à cette approche, nous avons pu traiter le problème de génération de données d'entrée (stimuli) respectant le critère du test structuré à partir de descriptions VHDL de type

comportemental. Cependant la principale limitation de cette approche est due au fait que la complexité de l'algorithme de la méthode « générer est tester » est exponentielle (problème NP-complet). En général, ce type de problème est résolu par la mise en place d'heuristiques, qui sont des règles générales d'action, applicables à toute situation, permettant la plupart du temps d'aboutir plus rapidement à la solution.

Le chapitre suivant concerne la réalisation informatique de cette approche ainsi que sa validation sur un ensemble de *benchmarks* [Santucci 1999] [ITC'Benchmarks 1999].

CHAPITRE

6

REALISATION INFORMATIQUE ET RESULTATS

Ce chapitre décrit un générateur de vecteur de test pour *test-bench* à partir de descriptions VHDL de type comportemental. Ce prototype logiciel appelé GENESI (pour GENErateur de StimulI pour *test-bench*) a pour objectif de valider la méthode qui a été exposée dans les chapitres précédents.

Nous explorons tout d'abord son architecture logicielle, réalisée pour partie en LISP, pour partie en Prolog. On décrit en particulier l'utilisation de deux logiciels appartenant au domaine public et d'un simulateur commercial VHDL. La seconde partie est consacrée à la définition et à la génération d'une structure de données. Cette structure contient toutes les informations concernant le contrôle et les données manipulés par la description VHDL. Dans la troisième partie on décrit le module de notre logiciel développé en XLISP-Plus, qui permet de construire les graphes et générer les chemins indépendants. Dans la quatrième partie, on présente le système GNU-Prolog qui inclut un langage de description de contraintes et un

résolveur sur les domaines finis. Nous montrons sur deux exemples comment utiliser ce système afin de générer les vecteurs de test correspondant aux chemins indépendants. Enfin, dans la dernière partie, un ensemble de résultats sont présentés et commentés.

6.1 Architecture générale de GENESI

La Figure 6-1 présente l'architecture logicielle du générateur de stimuli pour *test-bench* à partir d'une description VHDL de type comportemental.

Le programme VHDL est compilé en une structure hiérarchique intermédiaire de type arbre représentant non seulement les opérations spécifiées par le programme source VHDL mais aussi les types des différents objets VHDL manipulés. Nous avons utilisé un compilateur existant appartenant au domaine public : le VHDL-1076.1 Parser/Pretty-Printer [Thirunarayan 1997]. À partir de cet arbre, le prototype logiciel GENESI procède en cinq étapes :

i. *génération du code LISP* : cette étape consiste à traduire la structure hiérarchique issue du compilateur en une structure intermédiaire au format LISP.
ii. *construction des graphes :* cette étape permet, à partir de la structure intermédiaire LISP, de construire les graphes définis dans le chapitre 4 : le GFC, le GdD et GMP ;
iii. *production et analyse des chemins :* cette étape permet à partir de l'algorithme de Poole de générer la base primaire de chemins indépendants ; puis d'analyser cette base un utilisant le GdD et GMP. Les chemins à modifier sont inclus dans la base finale après combinaison grâce à la liste des chemins solution et une liste d'ordonnancement est produite pour chaque chemin à ordonnancer ;
iv. *génération et résolution des contraintes* : cette étape consiste à traduire chaque chemin de la base finale sous forme de contraintes en langage clp(FD). Chaque chemin, sous la forme d'un fichier-contrainte, est soumis au résolveur de contraintes du système GNU-Prolog qui produit un domaine de valeurs solution pour les ports d'entrée de la description VHDL ;
v. *production des séquences de test* : cette étape concerne uniquement les chemins à ordonnancer de la base. Les chemins à ordonnancer nécessitent qu'une séquence de test soit exécutée avant le vecteur de test déterminé lors de l'étape précédente. Cette séquence de test est construite à partir de la liste d'ordonnancement associée au chemin. Nous avons utilisé le simulateur VHDL commercial Model*Sim* afin de choisir

les valeurs appropriées à partir de l'ensemble des solutions produit par le système GNU-Prolog.

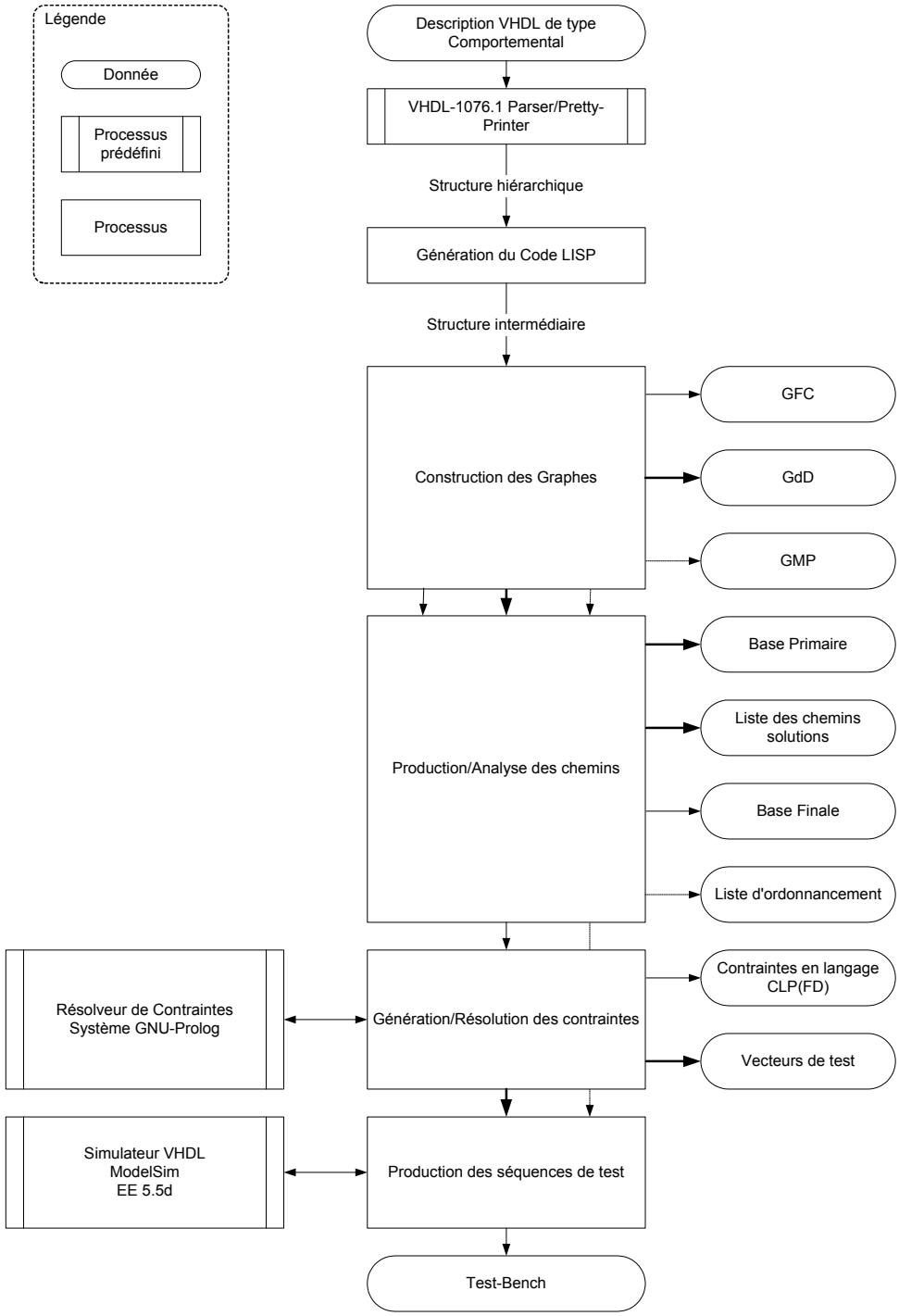


Figure 6-1. Architecture logicielle de GENESI.

Le produit final de GENESI est le *test-bench* permettant de valider par simulation la description VHDL sous test.

Ces étapes, implémentées en langage LISP, sont basées sur l'utilisation d'une structure intermédiaire, nécessairement au format LISP. La section suivante est consacrée à la définition et à la génération de cette structure.

6.2 La structure intermédiaire au format LISP

La structure intermédiaire au format LISP, utilisée par GENESI, se veut conforme au sous-ensemble VHDL défini dans le chapitre 4. Concernant la génération, nous avons choisi d'utiliser un compilateur existant du domaine public : le *VHDL-1076.1 Parser/Pretty-Printer* [Thirunarayan 1997]. On trouve dans la littérature ce type d'outil sous le nom de lexeur-parseur.

6.2.1 Description générale d'un lexeur-parseur

Un lexeur-parseur est un système qui analyse les phrases d'un langage en extrayant la structure grammaticale de la phrase. Un système lexeur-parseur est organisé en deux modules :

- l'analyseur lexical qui accepte en entrée un flot de caractères (un programme en format texte) et le découpe en lexèmes (un lexème correspond à la plus petite unité lexicale) et fournit une séquence d'unités lexicales à l'analyseur syntaxique ;
- l'analyseur syntaxique qui regroupe les unités lexicales en structures grammaticales, vérifie si la syntaxe (grammaire) du programme est correcte, et enfin génère un arbre syntaxique abstrait (AST de l'anglais *Abstract Syntax Tree*) qui représente la structure hiérarchique de la séquence de lexèmes.

La Figure 6-2 illustre le fonctionnement général d'un lexeur-parseur.

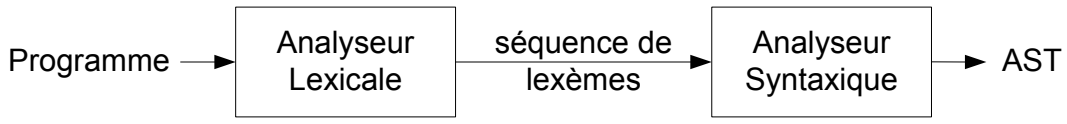


Figure 6-2. Description du lexeur-parseur.

Sur la Figure 6-3 on montre la représentation de l'AST correspondant à l'instruction conditionnelle en pseudo-langage : if A < B + C then X := Y / Z else X := F (A, B, C).

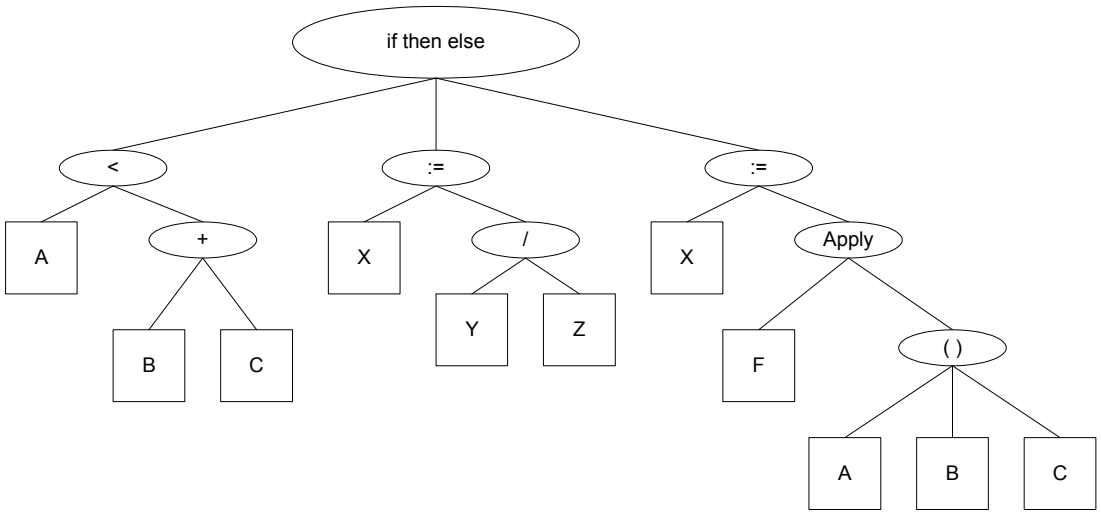


Figure 6-3. Représentation d'un AST.

6.2.2 Le VHDL-1076.1 Parser/Pretty-Printer

Le lexeur-parseur VHDL-1076.1 Parser/Pretty-Printer a été écrit initialement en Quintus Prolog par Peter Reintjes [Reintjes 1988] puis révisé en SWI-Prolog pour supporter le standard IEEE 1076-1993 de VHDL. Ayant opté pour l'utilisation du langage LISP pour implémenter notre prototype logiciel GENESI, nous avons dû modifier son générateur de code (*Pretty-Printer*) pour qu'il fournissent en sortie un format d'AST utilisable par GENESI et conforme au sous-ensemble VHDL défini dans le chapitre 4. La Figure 6-4 reprend l'organisation du lexeur-parseur en mettant en évidence notre modification.

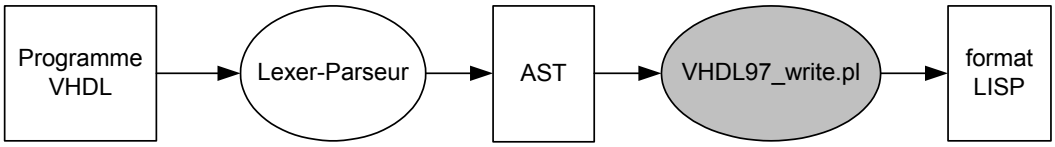


Figure 6-4. Organisation du lexer-parser modifié.

Le lexer-parser VHDL-1076.1 Parser/Pretty-Printer est composé de plusieurs fichiers écrits en Prolog. La liste suivante représente les plus significatifs :

- **vhdl97.pl :** ce fichier Prolog est la programme principal du lexeur-parseur ;
- **vhdl97_IEEE_grammar.pl :** à l'origine, ce fichier contient toutes les règles grammaticales du langage VHDL. Ces règles sont écrites en Prolog. Nous les avons modifié pour prendre en compte notre sous-ensemble VHDL ;

- **vhdl97_tokens.pl :** ce fichier correspond à l'analyseur lexical vu dans la section précédente ;
- **tokens.pl :** ce fichier Prolog contient les plus petits lexèmes du langage VHDL ;
- **vhdl_write.pl :** à l'origine, ce programme produisait un fichier texte à partir des structures de données représentant le programme VHDL. Nous l'avons modifié afin de produire un format LISP qui contient toutes les informations nécessaires non seulement à la construction des différents graphes (CFG, GdD, PMG), mais aussi à l'extraction des contraintes.

La structure de données LISP est détaillée dans le paragraphe suivant. Sa génération est définie par un ensemble de règles Prolog et consiste à construire en mémoire une structure de données. Elle est plus riche qu'un simple ensemble d'arbres syntaxiques et elle permet de représenter :

- toutes les déclarations de types de constantes, de signaux, de variables, de fonctions et de procédures de la description VHDL ;
- toutes les instructions concurrentes de type process ;
- toutes les instructions séquentielles du corps des fonctions, des procédures et des process.

6.2.3 Le format LISP

Le format LISP est généré à partir de l'AST et de règles d'écriture Prolog. Le format LISP représente la structure sémantique du programme VHDL donné en entrée du lexeur-parseur Prolog.

Ces règles ont été écrites dans le fichier Prolog : VHDL97_write.pl. Ce fichier contient environ 1500 lignes et plus de 300 règles. On présente ici seulement trois règles qui nous semble les plus significatives :

- La règle « design unit » est la règle principale qui génère le format LISP :

```
write_vhdl_design_units([],_) --> !, newline, [].
write_vhdl_design_units([Design|Designs],N) -->
{ NN is N+1, name(NN,NNCs) },
newline,
"--  VHDL DESIGN UNIT #", [ NNCs ],
newline,
write_vhdl_design_unit(Design),
write_vhdl_design_units(Designs,NN).
```

- La règle « VHDL entity declaration » concerne la partie entité du programme VHDL :

```
write_vhdl_entity_declaration(entity(ID,GIL,PIL,DIs,Ss)) -->
"(setq entity (",[indent],
"(name ",  write_vhdl_identifier(ID),") ",newline,
write_vhdl_opt_generic_statement(GIL),newline,
write_vhdl_opt_port_statement(PIL),newline,
"(opt_declarative_items    ",write_vhdl_opt_declarative_items(DIs),    ")
",newline,
write_vhdl_opt_entity_body(Ss),"))".
```

- La règle « VHDL architecture body » concerne la partie architecture du programme VHDL :

```
write_vhdl_architecture_body(arch(ID,Entity,DIs,Ss)) -->
      "(setq architecture '(",[indent],
   "(name ",  write_vhdl_identifier(ID),")",newline,
   "(entity_name ", write_vhdl_mark(Entity),")",newline,
   "(opt_declarative_items  ",[indent],
   write_vhdl_opt_declarative_items(DIs), ")",[undent],
   "(concurrent_statements ",[indent],
      write_vhdl_concurrent_statements(Ss),[undent],
      ")", [undent],"))".
```

Ces règles sont utiles pour la traduction de l'AST Prolog en format LISP. Le lecteur peut trouver en annexe 2 un exemple de description VHDL "example.vhd" et le format LISP généré avec les règles du fichier : "VHDL97_write.pl".

De cette façon, les structures sémantiques d'un programme VHDL sont représentées en sortie du lexeur-parseur sous la forme de deux listes :

- la première est stockée dans la variable LISP *entity ;*
- la seconde est stockée dans la variable LISP *architecture.*

Afin de représenter la complexité d'un programme VHDL, ces deux listes respectent un format complexe et précis. Le format de la liste représentée par la variable *entity* permet de représenter complètement la partie déclarative d'un programme VHDL. Ce format est décrit ci-après :

```
(setq entity '(
(name <name of the VHDL program> )
(generic-stat  nil)
(port-stat  (interface-element ( class nil)
      (identifier-list
            (identifier <first port>)
            (identifier <second-port>) …
            (identifier <last-port> ))
```

```
      (mode <kind of port>)
      (type-list <description of the type >) )
      (interface-element ( class nil)
      (identifier-list (identifier <first port>)
            (identifier <second-port>) …
            (identifier <last-port> ))
      (mode <kind of port>)
      (type-list <description of the type >) ) )))
```

Le format LISP correspondant à la description algorithmique VHDL qui contient la comportement de la description du circuit est donné ci-dessous :

```
(setq architecture'(
(name <type of architecture> )
(belong <name of the VHDL program>)
(opt-declarative-items  (object-declaration ( class <kind of object>)
      (identifier-list
            (identifier <first object>)
            (identifier <second-object>) …
            (identifier <last-object> ))
      (type-list <description of the type >) )
      (signal-kind <kind-of-signal> )
      (assignment <signal-initialization>) )
            (object-declaration ( class <kind of object>)
      (identifier-list (identifier <first object>)
            (identifier <second-object>) …
            (identifier <last-object> ))
      (type-list <description of the type >) )
      (signal-kind <kind-of-signal> )
      (assignment <signal-initialization>) ) )
(concurrent-statements (concurrent-statement
      (process (opt-label <first-process> )
      (opt-label <second-process> ) …
            (opt-label <last-process>) )
             (opt-sensitivity-list <list of sensitive signals>)
            (opt-decl-items <list of declarations>)
            (seq-statements
                  (statement <first sequential statement>)
                  (statement <second sequential statement>) …
                  (statement <last sequential statement>)))))))
```

Nous avons décrit dans cette sous-section le format LISP que nous produisons à partir de la sortie d'un lexeur-parseur Prolog existant. La prochaine section concerne l'implémentation des deux premières étapes de GENESI à partir de ce format.

6.3 Construction des graphes et analyse des chemins

La structure de données issue du lexeur-parseur Prolog ne met pas en évidence la séparation entre données et commandes. Afin de rendre cette séparation explicite, nous avons défini une nouvelle structuration des données. Nous avons choisi une approche objet qui propose un schéma simple de modélisation à base de classes et de composants. Ce mode de

conception facilite la maintenabilité, l'évolutivité du logiciel et favorise la réutilisabilité des composants. Neuf classes ont été créées :

- la classe BEHDESC qui permet de décrire le programme VHDL sous test. C'est la classe principale de GENESI ;
- la classe PROCESS qui modélise un *process* dans un programme VHDL ;
- la classe TYPE représente les types autorisés dans notre sous-ensemble VHDL ;
- les classes SIGNAL, VARIABLE et CONSTANT décrivent les éléments mémorisant de VHDL ;
- les classes GRAPH, NODE et EDGE permettent de modéliser les graphes associés à un programme VHDL ;
- la classe PATH permet de décrire un chemin.

Chacune de ces classes représente la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Elles sont composées de deux parties :

- les attributs : Il s'agit des données caractérisant l'objet. Ce sont des variables stockant des informations d'état de l'objet ;
- les méthodes : Les méthodes d'un objet caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées *opérations*) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets). De plus, les opérations sont étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier.

Par exemple, la classe BEHDESC qui permet de décrire une description VHDL a été définie comme suit :

```
(defclass BEHDESC
        (name parser_ent parser_arch port_in port_out int-sign const
            process_list type_list C_graph list_basis_path))
```

où les différents attributs sont :

- name : chaîne qui représente le nom de la description, donné par le nom de l'entité ;
- parser_ent : code associé avec la définition de l'entité (donné par le parseur) ;
- parser_arch : code associé avec la définition de l'architecture (donné par le parseur) ;
- port-in : liste des ports d'entrée de la description (liste des instances de la classe SIGNAL) ;
- port-out : liste des ports de sortie de la description (liste des instances de la classe SIGNAL) ;
- int-sign : liste des signaux internes de la description (liste des instances de la classe SIGNAL) ;

- const : liste des constantes de la description (liste des instances de la classe CONSTANT) ;
- process_list : liste des process de la description (liste des instances de la classe PROCESS) ;
- type_list : liste des différents types de la description (liste des instances de la classe TYPE) ;
- list_basis_path : représente l'ensemble des chemins indépendants ;
- c_graph : graphe représentant le flot de contrôle de la description (instance de la classe GRAPH).

Un exemple d'instance de la classe BEHDESC est l'objet suivant :

```
BEHDESC_0
      name = example
      parser_ent = ((opt_port_statement (interface_element (class …
      parser_arch = ((opt_declarative_items …
      port_in = ( SIGNAL_1 SIGNAL_2)
      port_out = ( SIGNAL_1 )
      int_sign = ( SIGNAL_2 )
      const = nil
      process_list = ( PROCESS_1 PROCESS_2 )
      type_list = nil
      list_basis_path = nil
      c_graph = GRAPH_0
```

Deux exemples d'instances de la classe PROCESS sont les objets suivants :

```
PROCESS_0
      linked_to = BEHDESC_0
      name = #:process0
      sensitive_list = (in1 in2)
      variable_list = nil
      in_sign = (in1 in2)
      out_sign = (a)
      parser_code = (process ( …

PROCESS_1
      linked_to = BEHDESC_0
      name = #:process1
      sensitive_list = (a)
      variable_list = nil
      in_sign = (a)
      out_sign = (out1)
      parser_code = (process ( …
```

6.3.1 Construction des graphes

La première étape est illustrée sur la Figure 6-5. Elle consiste à construire, à partir de la structure intermédiaire issue du compilateur, les graphes définis dans le chapitre 4 :

- le Graphe de Flot de Contrôle (GFC) ;
- le Graphe de Dépendance (GdD) ;
- le Process Model Graph (PMG).

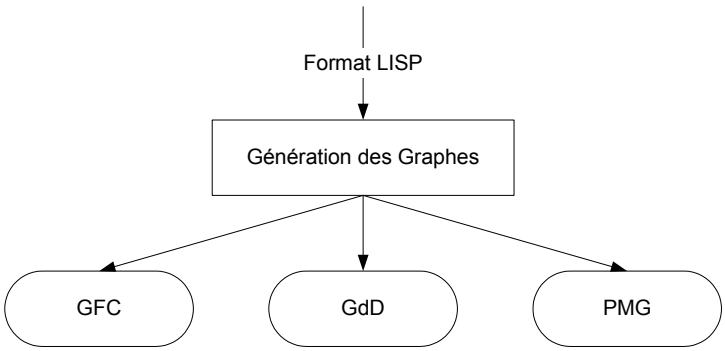


Figure 6-5. Construction des graphes.

La méthode construct() de la classe BEHDESC permet la création de l'objet GRAPH qui représente le GFC de la description. Elle procède ainsi :

1. appel de la méthode construct_end_node afin de créer une instance de la classe NODE qui représente le nœud de fin de process du GFC ;
2. appel de la méthode construct_source_node afin de :
    a. créer le nœud de début (instance de la classe NODE) ;
    b. créer le nœud process (instance de la classe NODE) ;
    c. créer les arcs de type contrôle liant les deux nœuds précédents ;
    d. commencer la génération du graphe pour chaque process par un appel de la méthode add_node ;
3. appel de la méthode close_graph afin de terminer la génération du GFC.

Le GdD est construit avec les mêmes nœuds que le GFC. Par contre, les nœuds du GdD sont liés par des arcs de type dépendance. Pour chaque signal interne et pour chaque variable on crée deux listes en mémoire :

- la liste des nœuds affectation impliquant le signal ou la variable ;
- la liste des nœuds de décision impliquant le signal ou la variable.

À partir de ces deux listes, on lie les deux types de nœuds par un arc de type dépendance. Le PMG n'est pas représenté comme un graphe. On extrait les informations nécessaires grâce aux attributs respectifs des classes PROCESS et BEHDESC :

- *in_sign* et *out_sign* représentent les listes des objets de la classe SIGNAL correspondant respectivement aux signaux entrants et sortants de l'instruction process ;

- *int_sign* représente la liste des signaux internes qui connectent les *process*. Cette liste est vide dans les cas où les *process* ne sont pas connectés.

6.3.2 Production et analyse des chemins

La seconde étape est illustrée sur la Figure 6-6. Elle a pour objectif de créer à partir du GFC une base de chemins indépendants appelée base primaire et surtout d'identifier la liste d'ordonnancement associée aux chemins à ordonnancer ainsi que la liste des chemins solution qui après combinaison avec les chemins à modifier sont injectés dans la base finale.

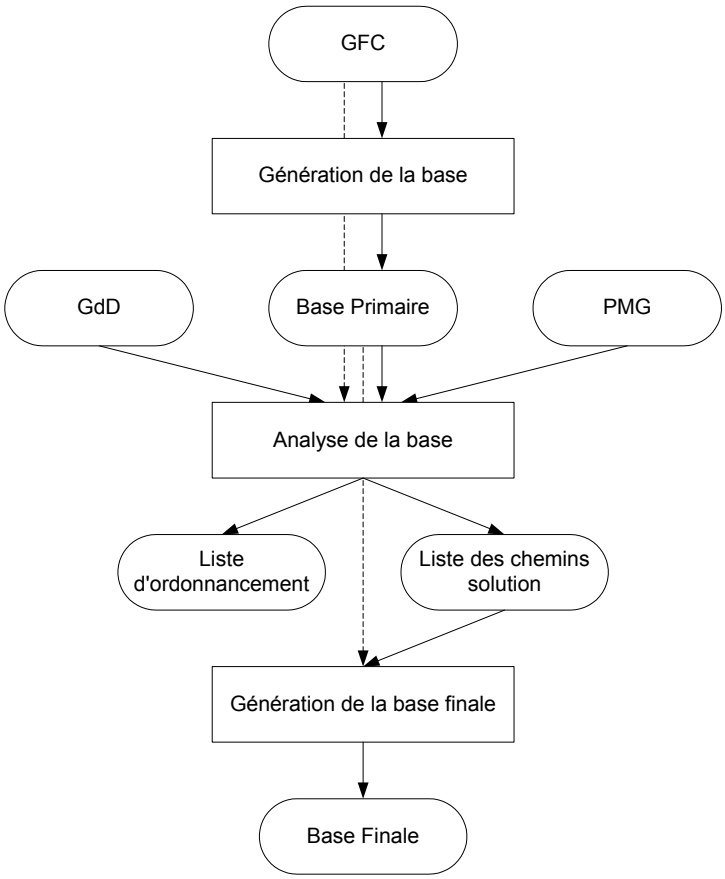


Figure 6-6. Production et analyse des chemins.

La génération de la base correspond à l'algorithme de Poole qui est implémenté dans la méthode find_basis() de la classe GRAPH. Le résultat renseigne l'attribut list_basis_path de la classe principale BEHDESC, représentant ainsi la base primaire.

La classe PATH qui représente l'ensemble des chemins indépendants a été définie comme suit :

```
(defclass PATH
        (name edge_list linked_path_list))
```

où les types des différents attributs sont :

- name qui représente le nom du chemin ;
- edge_list qui représente la liste des arcs parcourus par le chemin ;
- linked_path_list qui représente la liste des chemins liés par un arc de dépendance à un nœud parcouru par le chemin.

La méthode set_linked_path() de la classe PATH parcourt les nœuds du chemin à travers le GFC et GdD et renseigne l'attribut linked_path_list. La méthode booléenne process_linked() de la classe PATH renvoie une valeur VRAI si le *process* est connecté. La méthode modify() permet de combiner les chemins à modifier et les inclut dans la base finale. La liste d'ordonnancement et la liste des chemins solution sont disponibles grâce à l'attribut linked_path_list de la classe PATH.

6.4 Génération et résolution des contraintes

La troisième étape est illustrée sur la Figure 6-7. Cette étape est en cours d'implémentation. Elle a pour objectif de générer et résoudre les contraintes pour chaque chemin de la base finale. Pour cela on applique les algorithmes qui ont été décrits dans le chapitre 5.

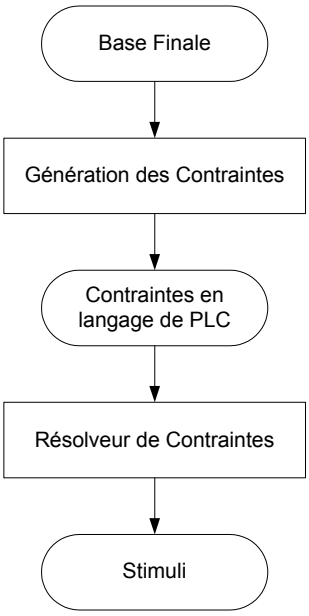


Figure 6-7. Génération et résolution des contraintes.

Pour chaque chemin on veut générer un fichier contrainte dans un langage de Programmation Logique avec Contraintes (PLC).

La première étape consiste en la traduction, pour chaque chemin de la base, des instructions VHDL traversées par le chemin considéré en terme de contraintes relationnelles et contraintes de domaine. Un format intermédiaire, vu au chapitre 5, a été utilisé pour représenter les incarnations spatiales et temporelles des objets VHDL traversés par le chemin. Ce format doit être traduit en langage de PLC.

La seconde étape concerne la résolution des contraintes générées au cours de l'étape précédente à l'aide du résolveur attaché au langage de PLC.

Nous présentons dans la section suivante le système GNU-Prolog, un compilateur natif pour le langage Prolog incluant un puissant résolveur de contraintes sur les domaines finis. Le système GNU-Prolog a été développé par Daniel Diaz [Diaz 1999] et retenu par GNU pour être le Prolog officiel de cette organisation. Après un aperçu général de GNU-Prolog, nous montrons quelques aspects du résolveur de contraintes sur les domaines finis de GNU-Prolog et, en particulier, le langage de description de contraintes. Enfin, nous montrons deux exemples d'utilisation de GNU-Prolog.

6.4.1 Aperçu général de GNU-Prolog

Le compilateur GNU-Prolog est basé sur la machine abstraite de Warren (WAM en anglais pour Warren Abstract Machine). Il compile d'abord un programme Prolog en un fichier WAM qui est alors traduit en langage bas niveau indépendant de la machine appelé « mini-assembly » spécialement conçu pour GNU-Prolog. Le fichier résultat est alors traduit en langage assembleur de la machine cible (à partir duquel un objet est obtenu). GNU-Prolog accepte donc des programmes Prolog avec contraintes et produit des exécutables (binaires) entièrement autonomes dont la taille reste réduite. GNU-Prolog offre également la possibilité d'exécuter un programme à l'aide de l'interpréteur pour en faciliter la mise au point. En termes de performances, GNU-Prolog est comparable aux meilleurs systèmes commerciaux. GNU-Prolog est un système complet, robuste, efficace et compatible avec la norme ISO pour Prolog mais offrant également bon nombre d'extensions telles que : les variables globales et les tableaux.

De plus il intègre un résolveur de contraintes très efficace alliant ainsi la puissance de la programmation par contraintes à l'aspect déclaratif de la programmation logique.

Le résolveur suit le schéma « Constraint Logic Programming » introduit par Jaffar et Lassez [Jaffar 1987]. Les contraintes sur les domaines finis sont résolues en utilisant des techniques de propagation, en particulier la consistance d'arc (AC an anglais pour arc-consistency). Le lecteur intéressé peut se référer à « Constraint Satisfaction in Logic Programming » de P. Van Hentenryck [Van Hentenryck 1989]. Le résolveur est basé sur le résolveur clp(FD) [Diaz 1999]. Le résolveur GNU-Prolog offre des contraintes arithmétiques, des contraintes booléennes, et des contraintes symboliques sur les variables de domaines finis.

6.4.2 Le langage clp(FD)

Le langage clp(FD) est un langage de programmation avec contraintes sur les domaines finis. Sa principale caractéristique est son implantation basée sur la décomposition de contraintes complexes en expressions utilisant une unique contrainte primitive. Le GNU-Prolog est basé sur la même idée mais étend la puissance de la primitive pour rendre possible des définitions de contraintes plus complexes. Par rapport à clp(FD), GNU-Prolog offre de nouvelles contraintes prédéfinies, et de nouveaux heuristiques prédéfinis.

Comme Prolog, le langage clp(FD) utilise la notion de clauses, de règles, de prédicats et d'arguments. Le langage clp(FD) contient des prédicats prédéfinis qui permettent :

- d'unifier, de tester l'égalité ou de comparer des termes ;
- de définir un domaine fini avec par exemple :
    - fd_domain(Vars, Lower, Upper) qui contraint chaque élément X de Vars à prendre les valeurs Lower .. Upper ;
    - fd_domain_bool(Vars) qui est équivalant à fd_domain(Vars, 0, 1) et est utilisé pour déclarer des variable de domaine fini booléennes.

On présente ici quelques contraintes prédéfinies :

- les contraintes arithmétiques :
  FdExpr1 #= FdExpr2 qui contraint FdExpr1 à être égal à FdExpr2 ;
  FdExpr1 #\= FdExpr2 qui contraint FdExpr1 à être différent de FdExpr2 ;
- les contraintes booléennes :
  FdBoolExpr1 #<=> FdBoolExpr1 qui contraint FdBoolExpr1 à être équivalent à FdBoolExpr1 ;

FdBoolExpr1 #\<=> FdBoolExpr2 qui contraint FdBoolExpr1 à être équivalent à NON FdBoolExpr2 ;

FdBoolExpr1 #/\ FdBoolExpr2 qui contraint FdBoolExpr1 ET FdBoolExpr2 à être VRAI ;

FdBoolExpr1 #\/ FdBoolExpr2 qui contraint FdBoolExpr1 OU FdBoolExpr2 à être VRAI ;

- les contraintes *labeling* :

  Fd_labeling (Vars, Options) affecte une valeur à chaque variable X de la liste Vars selon une liste d'options de *labeling* donnée par Options.

6.4.3 Exemples d'utilisation

La Figure 6-8 montre comment nous utilisons le système GNU-Prolog dans GENESI. La génération des contraintes, en un format intermédiaire, est utilisé pour créer le fichier en langage clp(FD).

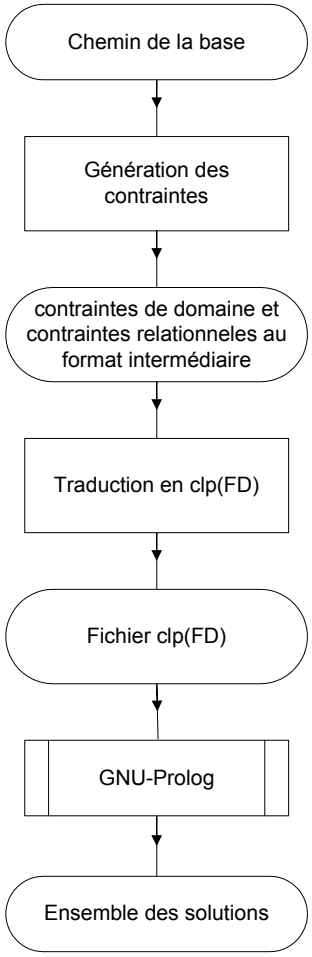


Figure 6-8. Un cas d'utilisation.

La Figure 6-9 reprend un exemple du chapitre précédent, où le code VHDL est à gauche et le résultat, en format intermédiaire, de la génération des contraintes est sur la droite.

```
entity example is
        port ( in_1,in_2: in bit;
                out_1: out bit);
end example;
architecture behavior of example is
signal a:bit;
begin
process1: process(in_1, in_2)
begin
        a <= in_1 AND in_2;
end process;
process2: process(a)
begin
        if a = '1' then
                out_1 <= '1';
end if;
end process;
end behavior;
```

```
Contraintes de domaines :
in_1, in_2, a ∈ [0,1]

Contraintes relationnelles
T=1, Delta=0, d(a)=0, s(a)=0

in_1^1_0 ≠ in_1^0_0 ou in_2^1_0 ≠ in_2^0_0
s(a)=1
a^0_1 = a^0_0
a^1_1 = in_1^1_0 et in_2^1_0
d(a)=1

Delta=1
a^1_1 ≠ a^0_1
a^1_1 = 1
Delta=2
```

Figure 6-9. Génération des contraintes.

On présente ci-dessous le fichier en langage clp(FD) que l'on doit soumettre au résolveur de contraintes du système GNU-Prolog :

```
solution(P):-
      P = [In_1_0_0,In_1_1_0,In_2_0_0,In_2_1_0],
      fd_domain_bool([In_1_0_0,In_1_1_0,In_2_0_0,In_2_1_0,
                      A_0_0,A_0_1,A_1_1]),
      (In_1_0_0 #\<=> In_1_1_0) #\/ (In_2_0_0 #\<=> In_2_1_0),
      (A_0_1 #<=> A_0_0),
      (A_1_1) #<=> (In_1_1_0 #/\ In_2_1_0),
      (A_1_1 #\<=> A_0_1),
      (A_1_1 #<=> 1),
      fd_labeling(P).
```

À partir de ce code clp(FD), on pose au résolveur une requête pour obtenir l'ensemble des solutions. Si on pose la requête solution(V), qui correspond aux contraintes du chemin traversant les deux process, on obtient les résultats suivants :

V = [0,1,0,1]

V = [0,1,1,1]

V = [1,1,0,1]

Ce résultat correspond à l'ensemble des solutions trouvées par le résolveur. Chaque ensemble de valeur, entre crochet, est un vecteur de test possible. Les deux premières valeurs

correspondent aux vecteurs à appliquer au port in_1 et les deux suivantes aux vecteurs à appliquer au port in_2.

La Figure 6-10 montre un second exemple du chapitre précédent, où le code VHDL est à gauche et le résultat, en format intermédiaire, de la génération des contraintes est sur la droite.

```
entity example is
        port (in_1,in_2:in bit;
              out_1:out bit);
        end example;
architecture behavior of example is
begin
process1: process(in_1)
variable a: bit;
begin
        if (in_1='1')
                then a:=not in_2;
        elsif (a='1')
                then out_1 <= '1';
        end if;
end process;
end behavior;
```

```
Contraintes de domaines :
in_1, in_2, a ∈ [0,1]

Contraintes relationnelles :
T=1, Delta=0, s(a)=0

« Chemin 1 »
in_1_0^1 ≠ in_1_0^0
in_1_0^1 = 1
s(a)=1
a_1 ≠ in_2_0^1
Delta=1

« Chemin 2 »
in_1_0^1 ≠ in_1_0^0
in_1_0^1 ≠ 1
a_0 = 1

Delta=1
```

Figure 6-10. Génération des contraintes.

On présente ci-dessous le fichier en langage clp(FD) que l'on doit soumettre au résolveur de contraintes du système GNU-Prolog :

```
chemin1(P):-
      P = [In_1_0_0,In_1_1_0,In_2_0_0,In_2_1_0,A_0,A_1],
      fd_domain_bool([In_1_0_0,In_1_1_0,A_0,A_1]),
      In_1_0_0 #\<=> In_1_1_0,
      In_1_1_0 #<=> 1,
      A_1 #\<=> In_2_1_0,
      fd_labeling(P).

chemin2(Q):-
      Q = [In_1_0_0,In_1_1_0,In_2_0_0,In_2_1_0,A_0,A_1],
      fd_domain_bool([In_1_0_0,In_1_1_0,In_2_0_0,In_2_1_0,A_0,A_1]),
      In_1_0_0 #\<=> In_1_1_0,
      A_0 #<=> 1,
      fd_labeling(Q).
```

Les deux prédicats chemin1() et chemin2() représentent respectivement les contraintes correspondant à deux chemins traversant le code VHDL. Le premier chemin correspond à la

condition in_1='1' = VRAI, alors que le second chemin correspond à la condition a='1' = VRAI. À partir de ce code clp(FD) on pose une requête au résolveur pour obtenir l'ensemble des solutions.

En posant la requête chemin1(V), qui correspond aux contraintes du premier chemin, on obtient les résultats suivants :

V = [0,1,0,0,0,1]
V = [0,1,0,0,1,1]
V = [0,1,0,1,0,0]
V = [0,1,0,1,1,0]
V = [0,1,1,0,0,1]
V = [0,1,1,0,1,1]
V = [0,1,1,1,0,0]
V = [0,1,1,1,1,0]

Ce résultat correspond à l'ensemble des solutions trouvées par le résolveur. Chaque ensemble de valeur, entre crochet, est un vecteur de test possible. Les deux premières valeurs correspondent aux vecteurs à appliquer au port in_1 et les deux suivantes aux vecteurs à appliquer au port in_2. Les dernières valeurs correspondent aux valeurs de la variable a. Elles sont utilisées dans le cas des chemins à ordonnancer.

En posant la requête chemin2(V), qui correspond aux contraintes du premier chemin, on obtient les résultats suivants :

V = [1,0,0,0,1,0]
V = [1,0,0,0,1,1]
V = [1,0,0,1,1,0]
V = [1,0,0,1,1,1]
V = [1,0,1,0,1,0]
V = [1,0,1,0,1,1]
V = [1,0,1,1,1,0]
V = [1,0,1,1,1,1]

Ce résultat correspond à l'ensemble des solutions trouvées par le résolveur. Chaque ensemble de valeur, entre crochet, est un vecteur de test possible. Les deux premières valeurs

correspondent aux vecteurs à appliquer au port in_1 et les deux suivantes aux vecteurs à appliquer au port in_2. Les dernières valeurs correspondent aux valeurs de la variable a. Elles sont utilisées dans le cas des chemins à ordonnancer.

6.5 Résultats

Le tableau suivant montre les premiers résultats obtenus pour les descriptions VHDL des *benchmarks* ITC'99 [ITC'Benchmarks 1999] :

	#lignes	#process	#ports d'entrée	#ports de sortie	#nœuds	#arcs	#v(G)	#t1	# t2
B01	111	1	4	2	50	67	19	0.16	0.06
B02	71	1	2	1	29	40	13	0.05	0.00
B03	142	1	6	1	73	90	19	0.22	0.00
B04	103	1	6	1	47	57	12	0.11	0.05
B05	333	3	3	6	153	203	52	0.77	0.06
B06	129	1	4	4	64	79	17	0.17	0.05
B07	93	1	3	1	45	58	15	0.11	0.00
B08	90	1	4	1	33	42	11	0.05	0.00
B09	104	1	3	1	45	54	11	0.11	0.00
B10	168	1	10	3	96	123	29	0.28	0.05
B11	119	1	4	1	56	77	23	0.22	0.06
B12	570	4	4	3	254	331	79	0.88	0.05
B13	297	5	5	7	138	192	56	0.60	0.05
B14	518	1	34	54	366	514	164	1.38	0.11
B15	648	3	37	70	381	485	111	1.38	0.10
B20	1040	3	34	22	736	1035	329	3.52	0.39
B22	1547	4	34	22	1100	1548	492	5.99	0.61

Tableau 6-1. Résultats sur les benchmarks ITC'99.

La première colonne correspond aux noms des descriptions. Nous n'avons pas pris en compte que les descriptions VHDL B16 à B19 et B21 car elles sont de type structurel. Les quatre colonnes suivantes indiquent les caractéristiques de ces programmes : nombre de lignes, nombre de *process*, nombre de ports d'entrée et nombre de ports de sortie. Les autres colonnes présentent les résultats obtenus avec GENESI. Les deux dernières colonnes correspondent au temps en seconde, pour la construction du GFC (t1), et pour la construction des chemins (t2).

La première remarque est que ces temps sont courts. En effet, le temps de construction pour environ 1000 nœuds et 1500 arcs n'est que de 6 secondes, alors que le temps de construction des chemins est à chaque fois en dessous des 6 secondes.

La seconde remarque est que la complexité cyclomatique v(G) est supérieure à 100 pour quatre descriptions VHDL. Watson et McCabe [Watson 1996a] recommande de limiter pour un module logiciel la complexité à 10. Les descriptions VHDL des *benchmarks* sont

toutes supérieures à ce nombre. On peut faire le constat que le test de telles descriptions est beaucoup plus coûteux que le test de modules de logiciels.

6.6 Conclusion

Au cours de ce chapitre nous avons tout d'abord présenté l'architecture logicielle du prototype GENESI qui a été implantée afin de valider notre méthode de génération de vecteurs de test à partir d'une description VHDL de type comportemental. Pour chaque module logiciel du prototype GENESI nous avons introduit les structures mises en jeu et leur évolution après chaque étape.

À partir d'un lexeur-parseur existant, nous avons généré un format LISP contenant non seulement les informations sur le flot de contrôle et de données de la description VHDL mais aussi les informations concernant la déclaration et le type des objets VHDL (port, signal interne, variable, constante). Cette structure LISP ne permettant pas la séparation explicite des données et des commandes, nous avons utilisé une approche orientée objet pour construire le Graphe de Flot de Contrôle, le Graphe de Dépendance et le Graphe de Modélisation de Process. L'algorithme de Poole a été implémenté pour générer une base de chemins indépendants. Cette partie du prototype logiciel a été validée sur les descriptions VHDL de type comportemental des *benchmarks* ITC'99 [ITC'Benchmarks 1999].

Nous avons présenté le système GNU-Prolog, un compilateur natif pour le langage Prolog incluant un puissant résolveur de contraintes sur les domaines finis. Nous avons montré que l'on peut traduire chaque chemin indépendant de la base en un ensemble de contraintes écrites en langage clp(FD). Nous avons ensuite vu que le résolveur permet d'obtenir un ensemble de solutions pour les ports d'entrée de la description VHDL. Deux exemples concrets pour lesquels GNU-Prolog a été utilisé avec succès ont été décrits.

Les programmes concernant l'analyse des chemins et la génération des contraintes en langage clp(FD) directement à partir du format LISP sont en cours de validation.

Ce travail s'insère dans le cadre d'un projet soutenu par l'EOARD (*European Office of Aerospace Research and Development*) sous le numéro de contrat : F61775-00-C0002 ;

Project 00-4005 “Validation of VHDL descriptions” (2000-2003) qui a fait l’objet de plusieurs rapports [Y1Q2_N°1&2&3 2000], [Y1Q3_N°4&5 2000], [Y1Q4_N°6&7 2001].

CHAPITRE

7

CONCLUSION GENERALE

Dans ce chapitre nous dressons tout d'abord le bilan de l'approche proposée dans le cadre de cette thèse. Nous donnons ensuite quelques perspectives de recherches qui peuvent être entreprises à la suite de nos travaux.

7.1 Le bilan des travaux effectués

L'objectif de cette thèse était de proposer une approche permettant de générer automatiquement à partir d'une description VHDL de type comportemental des vecteurs de test afin de valider la description RTL correspondante par simulation. Notre approche pour la génération des vecteurs de test s'appuie sur des techniques appliquées avec succès dans le domaine du test de logiciels. Nous avons restreint le langage VHDL à un sous-ensemble qui inclut toutes les constructions utilisées pour des caractéristiques comportementales.

Nous avons tout d'abord donné une vue d'ensemble des techniques utilisées dans le domaine du test de logiciels afin de choisir la technique la plus adaptée aux descriptions de circuits digitaux écrites en langage VHDL. Dans ce contexte, comme l'utilisation du code VHDL comportemental exige l'utilisation d'une méthode de test de boîte blanche, le test structurel basé sur un critère de couverture nous a semblé le point de départ le plus approprié. Plus précisément, notre choix s'est porté sur un critère de couverture de chemin qui est le critère le plus rigoureux et le plus efficace parmi ceux appartenant au test structurel (couverture d'instructions, couverture de branches). Un inconvénient majeur est que la couverture à 100% des chemins d'exécution est impossible dès qu'il y a des boucles. Or le critère de test structuré, basé sur l'utilisation du GFC du programme, établit que le cardinal d'un ensemble de chemins indépendants (égal à la complexité cyclomatique v(G)), est suffisant pour tester un programme puisque tout chemin supplémentaire sera combinaison linéaire de la base formée par cet ensemble. Un algorithme puissant (algorithme de Poole) qui permet de générer une base de chemin à partir d'un GFC a été présenté ainsi qu'une méthodologie capable d'extraire les valeurs d'entrée du programme à partir d'un chemin.

Ces concepts qui permettent de tester un logiciel ont été présentés dans le but de les appliquer à des descriptions VHDL de type comportemental. L'étape suivante a consisté naturellement à l'adaptation des techniques précédentes pour des programmes VHDL. Nous considérons une description VHDL de type comportemental comme un module de logiciel avec un point d'entrée et un point de sortie. Nous avons présenté un modèle interne de GFC pour les programmes VHDL basé sur les mêmes composants décrits pour les GFCs de logiciel. Nous avons vu en détail la correspondance entre les instructions de programmes VHDL et leur représentation graphique en tant qu'éléments du GFC. Nous avons détaillé pour chaque instruction leur structure en terme de nœud et d'arc. Les concepts suivants inhérents au langage VHDL : (i) notion de temps, (ii) interconnexion de *process* s'exécutant en parallèle, (iii) mécanisme de retard delta pour les affectations des signaux, nous ont conduit à développer des techniques adaptées. Nous avons notamment défini le nœud *process* qui modélise le parallélisme présent dans les descriptions VHDL à travers l'exécution des *process*. À partir de ce modèle interne, nous avons également défini la notion de chemin d'exécution ainsi que les notions de chemin à modifier, de liste de chemin solution, de chemin à ordonnancer et de liste d'ordonnancement. Afin de résoudre les problèmes liés à ces notions, nous nous sommes appuyés sur deux nouvelles structures de type graphe : le Graphe de Modélisation de Process (PMG) défini par Cho et Armstrong qui modélise la connectivité

entre les *process*, et le Graphe de Dépendance (GdD) qui modélise l'interaction entre les instructions d'affectation et les instructions de contrôle où elles sont utilisées.

Concernant le problème de génération des données d'entrée à partir d'un chemin, nous avons choisi de traduire ce problème par un problème de satisfaction de contraintes (CSP) en utilisant le modèle de contrainte défini par Vemuri et Kalynaraman. À chaque chemin correspond un système de contraintes constitué de contraintes de domaine (i.e. les déclarations des types des objets VHDL) et de contraintes relationnelles (i.e. les instructions traversées par le chemin). Pour résoudre ce système, nous avons utilisé le langage de programmation logique avec contraintes incluant un résolveur de contraintes : clp(FD). Le résolveur se charge d'obtenir l'ensemble des solutions pour chacune des variables d'entrée du programme VHDL.

Les différents algorithmes, utilisant les structures graphiques évoquées précédemment, qui permettent notamment l'analyse des chemins et leur modification ont été présentés en détail ainsi que les algorithmes nécessaires à l'extraction de ces données d'entrée (stimuli) à partir des chemins. Nous avons présenté l'architecture logicielle du prototype GENESI qui a été implantée afin de valider notre approche de génération de stimuli. L'implémentation des différentes parties de notre outil a été réalisée en LISP, sauf le lexeur-parser permettant de passer du code VHDL aux graphes et l'étape de génération et de résolution des contraintes qui ont été réalisés en Prolog. Concernant la partie LISP de notre logiciel, nous avons utilisé XLISP-Plus [Almy 1999] qui est une extension du LISP supportant la Programmation Orientée Objet (POO). La structure LISP, issue du lexeur-parseur Prolog, ne permettant pas la séparation explicite des données et des commandes, nous avons utilisé une approche orientée objet. Cette méthodologie de conception facilite la prise en compte d'éventuelles modifications, la maintenabilité et la réutilisabilité du logiciel.

Enfin nous avons mené un ensemble d'expériences sur des descriptions VHDL de type comportemental, notamment sur les *benchmarks* ITC'99. Nous avons montré les bonnes performances de notre logiciel malgré le fait que les descriptions soient décrites au niveau RTL et non au niveau algorithmique.

La partie logicielle du prototype GENESI, concernant l'analyse des chemins et génération des contraintes en langage clp(FD) directement exploitable par le résolveur, est en cours de validation.

La production automatique du *test-bench* sous la forme d'un programme VHDL est en cours d'implémentation. Nous pensons utiliser le format WAVES [IEEE1029.1 1992] pour la représentation des vecteurs de test.

Concernant le potentiel de GENESI, il réside en l'aide qu'il peut apporter aux concepteurs de circuits digitaux lors des premières phases de leur conception. En général, les concepteurs de circuits génèrent des vecteurs de test de façon aléatoire puis évaluent leur couverture en fonction de métriques (couverture d'instructions, de branches ou bien de fautes « collé-à »). Ils peuvent alors, si la couverture est faible, générer d'autres vecteurs de test pour tester les points critiques de la description non couverts. Ces techniques sont très consommatrices en temps et nécessitent une forte connaissance des fonctionnalités de la description, ce qui peut diminuer la capacité des vecteurs de test à détecter des erreurs de conception. GENESI permet en parti de répondre à ces difficultés, en proposant un ensemble de stimuli respectant le critère du test structuré qui est basé sur la couverture de chemins, et qui inclut le critère de couverture d'instructions et le critère de couverture de branches. De plus, la génération est automatique dans le sens ou l'ensemble des chemins n'a pas besoin d'être annoté, ceci contrairement à l'approche développée par Vemuri et Kalynaraman. Donc aucune connaissance des fonctionnalités de la description n'est requise a priori. Tout se passe comme si la vérification de la description sous test était effectuée par une autre personne que le concepteur, renforçant ainsi la sécurité de fonctionnement de la description en cas de succès du processus de vérification.

7.2 Perspectives de recherche

Nous avons vu que le temps d'exécution de l'algorithme principal de notre approche est exponentiel à cause du caractère NP-complet du problème (voir chapitre 5 section 5.6). La mise en place d'heuristiques constitue de facto une perspective à moyen terme pour cette étude.

Une fois l'implémentation de GENESI terminée, nous pensons mettre en place différentes métriques sur les chemins qui nous permettent d'estimer la contrôlabilité et l'observabilité des ports, signaux et variables. On utilisera ces métriques pour comparer les différents chemins solutions et faire un choix de chemin pour accélérer le temps d'exécution de nos algorithmes. La mise au point de ces métriques requière, en général, un travail de recherche long et fastidieux.

L'étude présentée dans ce mémoire a été développée dans le cadre du projet de recherche : « Validation of VHDL Descriptions », Contrat 0045 (2000-2003) avec l'*European Office of Air Force Research and Development* (*EOARD*), auquel je participe en tant que chercheur.

Ce projet de recherche sur lequel nous travaillons actuellement constitue une première étape vers une aide véritable à la conception de systèmes digitaux. En effet, pour l'instant la méthode que nous proposons permet de générer des vecteurs de test à partir d'une description VHDL de type comportemental au niveau algorithmique et de valider ou d'invalider des descriptions de circuits décrites au niveau RTL. Plusieurs perspectives sont ouvertes par ce travail.

La première perspective porte sur la possibilité de prendre en compte des descriptions VHDL de circuits digitaux encore plus complexes. Nous désirons développer un générateur de *test-bench* global pour des descriptions VHDL de type structurel. Pour cela nous utiliserons les *test-bench* correspondant aux différents composants de type comportemental de la description structurelle.

La seconde perspective vise l'intégration de notre outil pour le test de défauts physique de circuits digitaux. Un couplage de GENESI avec le simulateur de faute comportemental BFS [Federici 1999][Paoli 2000c], développé dans notre laboratoire devrait permettre de générer efficacement des séquences de test de défauts physiques modélisés à haut niveau d'abstraction par des fautes comportementales (voir Figure 7-1).

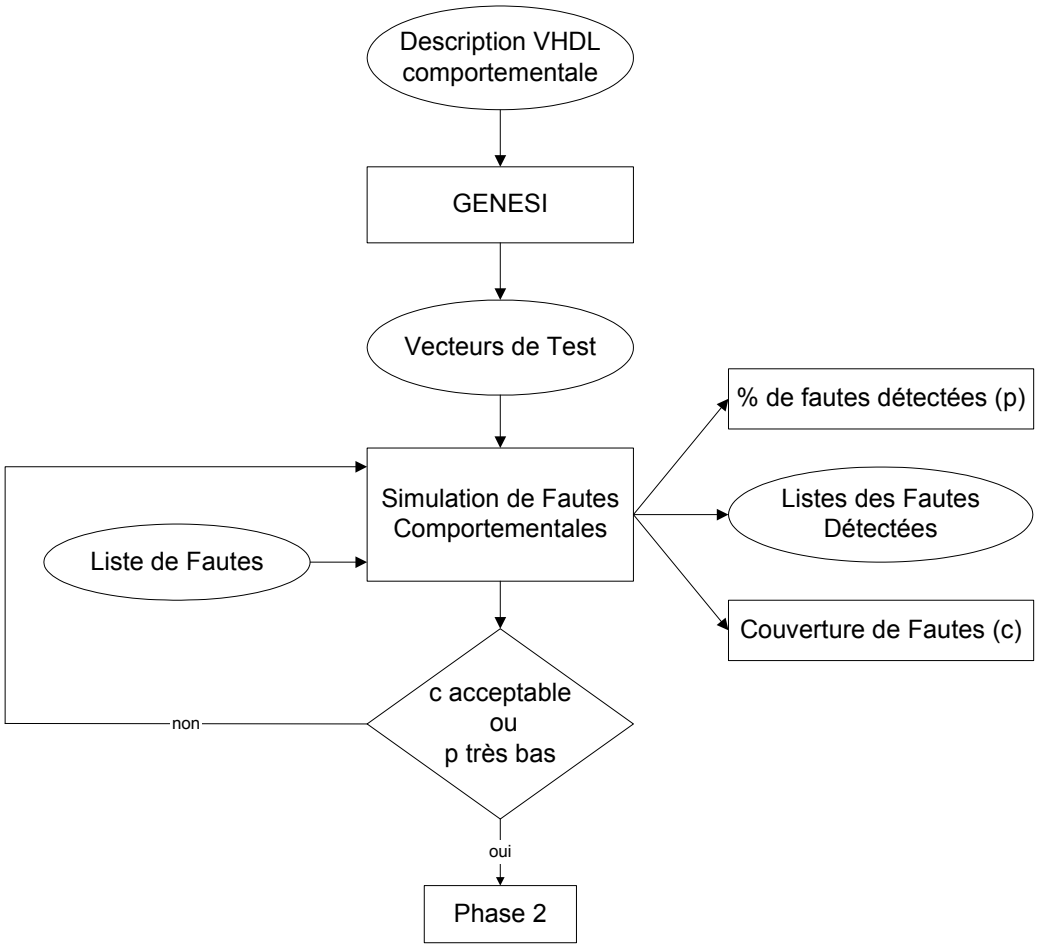


Figure 7-1. Couplage de GENESI à BFS.

La troisième perspective concerne l'aide à la localisation, à l'identification et à la correction des erreurs de conception mises en évidence par l'étape de validation. Nous pensons que la connaissance produite lors de cette étape peut permettre non seulement de valider la description, assurant qu'elle est conforme aux spécifications du circuit, mais aussi d'orienter le concepteur du circuit (comme cela se fait d'ores et déjà en génie logiciel) vers les parties de code susceptibles d'être à l'origine de l'erreur de conception et donc susceptibles d'être modifiées.

Enfin, dans une perspective à long terme, nous pensons qu'il est possible d'intégrer GENESI au sein d'un atelier logiciel regroupant non seulement les activités liées au test de descriptions VHDL de circuits, mais aussi les activités liées à la conception, en particulier l'aide à la conception de description plus facilement testable.

LISTE DES TRAVAUX SCIENTIFIQUES ET PUBLICATIONS

Publications

Manifestation d'audience internationale avec comité de lecture et publication des actes

[Paoli 2002]
C. Paoli, M-L. Nivet, J-F. Santucci et T. Campana, *Path-Oriented Test Data Generation of Behavioral VHDL Description,* IEEE International Workshop on Electronic Design, Test & Applications The Chateau on the Park Hotel Christchurch, New Zealand, January 29-31 2002, à paraître.

[Paoli 2000a]
C. Paoli, M-L. Nivet et J-F. Santucci, *Use of constraint solving in order to generate test vectors for behavioral validation*, communication, actes du « IEEE International High Level Design Validation and Test Workshop (HLDVT'00) », 8-10 Novembre , Berkeley, Californie, USA, pp. 15-20, 2000.
« http://www-cse.ucsd.edu/groups/hldvt/ »

[Paoli 1999a]
C. Paoli et J-F. Santucci, *Validation of Behavioral VHDL Descriptions Using Software Engineering Concepts*, poster, actes de la « IEEE Electronic Circuits and Systems Conference (ECS'99) », 6-8 septembre, Bratislava, Slovaquie, pp. 215-218, 1999.
« http://www.elf.stuba.sk/~ecs2001/ »

[Paoli 1999b]
C. Paoli et J-F. Santucci, *High Level Test Benches Generation for the Validation of VHDL Models of Microelectronic Systems*, communication, actes du « IEEE International Workshop on System Test and Diagnosis workshop (IWSTD'99) », 30 septembre - 1 octobre, Atlantic City, New Jersey, USA, 4 p, 1999.
« http://www.itctestweek.org/itc2000.html#work »

[Paoli 2000b]

C. Paoli, M.L. Nivet et J.F. Santucci, *Test Vectors Generation using Path Selection and Constraint Solving for the Validation of Behavioral VHDL Design*, Communication, actes du « IEEE International Workshop on System Test and Diagnosis workshop (IWSTD'00) », 5-6 octobre, Atlantic City, New Jersey, USA, 5 p, 2000.

« http://www.itctestweek.org/itc2000.html#work »

[Paoli 2000c]

C. Paoli, P-A. Bisgambiglia, D. Federici et J-F. Santucci, *High Level Validation of VHDL Description using Behavioral Fault Simulation*, poster, actes du congrès international Environnement et Identité en Méditerranée, programme de coopération scientifique Interreg II, 13-16 juin, Corté, 2000.

« http://www.univ-corse.fr/recherche/colloques/Interreg/Interreg_fr.html »

Manifestation d'audience internationale sans comité de lecture

[Santucci 1999]

J-F. Santucci et C. Paoli, *High level test bench generation using software engineering concepts*, participation au panel n°6 : « Test Benchmarks » de la « 30th IEEE International Test Conference (ITC'99) », 28-30 septembre, Atlantic City, New Jersey, USA, 1999.

« http://www.itctestweek.org/ »

[Paoli 1998a]

C. Paoli et J-F. Santucci, *Validation of VHDL Description : A Software Engineering Approach*, communication, actes du « 7th BEhavioraL deSIGN methodologies for digital systems Workshop (BELSIGN'98) », 7-8 mai, Twente, Pays-Bas, 4 p, 1998.

« http://www.die.upm.es/research/8thBELSIGN.html »

[Paoli 1998b]
C. Paoli et J-F. Santucci, *Application of Software Metrics for Validating VHDL Descriptions of Digital Circuits*[4], communication, actes du « Advanced Technology Workshop (ATW'98) », 19-20 mai, Ajaccio, 5 p, 1998.
« http://spe.univ-corse.fr/ATW.htm »

[Paoli 1999c]
C. Paoli et J-F. Santucci, *Data generation from VHDL behavioral descriptions*[5], communication, actes du « Advanced Technology Workshop (ATW'99) », 10-11 juin, Ajaccio, 4 p, 1999.
« http://spe.univ-corse.fr/ATW.htm »

[Paoli 2000d]
C. Paoli, M-L. Nivet et J-F. Santucci, *Behavioral VHDL Test Benches Generation*, communication, actes du « Advanced Technology Workshop (ATW'00) », 14-16 Juin, Ajaccio, 2000.
« http://spe.univ-corse.fr/ATW.htm »

Rapports de recherche

[Y1Q2_N°1 2000]
M-L. Nivet, C. Paoli et J-F. Santucci, *Definition of the internal model based on graph structures representing VHDL behavioral descriptions*, Université de Corse, 11 p, septembre 2000.

[Y1Q2_N°2 2000]
J-F. Santucci, C. Paoli et M-L. Nivet, *Algorithms allowing to automatically generate an internal model from a VHDL behavioral description*, Université de Corse, 13 p, septembre 2000.

[4] Cette communication a obtenue le prix du « best paper award ».

[5] Cette communication a obtenue ex æquo le prix du « best paper award ».

[Y1Q2_N°3 2000]
C. Paoli, M-L. Nivet et J-F. Santucci, *Definition of the algorithm alllowing the minimum set of path to be generated*, Université de Corse, 10 p, septembre 2000.

[Y1Q3_N°4 2000]
C. Paoli M-L. Nivet et J-F. Santucci, *Software Implementation of the automatic generation of the internal model*, Université de Corse, 11 p, septembre 2000.

[Y1Q3_N°5 2000]
C. Paoli, M-L. Nivet et J-F. Santucci, *Software implementation of the minimum set of paths generation - S2*, Université de Corse, 10 p, septembre 2000.

[Y1Q4_N°6 2001]
C. Paoli, M-L. Nivet et J-F. Santucci, T. Campana, *Validation of the Software S1 described in the Technical Report N°4 on a set of benchmark VHDL descriptions,* Université de Corse, 33 p, juin 2001.

[Y1Q4_N°7 2001]
C. Paoli, M-L. Nivet et J-F. Santucci, T. Campana, *Validation of the Software S2 described in the Technical Report N°5,* Université de Corse, 11 p, juin 2001.

[DFT 1999]
R. Kozakowski, C. Paoli et D. Saarinen, *Overview of Test Methods Using Boundary Scan*, University of New Hampshire, Durham, NH, USA, 28 p, décembre 1999.
« http://www.ece.unh.edu/Signals%20%26%20Noise/2000/2000visitinggrad.html »

REFERENCES BIBLIOGRAPHIQUES

[Almy 1999]
T. Almy, *XLISP-PLUS 3.04*, 1999.
« http://www.aracnet.com/~tomalmy/xlisp.html »

[Armstrong *et al.* 2000]
J.R. Armstrong and F. Gail Gray, *VHDL Design: Representation and Synthesis*, Prentice Hall, 2000.

[Beizer 1990]
B. Beizer, *Software Testing Techniques*, Van Nostrand Rheinhold, New York, second edition, 1990.

[Beizer 1995]
B. Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, Wiley: New York, 1995.

[Berge 1973]
C. Berge, *Graphs and Hypergraphs*, North-Holland Publishing Company, 1973.

[Cho *et al.*1991]
C. H. Cho and J. R. Armstrong. *VHDL semantics for behavioral test generation*, In D. Borrione and R Waxman, editors, CHDL 91: 10th International Symposium on Computer Hardware Description Languages and Their Applications, pp. 427-444. IFIP WG 10.2, North Holland, April 1991.

[Cho *et al.*1994]
C. H. Cho and J. R. Armstrong, *B-algorithm: A Behavioral Test Generation Algorithm*, Proc. ITC, pp. 968-979, October 1994.

[Corno *et al.* 2000]
F. Corno, M. Sonza Reorda, G. Squillero, A. Manzone and A. Pincetti, *Automatic Test Bench Generation for Validation of RT-Level Descriptions: An Industrial Experience*, Proc. IEEE DATE, March 2000.
« http://www.cad.polito.it/pap/FullDB/exact/codes2000.html »

[DeMillo *et al.* 1991]
R. A. DeMillo and A. J. Offutt, *Constraint-based automatic test data generation*, IEEE Trans. on Software Engineering, Vol. 17, No 9, pp. 900-910, September 1991.

[Devadas *et al.* 1996]
S. Devadas, A. Ghosh, and K. Keutzer, *An observability-based code coverage Metric for Functional Simulation*, Proc. IEEE ICCAD, pp. 418–425, November 1996.

[Diaz 1999]
D. Diaz, *GNU Prolog: A native Prolog compiler with constraint solving over finite domains*, Edition 1.1, November 1999.

[DFT 1999]
R. Kozakowski, C. Paoli et D. Saarinen, *Overview of Test Methods Using Boundary Scan*, University of New Hampshire, Durham, NH, USA, 28 p, décembre 1999.
« http://www.ece.unh.edu/Signals%20%26%20Noise/2000/2000visitinggrad.html »

[Evangelist 1984]
M. Evangelist, *An analysis of control flow complexity*. Proc. CSAC, pp. 388-396, 1984.

[Fallah *et al.* 1998a]
F. Fallah, S. Devadas, and K. Keutzer, *Functional vector generation for HDL models using linear programming and 3-satisfaiability*, Proc. ACM/IEEE DAC, pp. 528–533, 1998.

[Fallah *et al.* 1998b]
F. Fallah, S. Devadas, and K. Keutzer, *OCCOM: Efficient computation of observability-based code coverage metrics for functional verification*, Proc. ACM/IEEE DAC, pp. 152–157, 1998.

[Fallah *et al.* 1999]
F. Fallah, P. Ashar and S. Devadas, *Simulation Vector Generation from HDL Descriptions for Observability Enhanced Statement Coverage*, Proc. Of 36th ACM/IEEE conference on Design automation conference, New Orleans, LA USA, June 21 – 25, 1999.

[Federici 1999]
F. Federici, *Simulation de fautes comportementales de systèmes digitaux décrits à haut-niveau d'abstraction en VHDL*, Thèse de Doctorat, Université de Corse, janvier 1999.

[Ferrandi *et al.* 2000]
F. Ferrandi, F. Fummi, L. Gerli, and D. Sciuto, *Symbolic functional vector generation for VHDL specifications*, Proc. IEEE DATE, pp. 442–446, 2000.

[Ferrandi *et al.* 1998]
F. Ferrandi, F. Fummi, and D. Sciuto. *Implicit test generation for behavioral VHDL models*, Proc. IEEE ITC, pp. 436-441, 1998.

[Gajski 1988a]
D. D. Gajski, *Silicon Compilation*, Addison-Wesley, Reading, Massachusetts, 1988.

[Gajski 1988b]
D. D. Gajski. *A VHDL Subset for Synthesis*, In VHDL Users Group Meeting, October 1988.

[Ghosh 1991]
S. Ghosh, T. J. Chakraborty, *On Behavior Fault Modeling for Digital Designs*, Journal of Electronic Testing: Theory and Applications, Vol.2, pp.135-151, 1991.

[Govindarajan *et al.* 1999]
S. Govindarajan, N. Narasimhan and R. Vemurri, *Dependency analysis and operation graph generation for high-level synthesis from behavioral VHDL*, April 1999.
« http://www.ececs.uc.edu/~ddel/projects/dss/dss.html »

[Ho *et al.* 1995]
R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. *Architecture validation for processors*, Proc. Int. Symp. Computer Architecture, pp. 404-413, 1995.

[IEEE1076 1993]
IEEE Press. Standard 1076-1993, IEEE Standard VHDL Language Reference Manual, 1994.

[IEEE 1994]
IEEE Press. *IEEE Standards Collection, Software Engineerin*g, 1994.

[IEEE1029.1 1992]
IEEE Press. Standard 1029.1-1991, IEEE Standard for waveform and vector exchange (WAVES) VHDL, September 1992.

[IEEE1076.6 1998]
IEEE Press. Standard 1076.6.-1998, IEEE Draft Standard for VHDL Register Transfer Level (RTL) Synthesis, March 1998.

[IEEE-D&T 2001]
IEEE Press. *Formal Verification of Commercials ICs*. IEEE Design & Test of Computers, v. 18, no. 4, New York, July-August 2001.

[IEEE729 1983]
IEEE Press. IEEE Standard Glossary of Software Engineering Terminology; ANSI/IEEE Standard 729-1983; 1983

[ITC'Benchmarks 1999]
International Test Conference benchmarks. Benchmarks from Politecnico di Torino, 1999.
« http://www.cad.polito.it/tools/#bench »

[Jaffar 1987]
J. Jaffar and J.-L. Lassez, *Constraint Logic Programming*, ACM 14th POPL 87, Munich, Germany, pp. 111-119, January 1987.

[Kalyanaraman 1993]
R. Kalyanaraman, *Behavioral Test Vector Generation in VHDL/WAVES Environment*, M.S. Thesis, 1993.

[Kappoor *et al.* 1994]
S. Kapoor, J. R. Armstrong, S. R. Rao, *An Automatic Test Bench Generation System*, Proc. VHDL International Users Forum, pp. 8-17, 1994.

[Korel 1990]
B. Korel, *Automated software test data generation*. IEEE Transactions on Software Engineering, Vol. 16, No 8, pp. 870—879, 1990.

[Lajolo 2000]
M. Lajolo, L. Lavagno, M. Rebaudengo, M. Sonza Reorda, M. Violante, *Behavioral Level Test Vector Generation for Systems-on-a-Chip Designs,* IEEE International High Level Design Validation Workshop, The Claremont Resort & Spa, Berkeley, California, pp. 21-26, November 8-10 2000.

[Landrault]
C. Landrault, *génération automatique de séquences de test*, Polycopié de cours-LIRM.
« http://www.mea.isim.univ-montp2.fr/POLYCOPS/ »

[Laprie 1992]
J.C. (Ed.) Laprie, *Dependability : Basic Concepts and Terminology*. Springer-Verlag L.N.C.S, 1992.

[McCabe 1976]
T.J. McCabe, *A Complexity Measure*, IEEE Trans. Software Testing Engineering, N°2, pp. 308-320, December 1976.

[McCabe 1982]
T. McCabe, *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric,* U.S. Department of Commerce/National Bureau of Standards (U.S.), NBS Special Publication 500-99, December 1982.

[Offutt et al. 1997]
J. Offutt, Z. Jin, and J. Pan, *The Dynamic Domain Reduction Procedure for Test Data Generation*, Software Practice and Experience, Vol. 29, No. 2, pp. 167-193, January 1997.

[Paoli 1998a]
C. Paoli et J-F. Santucci, *Validation of VHDL Description : A Software Engineering Approach*, communication, actes du « 7th BEhavioraL deSIGN methodologies for digital systems Workshop (BELSIGN'98) », 7-8 mai, Twente, Pays-Bas, 4 p, 1998.
« http://www.die.upm.es/research/8thBELSIGN.html »

[Paoli 1998b]
C. Paoli et J-F. Santucci, *Application of Software Metrics for Validating VHDL Descriptions of Digital Circuits*, communication, actes du « Advanced Technology Workshop (ATW'98) », 19-20 mai, Ajaccio, 5 p, 1998.
« http://spe.univ-corse.fr/ATW.htm »

[Paoli 1999a]
C. Paoli et J-F. Santucci, *Validation of Behavioral VHDL Descriptions Using Software Engineering Concepts*, poster, actes de la « IEEE Electronic Circuits and Systems Conference (ECS'99) », 6-8 septembre, Bratislava, Slovaquie, pp. 215-218, 1999.
« http://www.elf.stuba.sk/~ecs2001/ »

[Paoli 1999b]
C. Paoli et J-F. Santucci, *High Level Test Benches Generation for the Validation of VHDL Models of Microelectronic Systems*, communication, actes du « IEEE International Workshop on System Test and Diagnosis workshop (IWSTD'99) », 30 septembre - 1 octobre, Atlantic City, New Jersey, USA, 4 p, 1999.
« http://www.itctestweek.org/itc2000.html#work »

[Paoli 1999c]
C. Paoli et J-F. Santucci, *Data generation from VHDL behavioral descriptions,* communication, actes du « Advanced Technology Workshop (ATW'99) », 10-11 juin, Ajaccio, 4 p, 1999.
« http://spe.univ-corse.fr/ATW.htm »

[Paoli 2000a]
C. Paoli, M-L. Nivet et J-F. Santucci, *Use of constraint solving in order to generate test vectors for behavioral validation*, communication, actes du « IEEE International High Level Design Validation and Test Workshop (HLDVT'00) », 8-10 Novembre , Berkeley, Californie, USA, pp. 15-20, 2000.
« http://www-cse.ucsd.edu/groups/hldvt/ »

[Paoli 2000b]
C. Paoli, M.L. Nivet et J.F. Santucci, *Test Vectors Generation using Path Selection and Constraint Solving for the Validation of Behavioral VHDL Design*, Communication, actes du « IEEE International Workshop on System Test and Diagnosis workshop (IWSTD'00) », 5-6 octobre, Atlantic City, New Jersey, USA, 5 p, 2000.
« http://www.itctestweek.org/itc2000.html#work »

[Paoli 2000c]
C. Paoli, P-A. Bisgambiglia, D. Federici et J-F. Santucci, *High Level Validation of VHDL Description using Behavioral Fault Simulation*, poster, actes du congrès international Environnement et Identité en Méditerranée, programme de coopération scientifique Interreg II, 13-16 juin, Corté, 2000.
« http://www.univ-corse.fr/recherche/colloques/Interreg/Interreg_fr.html »

[Paoli 2000d]
C. Paoli, M-L. Nivet et J-F. Santucci, *Behavioral VHDL Test Benches Generation*, communication, actes du « Advanced Technology Workshop (ATW'00) », 14-16 Juin, Ajaccio, 2000.
« http://spe.univ-corse.fr/ATW.htm »

[Paoli 2002]
C. Paoli, M-L. Nivet, J-F. Santucci et T. Campana, *Path-Oriented Test Data Generation of Behavioral VHDL Description,* IEEE International Workshop on Electronic Design, Test & Applications The Chateau on the Park Hotel Christchurch, New Zealand, January 29-31 2002, à paraître.

[Parissis 1996]
I. Parissis, *Test de logiciels synchrones spécifiés en LUSTRE*, Thèse de Doctorat, Université de Grenoble I, septembre 1996.

[Péraire 1998]
C. Péraire, *Le test de logiciels*, Polycopié de cours- EPFL, mai 1998.
« http://lglwww.epfl.ch/Team/CP/cp_cv.html »

[Pla 1993]
V. Pla, *Générateur de test comportemental*, Thèse de Doctorat, Université de Montpellier II, décembre 1993.

[Poole 1995]
J. Poole, *A Method to Determine a Basis Set of Paths to Perform Program Testing*, NISTIR 5737, November 1995.
« http://hissa.ncsl.nist.gov/publications/nistir5737/index.html »

[Reintjes 1988]
P. B. Reintjes, *A VLSI Design Environment in PROLOG*, ICLP/SLP, pp. 70-81, 1988.
« http://www.logic-programming.org/people/Reintjes_Peter/paris.html »

[Rouzeyre]
B. Rouzeyre, synthèse architecturale, Polycopié de cours - LIRM.
« http://www.mea.isim.univ-montp2.fr/POLYCOPS/ »

[Roy *et al.* 1992]
J. Roy and R. Vemuri, *Appropriate Usage of VHDL : The Synthesis Point of View*, Technical Memo-TM-DDE-89-08, 1992.
« http://www.ececs.uc.edu/~ddel/projects/dss/subset_doc »

[Rudnick 1998]
E.M. Rudnick, R. Vietti, A. Ellis, F. Corno, P. Prinetto, and M. Sonza Reorda, *Fast Sequential Circuit Test Generation Using High-Level and Gate-Level Techniques,* Proc. DATE, pp. 570-576, 1998.

[Santucci 1993]
J. F. Santucci, A. Courbois, and N. Giambiasi, *Behavioral testing of digital circuits*. Journal of Microelectronic Systems Integration, Vol. 1, No. 1, pp. 55-77, 1993.

[Santucci 1999]
J-F. Santucci et C. Paoli, *High level test bench generation using software engineering concepts*, participation au panel n°6 : « Test Benchmarks » de la « 30th IEEE International Test Conference (ITC'99) », 28-30 septembre, Atlantic City, New Jersey, USA, 1999.
« http://www.itctestweek.org/ »

[Shen *et al.* 1999]
J. Shen and J. A. Abraham, *Verification of Processor Microarchitectures,* Proc. IEEE VLSI Test Symposium, pp. 189-194, 1999.

[Sonza 1999]
M. Sonza Reorda, *High-level ATPG: a real topic or an academic amusement?* IEEE International Test Conference, Atlantic City (USA), Poster Session, pp. 1118, September 1999.

[Tasiran 2001]
S. Tasiran, K. Keutzer, C*overage metrics for functional validation of hardware designs,* IEEE Design & Test of computers, Vol. 18, No 4, pp. 36-45**,** July-August 2001.

[Thirunarayan 1997]
K. Thirunarayan, P. B. Reintjes, R. L. Ewing, *A VHDL 1076.1 Parser and Pretty-Printer in SWI-Prolog*, Tech. Report, 1997.
« http://www.cs.wright.edu/people/faculty/tkprasad/VHDL/VHDL-AMS/START.html »

[Van Hentenryck 1989]
P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, The MIT Press, 1989.

[Vemuri *et al.* 1995]
R. Vemuri, R. Kalyanaraman, *Design Verification tests from Behavioral VHDL Programs*, IEEE Trans. on VLSI, Vol. 3, N° 2, pp. 201-214, June 1995.

[Walsh *et al.* 1996]
P. Walsh, D. Hoffman. *Automated Behavioral Testing of VHDL Components*. Proc. Canadian Conference on Electrical and Computer Engineering, Calgary, Alberta, Canada, May 1996.

[Watson 1996a]
A.H. Watson, T.J. McCabe, *Structured Testing: A testing Methodology using the cyclomatic Complexity Metric*, NIST Special Publication 500-235, August 1996.
« http://hissa.ncsl.nist.gov/HHRFdata/Artifacts/ITLdoc/235/title.htm »

[Watson 1996b]
A. H. Watson, *Structured Testing: Analysis and Extensions*, TR-528-96, 171 p, November 1996.
« http://citeseer.nj.nec.com/watson96structured.html »

[Y1Q2_N°1 2000]
M-L. Nivet, C. Paoli et J-F. Santucci, *Definition of the internal model based on graph structures representing VHDL behavioral descriptions*, Université de Corse, 11 p, septembre 2000.

[Y1Q2_N°2 2000]
J-F. Santucci, C. Paoli et M-L. Nivet, *Algorithms allowing to automatically generate an internal model from a VHDL behavioral description*, Université de Corse, 13 p, septembre 2000.

[Y1Q2_N°3 2000]
C. Paoli, M-L. Nivet et J-F. Santucci, *Definition of the algorithm alllowing the minimum set of path to be generated*, Université de Corse, 10 p, septembre 2000.

[Y1Q3_N°4 2000]
C. Paoli M-L. Nivet et J-F. Santucci, *Software Implementation of the automatic generation of the internal model*, Université de Corse, 11 p, septembre 2000.

[Y1Q3_N°5 2000]
C. Paoli, M-L. Nivet et J-F. Santucci, *Software implementation of the minimum set of paths generation - S2*, Université de Corse, 10 p, septembre 2000.

[Y1Q4_N°6 2001]
C. Paoli, M-L. Nivet et J-F. Santucci, T. Campana, *Validation of the Software S1 described in the Technical Report N°4 on a set of benchmark VHDL descriptions,* Université de Corse, 33 p, juin 2001.

[Y1Q4_N°7 2001]
C. Paoli, M-L. Nivet et J-F. Santucci, T. Campana, *Validation of the Software S2 described in the Technical Report N°5,* Université de Corse, 11 p, juin 2001.

TABLE DES FIGURES

Figure 1-1. Processus de conception d'un circuit complexe. 2
Figure 1-2. Aperçu de notre approche. 5
Figure 2-1. Diagramme en Y. 9
Figure 2-2. Flot de conception de haut niveau. 11
Figure 2-3. Un test-bench. 13
Figure 2-4. Configuration de test-bench. 14
Figure 3-1: Programme correct. 20
Figure 3-2: Erreur de programmation. 20
Figure 3-3: Erreur de spécification. 21
Figure 3-4: Erreur d'omission. 21
Figure 3-5. Classification des techniques de validation. 23
Figure 3-6: Programme mutant. 25
Figure 3-7. Les différents types de nœud d'un GFC. 28
Figure 3-8. Un programme et son GFC. 30
Figure 3-9. Relation d'inclusion entre les critères. 31
Figure 3-10. Un programme et son GFC avec deux décisions identiques. 32
Figure 3-11. Algorithme de Poole. 33
Figure 3-12. Application de l'algorithme de Poole. 34
Figure 3-13. Trace de l'algorithme de Poole. 34
Figure 3-14. Un programme et son GFC. 37
Figure 3-15. Vue du domaine avec contraintes et un point de coupure. 38
Figure 4-1. Illustration d'une description VHDL de type comportementale. 44
Figure 4-2. L'affectation des signaux. 47
Figure 4-3. Programme VHDL et table de simulation. 49
Figure 4-4. Un modèle pour l'instruction architecture. 53
Figure 4-5. Un modèle pour l'instruction process. 54
Figure 4-6. Les instructions d'affectations. 55
Figure 4-7: Un modèle pour l'instruction *if_then_else*. 56
Figure 4-8: Un modèle pour l'instruction `case`. 57
Figure 4-9: Un modèle pour la structure de boucle while. 58
Figure 4-10: Un modèle pour la structure de boucle for. 59

Figure 4-11. GFC d'un programme VHDL. 60
Figure 4-12. Exemple de GFC avec des chemins à ordonnancer. 63
Figure 4-13. Flot de dépendance. 65
Figure 4-14. GFC et GdD d'un programme VHDL. 66
Figure 4-15. Chemin à ordonnancer dans la liste d'ordonnancement. 69
Figure 4-16. GFC d'un programme VHDL. 70
Figure 4-17. Graphe de modélisation de process. 71
Figure 4-18. Un CFG simplifié et sa base de chemins. 71
Figure 4-19. Cas de figure pour les chemins à modifier. 74
Figure 4-20. Cas d'utilisation du GdD pour un chemin à modifier. 76
Figure 5-1. Méthodologie pour la génération des séquences de test. 84
Figure 5-2. La production de la base. 85
Figure 5-3. Analyse des chemins. 86
Figure 5-4. Organigramme pour la détection des chemins à modifier. 88
Figure 5-5. Génération de la liste des chemins 90
Figure 5-6. L'extraction des vecteurs de test. 91
Figure 5-7. Exemple de génération de contraintes. 98
Figure 5-8. Production de la séquence de test. 100
Figure 5-9. Organigramme pour la détection des chemins à ordonnancer et pour la génération de la liste d'ordonnancement. 102
Figure 5-10. Méthode « générer et tester ». 104
Figure 6-1. Architecture logicielle de GENESI. 109
Figure 6-2. Description du lexeur-parseur. 110
Figure 6-3. Représentation d'un AST. 111
Figure 6-4. Organisation du lexer-parser modifié. 111
Figure 6-5. Construction des graphes. 117
Figure 6-6. Production et analyse des chemins. 118
Figure 6-7. Génération et résolution des contraintes. 119
Figure 6-8. Un cas d'utilisation. 122
Figure 6-9. Génération des contraintes. 123
Figure 6-10. Génération des contraintes. 124
Figure 7-1. Couplage de GENESI à BFS. 134

LISTE DES TABLEAUX

Tableau 3-1. Matrice de construction des chemins. 30
Tableau 3-2. Domaines de chaque variable d'entrée. 39
Tableau 3. Solution pour le chemin Ch3. 63
Tableau 4. Table de simulation. 64
Tableau 5. Création d'une liste d'ordonnancement. 68
Tableau 6. Table de simulation. 72
Tableau 6-1. Résultats sur les benchmarks ITC'99. 126

Annexe 1

Grammaire du sous ensemble VHDL

0. Description Syntactique

Ce document fournit un résumé de la syntaxe pour le sous-ensemble VHDL comportementale que nous utilisons. Nous suivons l'ordre d'apparence rencontré dans le manuel de référence du langage VHDL. La forme d'une description VHDL est décrite au moyen d'une syntaxe sans contexte, en utilisant une variante simple de la forme de Backus Naur (BNF); en particulier :

a. Les mots en minuscule en police Arial, dont une partie contenant des tiret bas, sont employés pour dénoter des catégories syntactiques, par exemple :

formal_port_list

Toutes les fois que le nom d'une catégorie syntactique est utilisé, indépendamment des règles de syntaxe, des espaces prennent la place des tiret bas (ainsi, " formal port list " apparaîtrait dans la description narrative en se rapportant à la catégorie syntactique ci-dessus).

b. Des mots de caractères gras sont employés pour dénoter des mots réservés, par exemple :

array

Des mots réservés doivent être utilisés seulement dans les endroits indiqués par la syntaxe.

c. Une *production* se compose d'un *côté gauche*, du symbole "::="(qui est lu comme " peut être substitué par "), et d'un *côté droit*. Le *côté gauche* d'une *production* est toujours une catégorie syntactique; le *côté droit* est une règle de remplacement. La signification d'une *production* est une règle de remplacement textuel: n'importe quelle occurrence du *côté gauche* peut être remplacée par un exemple du *côté droit*.

d. Une barre verticale sépare les éléments alternatifs du côté droit d'une production à moins qu'elle se produise juste après une accolade ouvrante, dans ce cas elle représente elle-même :

letter_or_digit ::= letter | digit

choices ::= choice { | choice }

En premier lieu, une occurrence de " letter_or_digit " peut être remplacée soit par " letter " soit par " digit " Dans le deuxième cas, des " choices " peuvent être remplacés par une liste de " choice, " séparé par les barres verticales (voir l'élément f pour la signification des accolades).

e. Les crochets entourent les éléments facultatifs du côté droit d'une production; ainsi les deux productions suivantes sont équivalentes :

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

return_statement ::= **return** ; | **return** expression ;

f. Les accolades entourent un élément répété ou des éléments du côté droit d'une production. Les éléments peuvent apparaître zéro fois ou plus; les répétitions se produisent de gauche à droite comme avec une règle récursive gauche équivalente. Ainsi, les deux productions suivantes sont équivalentes :

term ::= factor { multiplying_operator factor }

term ::= factor | term multiplying_operator factor

g. Si le nom de n'importe quelle catégorie syntactique commence par une partie en italique, il est équivalent au nom de catégorie sans partie en italique. La partie imprimée en italique est destinée pour à une information sémantique. Par exemple, le *type*_name et le *subtype*_name sont tous deux syntactiquement équivalents à name seul.

h. Le terme simple_name est utilisé pour n'importe quelle occurrence d'un identificateur qui dénote déjà une certaine entité déclarée.

1. Design entities and configurations

1.1 Entity declaration

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

1.1.1 Entity header

entity_header ::= [*formal*_port_clause]
port_clause ::= **port** (port_list) ;

1.1.1.2. Ports

port_list ::= *port*_interface_list

1.2 Architecture bodies

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

1.2.1. Architecture declarative part

architecture_declarative_part ::=
{ block_declarative_item }

block_declarative_item ::=
type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration

1.2.2. Architecture statement part

architecture_statement_part ::=
{ concurrent_statement }

Les divers concurrent_statement sont décrits dans la section 9. Concurrent statements.

2. Subprograms and Packages

3. Types

Cette section décrit les diverses catégories des types qui sont fournis par le langage aussi bien que ces types spécifiques qui sont prédéfinis. Les déclarations de tous les types prédéfinis sont contenues dans la package STANDARD, dont la déclaration apparaît dans la section 14 du manuel de référence de VHDL.
Nous considérons dans notre sous ensemble VHDL deux classes de types :

1. Les *scalar types* qui sont des types d'*integer*, et des types définis par une énumération de leurs valeurs; les valeurs de ces types n'ont aucun élément.
2. Les *composite types* sont des *array*; les valeurs de ces types se composent des valeurs d'élément.

L'ensemble de valeurs possibles pour un objet d'un type donné peut sous réserve d'une condition qui s'appelle une *constraint* (le cas où la *constraint* impose aucune restriction est également inclus); on dit qu'une valeur satisfait une *constraint* si elle satisfait la condition correspondante. Un sous-type est un type avec une *constraint*. On dit qu'une valeur appartient

à un sous-type d'un type indiqué si elle appartient au type et satisfait la *constraint*; le type donné s'appelle le type de base du sous-type. Un type est un sous-type de lui-même; un tel sous-type est dit sans *constraint* (il correspond à une condition qui n'impose aucune restriction). Le type de base d'un type est le type lui-même.
On ne prend pas en compte dans notre sous ensemble VHDL deux classes de types : *Access* types et *File* types. Concernant le scalar type, les *physical types*, et *floating point types* ne sont pas accepté.

3.1. Scalar types

Dans notre sous ensemble VHDL, les *scalar* types contiennent les *enumeration types*, les *integer types*. On ne prend pas en compte les *float* et les *physical*.

```
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
```

Un *range* indique un sous-ensemble de valeurs de *scalar type*. Un *range* est dit un *nul range* si le sous-ensemble indiqué est vide

3.1.1 Enumeration types

```
enumeration_type_definition ::=
        ( enumeration_literal { , enumeration_literal } )

enumeration_literal ::=  identifier | character_literal
```

3.1.2. Integer types

```
integer_type_definition ::= range_constraint
```

3.2. Composite types

Dans notre sous ensemble VHDL on ne traite pas les *record types*.

```
composite_type_definition ::=
        array_type_definition
```

3.2.1. Array types

array_type_definition ::=
unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::=
array (index_subtype_definition { , index_subtype_definition })
of *element*_subtype_indication

constrained_array_definition ::=
array index_constraint **of** *element*_subtype_indication

index_subtype_definition ::= type_mark **range** <>

index_constraint ::= (discrete_range { , discrete_range })

discrete_range ::= *discrete*_subtype_indication | range

```
Examples:
--Examples of constrained array declarations:

     type MY_WORD is array (0 to 31) of BIT ;
          --  A memory word type with an ascending range.

     type DATA_IN is array (7 downto 0) of FIVE_LEVEL_LOGIC ;
          --  An input port type with a descending range.

--Example of unconstrained array declarations:

     type MEMORY is array (INTEGER range <>) of MY_WORD ;
          --  A memory array type.

--Examples of array object declarations:

     signal DATA_LINE : DATA_IN ;
          --  Defines a data input line.

     variable MY_MEMORY : MEMORY (0 to 2^(n-1)) ;
          --  Defines a memory of 2^n 32-bit words.
```

3.2.2. Record types

On ne prend pas en compte le type record.

identifier_list ::= identifier { , identifier }

4. Declarations

declaration ::=

type_declaration
| subtype_declaration
| object_declaration
| interface_declaration
| entity_declaration

4.1 Type declarations

type_declaration ::=
full_type_declaration

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

type_definition ::=
scalar_type_definition
| composite_type_definition

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

4.3. Object

4.3.1. Object declarations

object_declaration ::=
constant_declaration
| signal_declaration
| variable_declaration

4.3.1.1 Constant declarations

constant_declaration ::=

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

4.3.1.2 Signal declarations

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

4.3.1.3 Variable declarations

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

4.3.2 Interface declarations

interface_declaration ::=
interface_signal_declaration

interface_signal_declaration ::=
[**signal**] identifier_list **:** [mode] subtype_indication [**:=**
static_expression]

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

Remarque: mode est optionnel, en cas d'absence le signal est considéré en mode in.

4.3.2.1 Interface lists

interface_list ::=
interface_element { ; interface_element }

interface_element ::=
interface_declaration

4.3.2.2 Association lists

association_list ::=
association_element { , association_element }

association_element ::=
actual_part

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

actual_designator ::=
expression
| *signal*_name

| *variable*_name

5. Specifications

On ne prend pas en compte les *specification* dans notre sous-ensemble VHDL.

6. Names

Les règles applicables aux diverses formes du nom sont décrites dans cette section.

6.1 Names

```
name ::=
          simple_name
        | indexed_name
        | attribute_name

prefix ::=
        name
```

6.2 Simple names

```
simple_name ::= identifier
```

6.2 Indexed names

```
indexed_name ::= prefix ( expression { , expression } )
```

6.6 Attribute names

```
attribute_name ::=
        prefix ' attribute_designator [ ( expression ) ]

attribute_designator ::=  attribute_simple_name
```

7. Expressions

Les règles applicables aux différentes formes de l'expression, et à leur évaluation, sont données dans cette section.

7.1 Expressions

Une expression est une formule qui définit le calcul d'une valeur.

```
expression ::=
          relation { and relation }
        | relation { or relation }
        | relation { xor relation }
        | relation [ nand relation ]
        | relation [ nor relation ]
        | relation { xnor relation }

relation ::=
        shift_expression [ relational_operator shift_expression ]

shift_expression ::=
        simple_expression [ shift_operator simple_expression ]

simple_expression ::=
        [ sign ] term { adding_operator term }

term ::=
        factor { multiplying_operator factor }

factor ::=
          primary
        | not primary

primary ::=
          name
        | literal
        | aggregate
        | fonction_call
        | ( expression )
```

7.2. Operators

Les opérateurs qui peuvent être utilisés dans les expressions sont définis ci-dessous. Chaque opérateur appartient à une classe d'opérateurs, qui ont le même niveau de priorité; les classes d'opérateurs sont énumérées par ordre de priorité croissante.

logical_operator	::=	and	|	or	|	nand	|	nor	|	xor	|	xnor
relational_operator	::=	=	|	/=	|	<	|	<=	|	>	|	>=
shift_operator	::=	sll	|	srl	|	sla	|	sra	|	rol	|	ror
adding_operator	::=	+	|	-	|	&						
sign	::=	+	|	-								
mutiplying_operator	::=	*	|	/	|	mod	|	rem				
micellaneous_operator	::=	**	|	abs	|	not						

7.3. Operand

7.3.1. Literals

Un *literal* est soit un *numeric literal*, soit une *enumeration literal*.

```
literal ::=
        numeric_literal | enumeration_literal

numeric_literal ::=
        abstract_literal
```

7.3.2. Aggregates

```
aggregate ::=
        ( element_association { , element_association } )

element_association ::=
        [ choices => ] expression

choices ::=  choice { | choice }

choice ::=
          simple_expression
        | discrete_range
        | element_simple_name
        | others
```

7.3.3. Functions call

```
function_call ::=
```

*function*_name [(actual_parameter_part)]

actual_parameter_part ::= *parameter*_association_list

8. Sequential statements

Les diverses formes des *sequential statements* sont décrites dans cette section.

sequence_of_statements ::=
{ sequential_statement }

sequential_statement ::=
wait_statement
| signal_assignment_statement
| variable_assignment_statement
| if_statement
| case_statement
| loop_statement
| null_statement

8.1 Wait statement

On ne prend pas en compte le **wait** dans notre sous ensemble.
On suppose que l'on transforme chaque **wait** trouvé par son équivalent. On ne peut avoir qu'une instruction **wait** par « process ». Quand on a un « wait on sensitivity_clause », on place sensitivity_clause dans la liste sensible du « process ». Quand on a un « wait until », on place le « signal » de la condition_clause dans la liste sensible du « process » et on place à l'endroit du « wait until » l'instruction « if (signal'event and condition_clause). Quand on a un « wait on sensitivity_clause until condition_clause », on place sensitivity_clause dans la liste sensible du « process » et on place à l'endroit du « wait on – until - » l'instruction « if (condition_clause). Comme la liste sensible existe, on ne rajoute pas le signal de la condition_clause dans la liste sensible du « process ».

wait_statement ::=
[label :] **wait** [sensitivity_clause] [condition_clause];

sensitivity_clause ::= **on** sensitivity_list

sensitivity_list ::= *signal*_name { , *signal*_name }

condition_clause ::= **until** condition

condition ::= *boolean*_expression

8.4 Signal assignment statement

```
signal_assignment_statement ::=
        [ label : ] target <= waveform ;

sensitivity_clause ::=  on sensitivity_list

target ::= name | aggregate

waveform ::= waveform_element
```

8.4.1. Waveform element

```
waveform_element ::= value_expression
```

8.5. Variable assignment statement

```
variable_assignment_statement ::=
        [ label : ] target := expression ;
```

8.7. If statement

```
if_statement ::=
        [ if_label : ]
                if condition then
                        sequence_of_statements
                { elsif condition then
                        sequence_of_statements }
                [ else
                        sequence_of_statements ]
                end if [ if_label ] ;

condition ::= boolean_expression
```

8.8. Case statement

```
case_statement ::=
        [ case_label : ]
                case expression is
                        case_statement_alternative
                        { case_statement_alternative }
                end case [ case_label ] ;

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

8.9. Loop statement

Il y a trois types de « loop » dans VHDL: « for loop », « while loop » et « infinite loop ». on considère que le « for loop » et le « while loop ».

```
loop_statement ::=
        [ loop_label : ]
                [ iteration_scheme ] loop
                        sequence_of_statements
                end loop [ loop_label ] ;

iteration_scheme ::=
          while condition
        | for loop_parameter_specification

parameter_specification ::=
        identifier in discrete_range
```

8.13 Null statement

```
null_statement ::=
        [ label : ] null ;
```

9. Concurrent statements

```
concurrent_statement ::=
          process_statement
```

9.2 Process statement

```
process_statement ::=
        [ process_label : ]
                process [ ( sensitivity_list ) ] [ is ]
                        process_declarative_part
                begin
                        process_statement_part
                end process [ process_label ] ;
process_declarative_part ::=
        { process_declarative_item }

process_declarative_item ::=
          type_declaration
        | subtype_declaration
        | constant_declaration
        | variable_declaration

process_statement_part ::= { sequential_statement }
```

13. Lexical elements

13.1 Character set

basic_graphic_character ::=
upper_case_letter | digit | special_character |space_character

graphic_character ::=
basic_graphic_character | lower_case_letter | other_special_character

basic_character ::=
basic_graphic_character | format_effector

l'ensemble des *basic character* est suffisant pour écrire n'importe quelle description. Les *characters* inclues dans chacune des catégories des *basic graphic characters* sont définis comme suit:

Uppercase letters
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ

Digits
0 1 2 3 4 5 6 7 8 9

Special characters
" # & ' () * + , - . / : ; < = > [ ] _ |

The space characters
SPACE, NBSP

Les *characters* inclus dans chacune des catégories restantes des *graphic characters* sont définis comme suit :

Lowercase letters
a b c d e f g h i j k l m n o p q r s t u v w x y z ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ

Other special characters
! $ % @ ? \ ^ ` { } ~ ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ~ ® ¯ ° ± ² ³ ' µ ¶ · ¸ ¹ º » 1/4 1/2 3/4 ¿ × ÷ -

13.3. Identifiers

identifier ::= basic_identifier | extended_identifier

13.3.1. Basic identifiers

Un *basic identifier* est constitués seulement de *letters*, de *digits*, et de *underlines*.

basic_identifier ::=
letter { [underline] letter_or_digit }

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

13.3.2. Extended identifiers

Les *Extended identifiers* peuvent contenir n'importe quelle *graphic character*.

extended_identifier ::=
\ graphic_character { graphic_character } \

1.3.4. Abstract litteral

abstract_literal ::= decimal_literal | based_literal

13.4.1 Decimal literals

decimal_literal ::= integer [. integer] [exponent]

integer ::= digit { [underline] digit }

exponent ::= E [+] integer | E - integer

13.4.2 Based literals

based_literal ::=
base # based_integer [. based_integer] # [exponent]

base ::= integer

based_integer ::=
extended_digit { [underline] extended_digit }

extended_digit ::= digit | letter

13.5 Character literals

character_literal ::= ' graphic_character '

13.6 String literals

string_literal ::= " { graphic_character } "

13.7 Bit string literals

bit_string_literal ::= base_specifier " [bit_value] "

bit_value ::= extended_digit { [underline] extended_digit }

base_specifier ::= B | O | X

14. Predefined language environment

Cette section décrit les “predefined attributes” de VHDL.

14.1 Predefined Attributes

Les résultats possible retourné par les attributs sont : une valeur, une fonction, un signal, un type ou un range.
Dans notre sous-ensemble VHDL, on ne prend en compte que l’attribut ‘EVENT.(synthétisable). Dans une description comportementale on ne prend pas en compte le « temps physique » dans les attribut, donc ‘QUIET(T) et ‘STABLE(T) n’ont pas de sens.

Annexe 2

Exemple de description VHDL “example.vhd” :

```
Entity example is
port ( in1,in2: in bit;
          out1: out bit);
end example;
architecture behavior of example is
signal a:bit;
begin
process1: process(in1, in2)
begin
      a <= in1 AND in2;
end process;
process2: process(a)
begin
if a = '0' then
      out1 <= '1';
end if;
end process;
end behavior;
```

Format LISP généré avec les règles du fichier : “VHDL97_write.pl” :

```
;;--  VHDL DESIGN UNIT #1
;;
(setq entity '(
    (name example)
    (opt_port_statement
        (interface_element
            (class port)
            (identifier_list in1 in2)
            (mode  in)
            (subtype_indication
                (subtype_name ())
                (subtype bit)
                (constraints ())
                (tolerance ())
            )
            (opt_assignment ()))
        (interface_element
            (class port)
            (identifier_list out1)
            (mode  out)
            (subtype_indication
                (subtype_name ())
                (subtype bit)
                (constraints ())
                (tolerance ())
            )
```

```
            (opt_assignment ()))

    )
))

;;--  VHDL DESIGN UNIT #2
;;
(setq architecture '(
    (name behavior)
    (entity_name example)
    (opt_declarative_items
        (object_declaration
            (class signal)
            (identifier_list a)
            (subtype_indication
                (subtype_name ())
                (subtype bit)
                (constraints ())
                (tolerance ())
            )
            (opt_assignment ())
        ))
    (concurrent_statements
        (process
            (opt_label process1)
            (opt_sensitivity_list in1 in2)
            (opt_declarative_items

            )
            (sequential_statements
                (<= a (and in1in2))
            )
        )
        (process
            (opt_label process2)
            (opt_sensitivity_list a)
            (opt_declarative_items

            )
            (sequential_statements
                (if
                    (= a 0)
                (then
                    (<= out1 1))
                )
            )
        )
    )
))
```

Validation de descriptions VHDL fondée sur des techniques issues du domaine du test de logiciels

Résumé :

L'objectif de cette dissertation est de développer une approche originale de validation de circuits digitaux complexes décrits dans le langage VHDL. Nous proposons de générer automatiquement, à partir d'une description VHDL comportemental au niveau algorithmique, les vecteurs de test à appliquer sur une description de niveau RTL.

Nous présentons d'abord la validation de descriptions VHDL au niveau algorithmique dans le contexte général du processus de conception de circuits complexes. Ce type de description étant similaire à un programme, nous explorons les techniques utilisées dans le domaine du test de logiciels, notamment celles basées sur un critère de couverture. Nous présentons le critère du test structuré, qui est fondé sur l'utilisation du graphe de flot de contrôle du programme sous test, et de la complexité cyclomatique de McCabe comme index du nombre de chemins à tester. Nous présentons également l'algorithme de Poole qui permet de générer cet ensemble de chemins.

Cependant, le langage VHDL possède des caractéristiques que l'on ne retrouve pas dans les langages de programmation traditionnels (notion de temps, interconnexion de « process » s'exécutant en parallèle, mécanisme de « retard delta »). Nous proposons donc une modélisation adéquate sous forme de graphes, permettant d'appliquer les techniques précédentes à des descriptions VHDL restreintes à un sous-ensemble prenant en compte un style de description algorithmique : un graphe de flot de contrôle, un graphe de modélisation de « process », un graphe de dépendance. Nous exposons ensuite une méthodologie pour la génération de vecteurs de test à partir des chemins générés depuis ces graphes : application de l'algorithme de Poole sur la base de la complexité cyclomatique, analyse et modification éventuelle des chemins, génération et résolution des contraintes, extraction des vecteurs de test.

L'approche est finalement illustrée par la réalisation du prototype logiciel GENESI qui nous a permis d'obtenir des résultats sur les « benchmarks ITC'99 ».

Validation of VHDL descriptions based on software testing techniques.

Abstract :

The objective of this work is to develop an original validation approach for complex digital systems in VHDL language. We propose to generate automatically from a behavioral VHDL description at the algorithmic level, the test vectors to be applied to a description at the register transfer level.

First, we present the validation of VHDL descriptions at the algorithmic level, in the general context of the design process of complex circuits. Since this type of description is similar to a software program, we explore the techniques used in the software testing field, in particular those using coverage criteria. We present the structured test criterion, which is based on the control flow graph of the program under test, and on the cyclomatic complexity of McCabe as an index of the number of paths to be tested. We also present the Poole's algorithm which allows to generate this set of paths.

However, VHDL language contains features which are not found in traditional software programming languages (concept of time, multiple communicating concurrent processes, delta delay mechanism). We thus propose an internal model in the form of graphs, allowing to apply the preceding techniques on VHDL descriptions: a control flow graph, a process model graph, a dependence graph. We restrict VHDL to a subset that is related to the algorithmic style. Then we expose a methodology for the test vectors generation in order to execute selected paths: application of the Poole's algorithm on the use of cyclomatic complexity, analysis and possible modification of the paths, constraints generation and solving, extraction of the test vectors.

The approach is finally illustrated by the implementation of the software prototype GENESI which enabled us to obtain results on ITC'99 benchmarks.

Discipline, Spécialité : Science pour l'Environnement, Informatique.

Mots Cles : VHDL, test-bench, validation, simulation, test structuré, complexité cyclomatique, programmation logique par contraintes.
Key Words : VHDL, test-bench, simulation based validation, structured testing, cyclomatic complexity, constraint logic programming.

SPE (Systèmes Physiques pour l'Environnement) UMR 6134
Quartier Grossetti, BP 52, 20250 Corte
http://spe.univ-corse.fr