THÈSE

pour l'obtention du titre de

DOCTEUR en INFORMATIQUE

à l'Université des Sciences et Technologies de Lille

par

Ammar ALJER

Co-design et Raffinement en B : BHDL Tool, plateforme pour la conception de composants numériques

---

Soutenue le : **21 décembre 2004** devant la Commission d'examen

Jury :

Président	:	**Jean-Luc DEKEYSER,**	Professeur à l'université de Lille 1
Rapporteurs	:	**Gilles GONCALVES,**	Professeur à l'université d'Artois
		Dominique MERY,	Professeur à l'université de Nancy 1
Examinateurs	:	**Przemyslaw BAKOWSKI,**	Professeur à l'université de Nantes
		Georges MARIANO,	Chargé de recherche à l'INRETS
Directeurs	:	**Philippe DEVIENNE,**	Chargé de recherche, CNRS-USTL
		Sophie TISON,	Professeur à l'université de Lille 1

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UMR 8022
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 – Télécopie : +33 (0)3 28 77 85 37 – email : `direction@lifl.fr`

Remerciements

Au terme de cette thèse pour laquelle j'ai passé une bonne partie de mon parcours scientifique et de ma jeunesse, je tiens à exprimer mes remerciements à chacune des personnes qui ont su - scientifiquement - contribuer, de près ou de loin, à l'élaboration de ce travail qui a été un effort de longue haleine :

- Monsieur Dominique MERY, Professeur à l'université de Nancy 1 et Monsieur Gilles GONCALVES, Professeur à l'université d'Artois pour l'intérêt qu'ils ont porté à mes travaux en acceptant d'être les rapporteurs de cette thèse.
- Monsieur Jean-Luc DEKEYSER pour avoir accepter d'en présider le jury.
- Monsieur Przemyslaw BAKOWSKI, professeur à l'Ecole polytechnique de l'université de Nantes pour avoir accepter d'être un examinateur attentif de mes travaux.
- Monsieur Gorges MARIANO pour m'avoir fait découvrir la Méthode B et pour avoir accepter être un membre de ce jury.
- Madame Sophie Tison et Monsieur Philippe Devienne pour leur accueil au sein de l'équipe STC, pour la direction et pour l'intérêt qu'ils m'ont apporté tout au long de cette thèse.
- Monsieur Jean-Louis Boulanger pour m'avoir assister dans mon exploration de la Méthode B et dans la mise au point de mon travail de programmation en B.
- Monsieur Jean Pierre STEEN et Monsieur Abdel Aziz BOUDLAL, pour leur contribution lors de la rédaction.

Je tiens, par ailleurs, à exprimer ma gratitude à toutes les personnes qui ont su - de par leur présence - m'accompagner, me soutenir, m'encourager durant ces longues années, lors des moments pénibles de doute ou de travail intense :

- Une fois de plus, Philippe pour son esprit humain, ses discussions scientifiques très riches et son aide très précieuse, à tous les niveaux ; Sophie, pour son soutien, sa patience et sa disponibilité.

- Mes anciens collègues de bureau, Isabelle SIMPLOT_RYL et Jean-Marc TALBOT qui n'ont pas hésité à répondre à mes diverses questions.
- Et tous mes amis, qu'ils soient collègues de Labo, ou étudiants de la résidence ; avec lesquels j'ai partagé une quelconque activité - scientifique, sociale ou autre ; tous ceux qui m'ont donné ce sentiment d'appartenir à une grande famille.

Je remercie particulièrement et du fond du coeur mes tendres parents dont la générosité admirable m'a permis de mener ce travail à bien. Très tôt, dès mon plus jeune âge, ils m'ont chaleureusement entouré et m'ont enseigné les premiers mots de la vie. A leur sujet, je n'en écrirai pas davantage, faute de trouver les mots qui puissent exprimer ce que je leur dois.

Enfin, je voudrais remercier cette force créatrice qui a créé l'univers et qui m'a donné vie et conscience pour apprécier sa beauté et en approcher les lois.

Table des matières

Table des figures

Introduction

L'architecture logicielle est un domaine de recherche très actif depuis une décennie. Son but principal est d'assister les développeurs de systèmes complexes (répartis, embarqués) à passer de l'énoncé des besoins ou exigences à l'implémentation. L'architecture logicielle décrit l'organisation d'un système par l'intermédiaire de ses composants et leurs interactions [81]. Il existe de nombreuses définitions de ce qu'est un composant et de ce qu'il n'est pas. Il y a deux perceptions principales [53], la première très proche des LOO[1] est plutôt utilisée par les programmeurs, les implémenteurs d'outils, donc assez près du système (ex API), la seconde par constraste a essayé d'abord d'établir des concepts et principes de haut niveau pour mettre l'accent sur la réutilisation et l'abstraction. Si les premiers se sont surtout focalisés sur les phases terminales de la conception, c'est à dire l'implémentation, l'intégration physique et l'exécution, les seconds se sont surtout intéressés aux toutes premières étapes de la conception au plus haut niveau d'abstraction possible.
La technologie à base de composants est bien-sûr un prolongement logique des langages de programmation dans leur souci d'aller vers un support beaucoup plus riche à l'exécution en matière d'analyse, d'interopérabilité, de chargement dynamique, vers un support plus riche conceptuellement en s'appuyant sur la notion de classe d'héritage, de polymorphismes, de méthodes et de notation à la *UML*. Pour tenter de s'abstraire le plus possible d'un modèle de programmation particulière, la communauté ***ADL***[2][41] a défini les objets et concepts de base sur lesquelles elle voulait s'appuyer :

- Les composants, leurs interactions et relations avec les autres composants.
- Architecture, configurations, connecteur, collage, propriétés, modèles hiérarchique, style, analyse statique, comportement.

Un des langages ADL le plus générique est sans doute le langage ***ACME*** développé à Berkeley par Garlan & all. [57] .

[1] Langages Orientés Objets.
[2] Architecture Description Language.

L'architecture matérielle a elle aussi connu une décennie plus tôt la même évolution vers plus d'abstraction et de modularité de conception [35]. À travers des travaux de standardisation et de modélisation, à la demande des industriels eux-mêmes. Le langage ***VHDL*** [8], normalisé en 1987, a été développé d'abord pour supporter la description et la simulation de circuits logiques [97]. Il a peu de temps après été utilisé pour supporter la synthèse, ce qui a permis de définir une nouvelle méthodologie de conception de type descendante qui est aujourd'hui très largement répandue. Cette méthodologie met l'accent sur le développement de modèles abstraits décrivant l'architecture du système à réaliser. A partir de cette synthèse, il est alors possible de produire une description plus détaillée satisfaisant aux contraintes physiques (surface, délai, consommation, etc.). Chaque modèle peut être validé en simulation avant ou après synthèse à différents niveaux d'abstractions et ainsi guider le concepteur lors de certains choix.

Il existe de **nombreuses analogies entre ces deux domaines**, analogies sur lesquelles cette thèse essaiera de s'appuyer de la façon la plus judicieuse (*ACME* 2002 vs ***VHDL-AMS*** 2001 [66]).
Un composant logiciel ou matériel est une unité qui masque son comportement à travers une interface bien définie. Les interfaces sont des points d'interaction du composant avec son environnement (c'est à dire : les autres composants du système) et modélise les services que le composant fournit ou sollicite. Les logiciels/circuits sont construits par création d'instances de types de composants et collage de leurs interfaces ensemble. La structure d'un programme *ADL* ou d'un circuit ***HDL***[3] est hiérarchique, c'est à dire : un arbre dans lequel les non feuilles sont des composants *composites* et les feuilles sont des composants *primitifs*. Un composant primitif n'a pas de structure et exprime un comportement. La structure d'un composant « composite » définit son comportement comme celui de ses sous-composants. Un composant composite encapsule toutes les interactions entre ses sous-composants sauf celles-ci sont définies explicitement dans son interface.
Le problème avec les standards ou les outils de standardisation est qu'il est souvent difficile de comprendre ce dont ils ont besoin, ce qu'ils fournissent et donc d'analyser leur puissance réelle [108]. Dans certains domaines d'applications, il est nécessaire de parler d'exigences autant que de besoins tant la sûreté des systèmes modélisés doit être garantie, ce qui oblige les développeurs à spécifier et tester le plus formellement possible le code [20, 49, 109]. En pratique, les standards actuels d'intégration se basent typiquement sur une

[3]Hardware Description Language.

documentation semi formelle ou informelle (ex : *UML*).
L'utilisation de modélisation réellement formelle peut permettre à la fois une meilleure compréhension pour le concepteur et la possibilité d'analyse automatique [109].
Les points importants dans cette approche sont :

1. la définition d'un modèle d'architecture formelle passe d'abord par une bonne sémantique des connecteurs : type-checking, model-checking protocoles (séquentielle, concurrence, non-déterminisme),
2. l'abstraction apporte une meilleure compréhension tant du point de vue intellectuel qu'automatique via les outils de model-checking,
3. la structuration de la spécification elle-même peut faciliter le partitionnement du travail et ceci aussi de façon incrémentielle pour permettre aussi un suivi depuis la spécification originelle. Plus tôt une erreur est découverte, moins elle coûte chèr à corriger.

Il existe une méthode de spécification formelle qui inclut elle-même la plupart des objets de base et de concepts utilisés dans les modèles de conception *ADL, HDL*. Il s'agit de la méthode ***B*** [48]. Cette méthode développée en parallèle de la communauté *ADL*, la rejoint et la précise sur plusieurs points. La méthode *B* pousse la notion de généricité et de spécialisation jusqu'à la notion de raffinement prouvable [107, 3, 32]. En d'autres termes, il devient possible de **prouver formellement** que chaque étape de conception dans une approche descendante n'est jamais en contradiction avec les spécifications plus abstraites déjà décrites. Elle intègre des outils de simulation formelle et propose un cycle de développement complet des aspects les plus abstraits de la conception à l'implémentation en langage *C* ou *Ada*.

Le travail que nous avons mené a consisté à mettre en correspondance la taxinomie des langages *ADL*, le modèle de développement utilisé dans le cadre des composants électroniques et la méthode de conception par raffinement basée sur la Méthode *B*. Ceci nous a permis de décrire ***BHDL*** Tool [12, 23].

Nous nous sommes basés aussi sur les critères que devrait satisfaire une bonne méthode formelle :

1. Utilisation immédiat : Pour encourager les développeurs à utiliser des méthodes, il faut que l'intérêt soit immédiat : interface agréable et facile, résultat produit en sortie compréhensible,
2. Gain progressif de productivité via une maîtrise progressive de l'outil,
3. Conception, Analyse et implémentation intégrées et intégrables,

4. Incrémentalité et Reutilisablilité,
5. Automaticité : si trop d'interactions sont nécessaires alors cela induit une compréhension approfondie de la représentation interne,
6. Détection d'erreurs et corrections,
7. Domaine d'application pertinent : savoir évaluer l'apport et les limites de la méthode,
8. Flexibilité (systèmes, notations, approches différentes).

Comment ce travail est présenté ?

Dans le premier chapitre on va présenter :

1. Les HDLs :
   - Les langages les plus connus dans cette domaine.
   - Leur méthodologie de conception.
   - Leur méthodes de vérification surtout de simulation.
   - La conception au niveau système.
2. Les ADLs :
   - Les langages les plus connues dans cette domaine.
   - La notion de composant, de point de vue ADL.
   - La conception Top-Down en ADL.
   - Les outils existants.
3. Les méthodes formelles :
   - La méthode B et le principe de raffinement.
   - L'utilisation des méthodes formelles pour la conception électronique.
4. Le co-design et ses langages.

Dans la deuxième chapitre on précise les objectifes de l'outil souhaité.

Dans la troisième chapitre on parle de l'implémentation de cet outil :
- VGUI, l'interface graphique de conception.
- VHDL et les composants emploiyés dans notre projet.
- Le code enrichi B-VHDL et les principes de traduction VHDL B.
- L'implémentation de ces principes en utilisant le compilateur ANTLR.

Le quatrième chapitre contient des examples.

Et le cinquième contient les perspectives.

Chapitre 1

Etat de l'Art

1.1 Langages de description de matériel

1.1.1 Introduction

Les ***HDLs***[1] sont des langages de programmation qui ont été développés et optimisés pour la conception et la modélisation de circuits numériques [97]. Ils combinent diverses méthodes de spécification pour la programmation de logiciels, des autres méthodes pour la modélisation **matérielle**, des langages de conception au niveau **système** [2], des langages de **test**, et des langages d'**interconnections**[3]. En tant que tel, un ***HDL*** inclut donc plusieurs dispositifs appropriés pour décrire le comportement des composants électroniques depuis les portes logiques simples jusqu'aux microprocesseurs et circuits intégrés.

Un *HDL* ne vise pas une *exécution* même s'il y a souvent un simulateur associé. Par exemple, un langage de description de matériel peut servir à faire de la *preuve formelle*[60], de la synthèse (le langage sert alors d'entrée à un outil *intelligent* pour la réalisation rapide de matériel), de la spécification et de la documentation. La simulation est tout de même un des aspects les plus importants d'un tel langage, juste avant la synthèse.

[1] Hardware Description Languages.
[2] Design Entry. Voir la description de HDL au niveau système et l'entrée de conception pages 22 et 22.
[3] Netlist. Voir page 7.

Des dispositifs de ***VHDL*** et de ***VerilogHDL***[4][21, 42] permettent de décrire, précisement, des aspects électriques du comportement de circuits, tels que la montée et la descand et la chute de signal, la latence au niveau des portes ou\et ses fonctions logiques. Les modèles résultants de simulation de *HDL* peuvent alors être employés comme modules pour simuler des circuits plus grands (employant des schémas ou des descriptions de *HDL* au niveau système).

Les langages de programmation et les HDLs

D'un côté, comme les langages de programmation de haut niveau, les *HDLs* permettent de conceptualiser des systèmes complexes et de les exprimer comme des programmes d'ordinateur. Ils permettent la capture du comportement des circuits électroniques, dans une structure complexe de conception, pour la synthèse automatique de circuit ou pour la simulation de système. ***L'entrée de conception*** « *design entry*fg est la première étape à exécuter dans les outils de conception assistée par ordinateur. Cette étape ressemble beaucoup à la conception de logiciel par un langage de programmation haut niveau comme *Pascal*, *Ada*, *C* ou *C++*. Ainsi les *HDLs* incluent des dispositifs pour les techniques de conception structurées. Ils offrent les moyens d'exprimer le contrôle, via des instructions puissantes et des types de données assez riches. Et à ce titre, certains langages *HDLs*, comme *VHDL*, se présentent comme des langages universels propres à modéliser n'importe quel système.

Mais, d'un autre côté, à **la différence** des langages de programmation classiques ou des langages de description d'architecture de logiciels, les *HDLs* permettent de **décrire des événements**, des opérations **parallèles ou concurrentes, synchrones ou asynchrones** en temps continu ou discret[5], en d'autres termes le comportement réel de composants électroniques. Le test est un aspect important qui diffère entre les deux familles de langages *HDL* et *ADL* [41]. Un *HDL*, comme *VHDL* ou *VerilogHDL*, peut décrire la performance d'un circuit par des jeux de test[6]. Ils décrivent pour des entrées données, à partir de la spécification du circuit, des valeurs de sorties attendues et, ainsi, vérifier le comportement du circuit réel. Cette étape est une partie importante et doit exister dans tout projet *HDL*. Les *jeux de test*

[4] *VHDL* et *VerilogHDL* sont deux langages distincts, ils sont les deux *HDLs* prédominant aujourd'hui. Voir page 5 et page 16.

[5] Voir page 9.

[6] Test Benches.

doivent être réalisés en parallèle avec les autres descriptions de conception.

Une **autre différence** importante entre un langage de programmation et un langage de description matériel, est celle qui sépare **la *vie* de leurs *briques de base*** : le sous-programme pour le langage algorithmique et le composant pour le langage de description *HDL* (même en *ADL*).

Le sous-programme est appelé à un certain moment, il remplit sa fonction. Puis il est, en quelque sorte, oublié (normalement il est téléchargé dans le *RAM* ; quand il se finit, il est effacé).
Le composant électronique existe en soi, car une description de matériel au niveau structurel, est « hors du temps ». Des langages comme *VHDL-AMS*[7] permettent de traiter à la fois des signaux digitaux et des signaux analogiques avec une fonction temps *continue*. De plus, les composants électroniques d'un système existent de façon statique et concurrente. Le temps n'intervient, en fait, que dans la partie flot de données (ou comportementale) d'un tel langage. Ceci permet d'écrire des choses comme :

```
A reçoit 0, puis 2 après 9 ns, puis 0 après 6 ns
```

Une **troisième différence** entre les deux catégories de langages provient des abstractions qui portent des valeurs. Une variable dans un langage de programmation a une durée de vie limitée à celle du sous-programme qui la contient, et porte successivement (ou peut porter) plusieurs valeurs. Un signal en *VHDL* existe tout au long de la simulation (si simulation il y a) et porte sur un échéancier la liste des valeurs qui ont été provoquées par son interaction avec les autres signaux, ou postées par les sous descriptions fonctionnelles ou *flot de données*. Il porte aussi, implicitement, la liste des valeurs qu'il a prises dans le passé (son historique). De façon générale, une affectation de variable (notée « := » en *VHDL*) a un effet limité dans le temps. Par exemple,

```
A := B ; -- instruction d'affectation de variable
```

signifie que la variable A prend, de façon immédiate, la valeur présente dans B lorsque l'on exécute l'instruction. Si la valeur de B change dans le futur, A garde la valeur de B, d'avant.

Il existe une autre instruction d'affectation : l'affectation de signal

```
A <= B ; instruction d'affectation de signal
```

[7] VHDL-Advanced and Mixed Signal. Voir page 5.

que l'on prononce « A reçoit B ». Contrairement à l'affectation de variable, ici la valeur de A reste attachée dans le temps à celle de B, c'est-à-dire que toute modification de B dans le temps entraînera une mise à jour de A aussi avec une certaine latence ou un filtrage (`transport, after`).

Les types de description de HDLs

Un langage de description de matériel propose souvent plusieurs niveaux de description. Ces niveaux, et même leurs définitions, peuvent varier d'un langage à un autre car une définition rigoureuse serait ardue et irait à l'encontre du caractère pragmatique de ces langages. Un *HDL* représentatif dans ce domaine, comme *VHDL*, peut faire référence à trois types de descriptions : structurelle, comportementale et de type *flot de données*. Il peut aussi être un mélange de ces trois types de base.

La description structurelle consiste à décrire le modèle par sa structure, c'est-à-dire par un ensemble d'éléments interconnectés, comme une association de composants. On parlera ici de modèle **combinatoire** [71]. Deux propriétés se déduisent immédiatement. Une telle description ne fait pas intervenir le temps. Il est nécessaire afin de simuler le circuit de décrire de façon comportementale ou flot de données toutes les feuilles (les éléments terminaux dans la hiérarchie de description) au moment de la simulation.

La description de niveau comportemental, dite aussi fonctionnelle, voire *algorithmique*, s'attache à décrire le fonctionnement d'un modèle sans se soucier d'un éventuel découpage proche de la réalisation, donc de la structure. La description prend souvent la forme d'un algorithme du genre de ceux que l'on utilise dans les langages de programmation classiques. Ce niveau de description, bien que présent dans les feuilles de la hiérarchie, peut aussi s'adresser aux concepteurs désirant modéliser un système avec un haut niveau d'abstraction. Le temps peut y est modélisé. S'attacher au comportement signifie se préoccuper de la fonctionnalité du modèle à décrire. C'est pourquoi le terme *description fonctionnelle* est souvent utilisé en lieu et place de description comportementale.

La description flot de données, dite aussi *booléenne*, exprime les flots de données sortants du modèle en fonction des entrants sans se préoccuper de sa structure. Nous détaillerons dans la suite ces derniers modèles de description.

Bien que *VHDL* et *VerilogHDL* soient actuellement les deux langages standards (approuvés et contrôlés par l'IEEE), *VHDL* est le langage le plus abouti et le plus utilisé. La standardisation de *VHDL* (et de *VerilogHDL*) est un facteur important pour son utilisation. La norme permet au concepteur d'accumuler ses expériences, d'utiliser l'expérience d'autres concepteurs et de profiter d'un nombre croissant d'outils qui supportent la norme.

1.1.2 VHDL

VHDL (VHSIC[8] Hardware Description Langage) est un langage de description de systèmes électroniques digitaux [8, 98, 91] (et analogiques\digitaux en *VHDL-AMS* [66]). C'est un langage moderne, puissant et présenté même comme universel. Ainsi il est connu pour ses avantages liés

- à son excellente lisibilité,
- à sa haute modularité [Voir page 54],
- à sa sécurité d'emploi
- et à la fiabilité de ses descriptions.

Ces caractéristiques découlent directement de notions modernes de langages de programmation, telles que les unités de compilation séparées, le typage fort ou la généricité. Mais *VHDL* profite aussi de l'expérience acquise dans les langages de description de matériel pour proposer (entre autres)

- une bonne prise en compte de **la concurrence**, une notion de temps solidement définie,
- la possibilité de mélanger à volonté des descriptions structurelles, comportementales et flot de données
- et celle de définir des fonctions de résolution (gestion de conflits) évoluées.

Ce langage a été développé au début des années 80 comme résultat inattendu du projet de recherche VHSIC financé par le département de la défense américain. Lors de ce programme, des chercheurs ont été confrontés à la tâche délicate de décrire des circuits composés d'un très grand nombre de composants et de contrôler les problèmes liés à la conception de tels circuits. Étant donné les outils de **conception de niveau porte**[9] disponibles à l'époque, est apparu clairement le besoin de méthodes de conception structurée et d'outils qui supportent de telles méthodes. Le résultat souhaité a nécessité les efforts conjoints de nombreuses sociétés rassemblées autour de

[8] Le V en VHDL est un raccourcie de VHSIC, qui est à son tour un raccourcie de « Very High Speed Integrated Circuits ».

[9] Gate-Level. Voir page 61.

INTERMETRICS, IBM et TEXAS INSTRUMENTS pour trouver un standard de la conception de circuit. Ce standard a abouti à la norme IEEE 1076B[10], approuvée le 10 décembre 1987. Depuis, *VHDL* est maintenu par le groupe américain *V-ASG*[11] qui est un sous-comité des DASS[12], eux-mêmes émanation de l'IEEE. Ensuite beaucoup d'efforts de standardisation ont été entrepris autour de *VHDL* et ont donné des résultats intéressants : on citera, par exemple, le paquetage de logique neuf états 1164, l'initiative VITAL de bibliothèques portables ou des paquetages standard pour la synthèse.

Les niveaux de l'abstraction en VHDL

VHDL permet de décrire plusieurs modèles de conception. Ces modèles peuvent exprimer plusieurs niveaux d'abstraction.

- **La description comportementale**
  Ce niveau est considéré par certains chercheurs, comme David Pellerin [97], comme le niveau le plus abstrait en *VHDL*.
  Ici, on décrit le circuit en expliquant son comportement par rapport au temps. Le concept du temps est l'élément critique de distinction entre les descriptions comportementales des circuits et les autres descriptions.

  En description comportementale, on utilise des **déclarations précises du temps**, les retards réels entre les événements relatifs (tels que les retards de la propagation dans des portes et dans des fils), ou de manière plus asynchrone par un ordre des opérations, exprimé séquentiellement (comme la description fonctionnelle d'une bascule « flip-flop »). Quand on écrit une description *VHDL* qui sera utilisée par les outils de synthèse, on peut employer un modèle comportemental, qui sera traduit automatiquement en termes de registres et de transfert entre registres « RTLs[13] » pour la synthèse.

  Il est peu probable, cependant, que cet outil de synthèse soit capable de créer, avec précision, le même comportement dans les circuits réels que dans le langage. Les outils de synthèse ignorent aujourd'hui des caractéristiques physiques de synchronisation, laissant les résultats réels de synchronisation à la merci de la technolo-

[10] Voir page 13.
[11] *VHDL* Analysis and Standardisation Group.
[12] Design Automation Standard Subcommittees.
[13] Register Transfer Level. Voir page 61.

gie cible ; cette imprécision s'accentue avec les changements d'échelle en terme d'intégration (nano-métrique et même submicronique). Comme dans un langage de programmation, on écrit un ou plusieurs petits programmes qui fonctionnent séquentiellement et communiquent entre eux par leurs interfaces. La seule différence entre le niveau comportemental *VHDL* et un langage de programmation est la plateforme fondamentale d'exécution : dans le cas du programme, c'est un certain logiciel d'exploitation fonctionnant sur une unité centrale de traitement ; dans le cas de *VHDL*, c'est le simulateur.

– **Le flot de données**

Ce niveau d'abstraction est bien connu par la plupart des concepteurs de circuit numérique. On décrit, à ce niveau, le circuit en précisant la façon dont le système déplace les données. Puisque les registres sont aujourd'hui au coeur de la plupart des systèmes numériques, la description de flot de données décrit **comment l'information est passée d'un registre à l'autre dans le circuit**. On peut utiliser la logique combinatoire pour décrire la partie combinatoire de conception à un niveau relativement haut. Puis, utiliser un outil de synthèse, pour préciser les détails de l'implémentation en forme de portes logiques. Mais, on doit être tout à fait précis en ce qui concerne le placement et le fonctionnement des registres dans le circuit complet.

– **La Description structurelle**

Elle est employée pour décrire un circuit en terme de **composants et de connections**. La structure peut être employée, aussi bien, pour créer une description de très bas niveau d'un circuit (telle qu'une description au niveau des transistors) ou que pour une description à un niveau très haut (telle qu'un schéma modulaire). On trouve cela au niveau des ports et des registres des circuits les plus élémentaires (comme les portes logiques ou les bascules « flip-flop »). C'est ce que l'on appelle une *netlist*.

Au niveau « haut » de conception, la description structurelle sert à segmenter la description abstraite ou fonctionnelle du système, en plusieurs pièces, qui sont plus maniables.

Une puce ne se conçoit plus sans interconnexion ni packaging. Les choix

technologiques affectent les performances (vitesse, puissance, fiabilité), ainsi que le coût de fabrication. Les dispositifs du niveau structurel en *VHDL* tels que les composants et les configurations sont très utiles pour contrôler la complexité. L'utilisation des composants peut, spectaculairement, améliorer la capacité de réutilisation des éléments de vos conceptions, et ils peuvent permettre de travailler en utilisant une approche de conception descendante.

Les unités de conception en VHDL

Un aspect important de la conception en *VHDL* est le **développement hiérarchique**. Cet aspect peut être obtenu grâce à la grande variété de méthodes de conception, ainsi qu'au concept d'**unités de conception**, concept qui favorise *VHDL* par rapport aux langages de programmation et à son rival principal, *Verilog*. Les unités de conception dans *VHDL* (ou les unités de bibliothèque) sont des segments de code *VHDL* qui peuvent être compilés séparément et stockés dans les bibliothèques.

Il y a, en fait, réellement cinq types d'unités de conception dans *VHDL* :

1. entités,
2. architectures,
3. paquets,
4. corps de paquet,
5. configurations.

Les grands constituants d'un modèle *VHDL-AMS*[14] sont :

1. Ouverture de bibliothèques (*LIBRARY*) et déclarations d'utilisation de leur contenu (*USE*),
2. Spécification d'entité (*ENTITY*) :
   - Paramètres génériques (*GENERIC*),
   - Ports de connexion (*PORT*) de type *SIGNAL, QUANTITY* ou *TERMINAL*,
   - Corps de l'entité : ensemble d'instructions passives.
3. Architecture de l'entité (*ARCHITECTURE*) :
   - Zone de déclarations,
   - Corps de l'architecture : description algorithmique basée sur des

[14] *VHDL* pour Analog and Mixed-Signal. Voir page 5.

(a) Instanciations de composants (hiérarchie),

(b) Instructions concurrentes (événements discrets),

(c) Instructions simultanées (temps continu).

L'entité

En *VHDL*, C'est une unité (indiqué par le mot-clé *ENTITY*) qui définit les **spécifications externes** d'un circuit. La description de conception minimum de *VHDL* doit inclure au moins une entité et une architecture correspondante.

Quand on écrit une déclaration d'entité, on doit fournir un **nom** unique pour cette entité et on doit déclarer l'**interface** complète du circuit. Cette interface décrit, à la fois, les paramètres ou constantes (*GENERIC*) et les ports d'entrées et de sorties (*PORT*). Chaque port, dans cette dernière liste, est défini par son nom, son type et par la direction du signal quand il traverse ce port.

Exemple

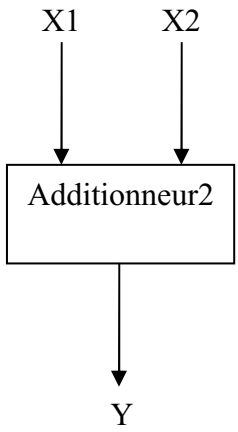

FIG. 1.1 – Additionneur.

```
ENTITY  Additioneur2  IS
    PORT(   X1, X2, :   IN   INTEGER ;
            Y       :   OUT  INTEGER
        );
END Additioneur2;
```

L'architecture.
En *VHDL*, c'est une unité (commençant par le mot-clé *ARCHITECTURE*) qui décrit la fonction et\ou la structure fondamentales d'un circuit. Chaque architecture, dans une conception, doit être **associée au nom d'une entité**.

VHDL permet de créer plusieurs architectures différentes pour chaque entité. Ce dispositif est particulièrement utile pour la simulation et quand le *spécificateur* de l'interface (*ENTITY*) n'est pas celui qui réalisera l'implémentation, c'est-à-dire la description structurelle (*ARCHITECTURE*).

Une déclaration d'architecture se compose d'abord, de déclarations (de paramètres, de signaux, et de ports), puis, du Corps de l'architecture. Ce dernier correspond à une description algorithmique de comportement de l'unité en se basant sur des instanciations d'autres entités (sous-composants) et d'instructions concurrentes ou parallèles.

Exemple

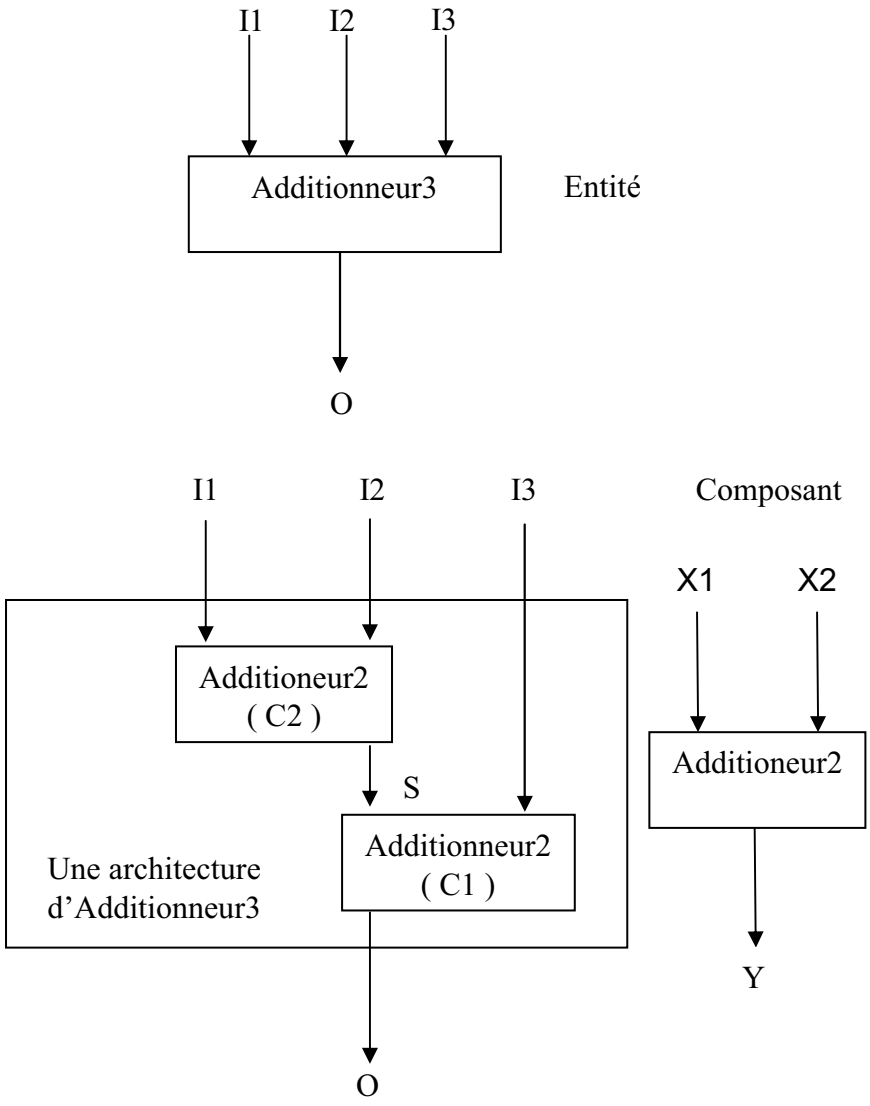

FIG. 1.2 – Additionneur à trois entrées.

```
----Additionneur à trois entrées. Deux niveaux d'abstraction.
----Niveau abstrait
Entity  Additioneur3  IS
  PORT(  I1, I2, I3 : IN INTEGER ;
          O : OUT INTEGER
      );
END Additioneur2;
----
----Niveau concret
Architecture ArAdd3 of Additioneur3;
--
  SIGNAL S : Integer;
--
  Component  Additioneur2  IS
  PORT(  X1, X2, : INTEGER ;
          Y : OUT INTEGER
      );
  END Additioneur2;
--
begin
  C1 : Additioneur2;
      PORT MAP (I1, I2, S);
  C2 : Additioneur2;
      PORT MAP (S, I3, O);
End ArAdd3;
----
```

Les paquetages et la déclaration des corps d paquetages.
Un paquetage de *VHDL* est identifié par le mot-clé *PACKAGE*. Il est employé pour rassembler des déclarations qui seront utilisées grâce à des clauses comme *LIBRARY* et *USE*. On peut les considerer comme des *boites noires* dont seule l'interface constitue la partie visible.

Un paquetage se compose de deux parties de base :
- une partie spécification (*PACKAGE*),
- et un corps (*PACKAGE BODY*) qui est facultatif.

Le rapport entre un paquetage et son corps est comparable au rapport entre une entité et son architecture correspondante, qui est ici unique. Ainsi, la déclaration fournit les informations requises pour employer les éléments qui y sont définis, tandis que, le comportement des procédures, fonctions,etc.

est indiquée dans des corps.

La partie spécification contient, de façon standard,
- des paramètres ou constantes,
- des déclarations de types et de sous-types,
- des déclarations de signaux,
- des déclarations de fonctions et de procédures,
- des déclarations de composants,
- des déclarations de fichiers.

Cette partie peut être rendu visible pour d'autres unités de conception par l'utilisation de terme *USE*.

Si cette partie contient des déclarations de sous-programmes (des fonctions ou des procédures) ou de constantes, dont la valeur n'est pas indiquée directement, alors un corps de paquetage est bien sûr nécessaire pour expliciter la définition des dites fonctions ou la valeur des dites constantes. Par contre, cette partie n'est pas accessible aux utilisateurs non autorisés.

La partie **configurations** permet de réaliser l'association effective entre l'instanciation d'un composant et un modèle de la base de données au moment de l'élaboration sans avoir en recompiler le modèle. Ces déclarations de configuration sont facultatives. En l'absence d'une déclaration de configuration, la norme de *VHDL* indique un ensemble de règles, qui vous fournissent une configuration par défaut. Par exemple, dans le cas où on a fourni plusieurs architectures pour une entité, la dernière architecture compilée aura la priorité et sera liée à l'entité. C'est un mécanisme très puissant, qui est souvent négligé par les développeurs.

Les normes de VHDL

- **La norme IEEE 1076**
  La première version publique disponible de *VHDL*, la version 7.2, a été diffusée en 1985. En 1986, IEEE[15] a proposé une normalisation du langage, normalisation finalisée en 1987 après des perfectionnements et des modifications substantiels. Ceci a été fait par une équipe associant des partenaires venus du monde industriel, gouvernemental et académique. La norme résultante, IEEE 1076, sert de base à chaque produit de simulation et de synthèse de *VHDL*, vendu aujourd'hui. Le langage a été enrichi et mis à jour pendant les années suivantes, et a eu, pour

[15]Institute of Electrical and Electronics Engineers.

résultat, la norme 1076-1993 aussi bien que diverses autres normes.

– **La norme IEEE 1164**
Bien que la norme 1076 de IEEE définisse le langage complet de *VHDL*, il y a des aspects du langage qui rendent difficile des descriptions de conception complètement portables (descriptions peuvent être simulées, identiquement, à l'aide des outils des fournisseurs différents). Le problème provient du fait que *VHDL* modélise beaucoup de types abstraits de données, tout en ne traitant pas le problème simple de comment caractériser les différents niveaux (ou valeurs) de signal, ou les conditions de simulation, qui sont utilisés souvent comme des valeurs inconnues et de grande impédance.

Après que la norme IEEE 1076-1987 ait été adoptée, les compagnies de simulateur ont commencé à enrichir *VHDL* par de nouveaux types de signal (typiquement, par l'utilisation des types énumérés, qui étaient déjà possibles, mais non standards) pour permettre à leurs clients de simuler, exactement, les circuits électroniques complexes. Ceci a posé des problèmes parce que les descriptions de conception, écrites à l'aide d'un simulateur, étaient souvent incompatibles avec d'autres environnements de simulation. *VHDL* devenait rapidement non standard.

Pour venir à bout du problème des types de données non standards, une autre norme a été créée, par un comité de IEEE. Cette norme, numéro 1164, définit un paquetage standards (un dispositif de *VHDL* qui permet à des déclarations, utilisées d'une façon générale, d'être rassemblées dans une bibliothèque externe) contenant des définitions pour un type de données standard de neuf valeurs. Ce type de données standard s'appelle std_logic ; et le paquetage de IEEE 1164 est désigné souvent sous le nom de paquetage standard de logique, ou de MVL9 (pour logique à valeurs multiples, neuf). Les normes IEEE 1076-1993 et IEEE 1164 forment, ensemble, la norme complète de *VHDL*. Ceci est dans l'utilisation la plus large aujourd'hui.

– **La norme IEEE 1076.3** (norme numérique)

Cette norme, IEEE 1076.3, qui est souvent appelée la norme numérique ou norme de synthèse, définit les paquetages standards et les interprétations pour des types de données de *VHDL*. Cette norme est prévue pour remplacer les nombreux paquetages (non standards) que les four-

nisseurs d'outils de synthèse ont créés et distribués avec leurs produits. La norme IEEE 1076.3 fait pour les utilisateurs de synthèse ce que IEEE 1164 fait pour les utilisateurs de simulation : augmentation de la puissance de la norme 1076, en assurant sa **compatibilité avec différents outils**. Un paquetage arithmétique se présente toujours avec la structuration suivante :

- la définition de deux types numériques SIGNED et UNSIGNED (norme IEEE 1076, comme la norme IEEE 1164 pour le paquetage std_logic 1164) ;
- la déclaration des opérations arithmétiques (`+, -, *, /, rem, mod`), de comparaisons (`>, >=, <=, <`) et de décalages (`sll, srl`) et de rotations (`ror,rol`) ;
- la déclaration d'un ensemble de fonctions de conversions qui permettent de passer du domaine des vecteurs de bits vers les entiers et vice-versa.

La norme 1076.3 inclut aussi une fonction qui fournit un test de valeurs *don't care* ou « *wild card* », qui sont basées sur le type std_logic. C'est utile, en particulier, pour la synthèse, puisqu'il est souvent utile d'exprimer la logique en termes de valeurs *don't care*.

- **La norme IEEE 1076.4** appelé *(VITAL)*
  L'annotation d'information liée au temps est un aspect important de la simulation numérique. La norme *VHDL* 1076 décrit une variété de dispositifs du langage, qui peuvent être employés pour l'annotation du temps. C'est aussi un des aspects principaux de *VerilogHDL* (le rival le plus direct de *VHDL*.
  Son dispositif, le format standard de retard, SDF[16], permet à des signaux de temps, d'être exprimés sous forme tabulaire et d'être inclus dans les aspects temporels de *Verilog*[17], le modèle actuel de la simulation. La norme de IEEE 1076.4 VITAL[18] éditée par l'IEEE vers la fin de 1995, ajoute ces possibilités à *VHDL* comme paquetage standard et permet, aux fournisseurs et concepteurs d'ASIC[19], de produire des modèles de synchronisation applicables en *VHDL* et en *VerilogHDL*.

[16] The Standard Delay Format.
[17] *VerilogHDL.*
[18] *VITAL signifie VHDL Initiative Toward ASIC Libraries.*
[19] Application Specific Integrated Circuits.

Pour cette raison, les formats de données fondamentaux, de l'IEEE 1076.4 et de SDF de *Verilog*, sont identiques.

Désormais, presque tous les fabricants de composants, au niveau mondial, livrent, non seulement les descriptions schématiques, électriques et physiques de composants, mais aussi le modèle VITAL de ces composants. L'écriture du modèle VITAL est très exigeante dans l'expression du modèle temporel, dans la fonction logique et dans l'interface. VITAL s'appuie sur la stdlogic 1164, le format SDF, des bibliothèques de primitives, et, surtout, sur une structuration stricte du code et une réduction de la syntaxe *VHDL* autorisée.

Il y a une catégorie de modèles qui a pour vocation d'être disponible sans restriction, ce qui ne veut pas dire à bas prix : celle des modèles de circuits les plus répandus. Par exemple, les circuits de la série 74xxx, parce que fabriqués par de nombreux constructeurs, ou, encore, les circuits, comme les 80x86 ou 1es derniers Pentium, qui sont des standards (industriels) de fait, et qui possèdent une large famille.

De telles descriptions sont appelées à être rapidement développées et vendues sous forme de bibliothèques, car les paquetages d'environnement, communs à tous les modèles, représentent une partie importante du coût de développement.

1.1.3 VerilogHDL

Jusqu'à ces dernières années, le seul véritable concurrent de *VHDL* avait pour nom *VerilogHDL*. Souvent on utilise le nom, plus court, ***Verilog***[88, 94, 52]. Parfois le nom complet, de *VerilogHDL*, est justifié pour distinguer la norme *Verilog* et la famille de simulateurs *Verilog-XL*.

VerilogHDL a été complètement défini en 1995, dans la norme IEEE-1364. Ce standard est connu comme le manuel de référence du langage, ou LRM[20]. La norme IEEE-1364 définit également l'interface avec les langages de programmation, ou PLI[21]. Ce dernier est une collection de routines de logiciel, qui permettent une interface bidirectionnelle entre *Verilog* et d'autres langages (habituellement C). Le langage *Verilog* supporte une description textuelle de circuits électroniques, afin de simuler leur comportement ; ainsi il peut être utilisé pour la réalisation de test et les synthèses logiques.

[20]Language Reference Manual.
[21]Programming Language Interface.

Verilog permet plusieurs types de description, de la plus abstraite à la plus « physique » :

- la modélisation comportementale,
- la description au niveau transfert entre registres,
- la description au niveau des portes logiques
- et la description au niveau « *SWITCH* », c'est-à-dire au niveau du transistor[22].

La modélisation comportementale peut être employée pour décrire la fonctionnalité d'un système à un niveau élevé d'abstraction pour analyser le système et le segmenter.

La seconde description, celle au niveau transfert entre registres (RTL), est employée pour la conception détaillée des circuits numériques. Les outils de synthèse peuvent transformer les descriptions de conception de type RTL en une description au niveau de portes logiques. Seuls les deux derniers niveaux de descriptions (port logique, switch) seront utilisés pour la vérification des circuits, y compris la simulation, l'analyse statique et dynamique de synchronisation et la testabilité. Le niveau RTL est utilisé dans un premier temps pour la simulation fonctionnelle, avant de servir pour la synthèse et la génération de portes. Ce dernier niveau d'abstraction, même si il est choisi pour les outils EDA[23], n'est, en fait, pas créé par le concepteur, mais est généré automatiquement lors de la synthèse.

À la différence de *VHDL*, ***VerilogHDL* ne fournit pas une description abstraite efficace au niveau System**, avant le partitionnement entre hardware et software[24]. Si *VHDL* offre, au concepteur, la possibilité de définir ses propres types et de surcharger les opérateurs, ce qui lui permet de formaliser son analyse dans le domaine du problème, *Verilog*, lui, contraint le concepteur à travailler avec les fonctions système prédéfinies, ou des exécutions pour des simulations stochastiques et des analyses de performance. La profondeur d'analyse est, ici, étroitement liée aux concepts et fonctions prédéfinis dans le langage.

Simulation en verilog

Verilog est employé pour décrire des environnements de simulation et examiner les vecteurs, les résultats prévus, la comparaison de résultats et

[22] Voir page 60.
[23] Automatisation de conception électronique, Electronic Design Automation.
[24] VerilogSystem essaie de résoudre ce défaut. Voir page 22.

l'analyse.

Verilog peut être employé pour contrôler la simulation. Par exemple, on peut

- déclarer des jeux de test[25],
- précisant des points de contrôle,
- gérer le temps et les horloges,
- visualiser les histogrammes.

Cependant, la plupart de ces fonctions ne sont pas incluses dans la norme 1364, mais sont des ajouts proposés par les simulateurs commerciaux.

L'exemple suivant représente un multiplexeur :

```
//Code Verilog pour un Multiplexeur

module Multi (In1,In2, Select, O);
  input In1,In2, Select;
  output O;
  assign O = ((In1 and Select) or (In2 and  (not Select)));
endmodule

//fin du code Verilog
```

Il faut faire l'analyse. Pour cela, on va simuler son comportement. On propose des valeurs de vecteurs de test pour simuler le comportement de ce multiplexeur et on récupèra les résultats pour les comparer, ensuite, avec les valeurs attendues.

La simulation s'exécute dans un module plus grand, qui inclut le module à tester. Par exemple, le module suivant analyse l'exemple précédent :

```
module MultiTEST;  // Pas des ports!

//Code Verilog pour un Multiplexeur

module Multi (In1,In2, Select, O);

input In1,In2, Select;

output O; assign O = ((In1 and Select) or (In2 and (not Select)));
```

[25] Test Benches.

```
endmodule

// fin du module \og Multi\fg{}

// Simulation
Multi (In1,In2, Select, O);

begin
    Select = 0; In1 = 0; In2 = 0;
    #5 In1 = 1;
    #5 Select = 1;
    #5 In2 = 1;
end

// Analysis
$monitor($time, , SEL, A, B, F);
endmodule
```

Les instructions d'instanciation initialisent les valeurs des entrées de circuit avec un retard de 5 secondes. En $monitor on précise les signaux que l'on voudrait analyser. Les valeurs de ces signaux seront affichées à chaque fois que l'un d'entre eux change sa valeur.

Un programme (ou des spécifications) *Verilog* est une description d'un dispositif ou d'un processus, **semblable à un logiciel écrit en *C*** ou en *Pascal*. Cependant, *Verilog*, comme *VHDL*, inclut également des constructions spécifiques pour décrire le matériel.

Par exemple, mentionnons les variables de type de transfert entre registres, où les noms eux-mêmes suggèrent un environnement de matériel.

Une **différence** principale, entre un langage comme *C* et *Verilog*, est que *Verilog* permet à des processus de **s'exécuter en parallèle et de façon concurrente,** avec des instructions temporelles très précises. C'est évidemment souhaitable si l'on doit décrire le comportement du matériel d'une manière réaliste.

Le mécanisme principal de **synchronisation**, en *Verilog*, est le **partage de variables**. Un processus peut attendre une variable particulière pour devenir vraie, tandis qu'un autre processus parallèle, peut retarder le changement de

valeur de cette variable jusqu'à ce qu'il soit hors d'une section critique. Dans le cas de modélisation synchrone, les simulateurs *Verilog* doivent tenir compte de la durée précise de chaque processus. Leur capacité à simuler, de manière précise, le comportement réel du circuit, est directement liée à leur puissance de calcul. Il peut observer des différences de temps entre la simulation et le circuit modélisé en *Verilog* réel.

1.1.4 Méthodologie générale de conception en *HDL*

Nous venons de présenter les deux langages de description matérielle les plus utilisés. D'un point plus méthodologique, voici les étapes communément citées pour la conception dans le domaine matérielle :

1. **Etablissement de cahier des charges**[26]
   L'établissement du cahier des charges complet et précis d'un système complexe est une tâche très ardue. Les fonctionnalités et les contraintes associées sont décrites en langage naturel dans un document qui peut être extrêmement volumineux. Cette technique est sujette à imprécision, ambiguïté, oubli, voire contradiction. Aucune prédiction de performances ou de coût n'est possible sans l'expérience des chefs de projets et des architectes systèmes.

   Le cahier des charges, donnant accès à une simulation fonctionnelle partielle ou complète, doit permettre d'**exprimer le QUOI** (la fonction) **indépendamment du COMMENT** (l'implémentation physique). La conception sera l'exercice du passage du premier au deuxième niveau. Avec ces outils, toutes les conceptions pourront se référer, pour comparaison, aux résultats de simulation fonctionnelle du cahier des charges.

2. **Conception descendante, ascendante ou mixte**
   La conception descendante ou « *top-down* » est le fait d'effectuer une conception en partant d'un haut niveau d'abstraction (une description fonctionnelle) pour aboutir, par étapes successives, à une réalisation physique par interconnexion de composants de base (circuits discrets ou circuits intégrés complexes, ASIC[27], processeurs, logiciel, etc.). Cette approche est complémentaire de l'étape précédente.
   La conception descendante permet de retarder le choix de la technologie le plus tard possible dans le cycle de conception. De la même façon,

[26] Voir page 50.
[27] Application Specific Integrated Circuits.

une modification de technologie ne remet pas en cause les premières étapes de la conception.

Par opposition, dans la conception ascendante ou « *Bottom-Up* », on va commencer par concevoir technologiquement toutes les briques de base pour ensuite les assembler (avec les risques d'erreurs dans les définitions de connexions).
Dans la réalité, on utilise conjointement les deux techniques, on parlera de *conception mixte* : par exemple la conception d'ASIC, à partir d'une technique de cellules pré-caractérisées, peut être menée avec une conception « Top-Down », en utilisant des cellules issues d'une approche « Bottom-Up » (mais définie une fois pour toutes).

3. **Multi abstraction**
La multi-abstraction est le fait de pouvoir mélanger, dans une même description, des objets décrits de façon très différente. On peut alors simuler un système avec des parties décrites au niveau fonctionnel (par exemple une équation mathématique), alors que d'autres parties sont décrites par l'instanciation structurelle de transistors, de la technologie choisie, tenant compte des phénomènes physiques au niveau du semi-conducteur.

La puissance de vérification des conceptions en cours, augmentée : un concepteur prend en charge la conception d'un sous-système Xk au sein d'un système X. Ce dernière est décrit sous la forme d'un assemblage structurel (interconnexion de modèles) de chacun de ses sous-ensembles Xl, X2, ..., Xk et cette description est simulable. Les vérifications de toutes les étapes de conception du sous-ensemble Xk en cours d'étude pourront se faire en tenant compte des interconnexions avec les autres sous-systèmes.
Quand un système est constitué de plusieurs parties, et que la simulation pose des problèmes de temps de calcul, on peut profiter de cette méthode pour améliorer la situation. Plus un objet est décrit à un haut niveau d'abstraction, plus sa simulation est performante en temps. En associant les modèles à différents niveaux d'abstraction, on peut valider complètement un système sans avoir à le simuler totalement au plus bas niveau.

4. **Simulation environnementale**
Un système (S), ou un circuit intégré, n'est jamais conçu pour fonction-

ner seul. Il est toujours au sein d'un système plus global, éventuellement interconnecté avec d'autres entités de celui-ci. Dans les méthodes de conception moderne, on fera l'étude du système en tenant compte des **interactions avec l'environnement**, en fabriquant un modèle global.

Il sera possible, par exemple, de prévoir l'augmentation de température liée au fonctionnement du système global (en tenant compte de la dissipation propre de (S)) et d'en tenir compte dans les performances du système en cours d'étude.

Les signaux d'entrée et de sortie ont besoin, pour eux-mêmes, d'un langage de description. *VHDL* permet de décrire, de façon sommaire, une forme d'onde, mais un langage spécifique, dérivé de *VHDL*, est utilisé.

1.1.5 La description de HDL au niveau Système

La complexité de la conception augmente avec le nombre de portes intégrées dans les circuits électroniques [110, 80]. Il est impossible de réaliser la conception correcte et détaillée de plusieurs millions de portes, sans avoir un plan global pour cette conception.

Le temps de simulation complète du système devient aussi très grand et non applicable. Ceci nécessite la conception au niveau système. A titre d'exemple, une unité arithmétique et logique 32 bits, partie centrale d'un processeur possède 77 entrées ce qui porte à 2^{77} le nombre de cas à simuler, dans le cas simple (logique à 2 valeurs), soit un temps de calcul de l'ordre du milliard d'années.

La conception mixte [7, 122] de matériel-logiciel est une autre justification de ce niveau : les conceptions actuelles imposent aussi bien des composants matériels que logiciels. En pratique le temps de mise au point est dans la plupart du temps coté logiciel plutôt que matériel. On a besoin de plus en plus d'un langage qui décrit le système le plus abstrait déclinable lors de l'implémentation pour partie en logiciel et l'autre en matériel [117].

Les *HDLs* décrivent seulement le matériel. De plus, les éléments importants de la conception peuvent être amenés à *migrer* du logiciel au matériel, et inversement, dans les générations successives, en particulier parce que des normes se sont établies ou sont changées. Et pour cela, il est essentiel que le système soit défini d'abord, et l'implémentation exacte (matériel ou logiciel) est précisée plus tard, pendant le processus de conception.

La première solution pour avoir la description au niveau Système était d'**étendre les *HDLs*** existants. Même si *VHDL* peut décrire la hiérarchie des composants du circuit, il n'est pas capable de déclarer le système complet, en particulier pour ce qui touche la couche *logiciel de base* liée aux systèmes d'exploitation. Une tentative de développement de *VHDL* dans cette direction a conduit à VSPEC, une interface basée sur *VHDL*, qui peut représenter le matériel et le logiciel, ainsi que les propriétés et contraintes formelles, en ajoutant une annotation aux entités de *VHDL*.

Selon VSPE[28], la dernière version de l'analyseur de VSPEC, qui est encore en cours de développement, a été faite en juin 1999. Pendant les cinq dernières années, plusieurs propositions pour étendre *VHDL*, et aussi, *Verilog* ont été étudiées. Beaucoup de ces propositions ont été publiées, mais peu ont été suivies.

De son côté OVI, qui permet l'utilisation libre et internationale de *Verilog*, a dévoilé en novembre 1999, ses plans pour la conception au niveau système, et le manuel de référence (SRM) correspondant a été diffusé en juin 2000.

Certains ont essayé de **développer des langages totalement nouveaux**. Partant du principe que, dans le domaine du co-design, le temps de conception consacrée au logiciel est 2 à 10 fois plus important que celui dédié au matériel, certains ont imaginé une **extension du langage C** aux spécificités de la conception matérielle.

Le plus grand problème lié à l'utilisation de C, ou de *C++*, pour décrire le matériel est que, ni l'un ni l'autre, n'ont de solution élégante de représentation de types de données contraints, des horloges, de la simultanéité ou la synchronisation. Se pose aussi et surtout la question de la synthèse de circuits, c'est-à-dire la traduction de la spécification matérielle en modèle RTL, porte, (voire transistor).

En septembre 1999, plus de 55 compagnies de spécialistes (en systèmes, semi-conducteurs, IP « propriétés intellectuelles », logiciels et EDA) se sont réunis pour approuver l'initiative ***Open SystemC*** [115, 15, 42]. Celle-ci propose de faciliter, promouvoir et accélérer l'échange et la conception d'un modèle, en protégeant la propriété intellectuelle, au niveau système, en utilisant une plateforme *C++* commune. Les concepteurs, grâce à un modèle

[28] Voir http ://www.ececs.uc.edu/k̃bse/projects/vspec/#VSPEC

OLC [29], peuvent créer, valider et partager des modèles, avec d'autres compagnies, en utilisant le *SystemC* et un compilateur *ANSI C++* standard. [30]

Le but du *SystemC* est de définir une plateforme de développement qui utilise des classes C de bibliothèques et un noyau de simulation, qui fournit plus d'interopérabilité, portabilité et lisibilité. Avant que le *SystemC* soit proposé, il n'y avait pas de style *C++* commun accepté en industrie, ce qui forçait les compagnies de maintenir de multiples modèles *C++* afin d'échanger et de réutiliser des modèles au niveau système.

L'intérêt de l'utilisation d'une classe de bibliothèques *C++* réside dans le fait que cela permet aux concepteurs d'exprimer des concepts matériels comme la concurrence, le parallélisme, les ports, les connexions et la réactivité. Se mettre d'accord sur une bibliothèque standard permettra aux fournisseurs de propriétés intellectuelles et aux concepteurs de systèmes d'utiliser un même dialecte *C++*, ce qui entraîne l'interopérabilité et fournira une plateforme commune, pour la construction des synthèses, des conceptions matérielles/logicielles et des outils de vérification [79].

Un domaine dans lequel le *SystemC* se distingue, par rapport à d'autres variantes *C/C++*, et notamment par rapport à des remplaçants de *Verilog*, est le domaine des abstractions des communications. Ces constructions forment une partie importante de la contribution technique à l'initiative Open *SystemC*. Les abstractions des communications permettent aux concepteurs de modéliser la communication entre les différents composants matériels du système, sans avoir à concevoir des détails de bas niveau, comme les protocoles de bus.

Le *SystemC* est le résultat d'une collaboration technique entre Synopsys, CoWare et Frontier Design. Synopsys et CoWare ont développé des solutions conceptuelles similaires en C dans les dernières années. Frontier Design et Synopsys ont collaboré à la définition des types de données nécessaires aux applications dans les communications digitales, audio digitales et audio vidéo. De plus, la base du *systèmeC* est le fruit de la recherche innovatrice, en conception C, développée à IMEC, MIT, Stanford et Irvine.

[29] Open Community Licensing.
[30] Voir le site (www.systemc.org).

1.1.6 La vérification formelle de conception HDL

Le terme « **vérification** » regroupe communément l'ensemble de toutes les techniques de **recherches de bugs** et tout particulièrement la **simulation** [102, 99, 62, 46, 45]. La **vérification formelle** désigne, elle, les méthodes garantissant le « **zéro défaut** », méthodes le plus souvent basées sur la logique et/ou les automates [33, 121]. Pendant longtemps, les méthodes formelles ne semblaient pas être utilisables en pratique pour la conception de systèmes matériels ou logiciels [54, 64, 26, 1, 39].

En effet,

1. leurs syntaxes étaient jugées trop rébarbatives,
2. leur puissance trop faible par rapport à la dimension des problèmes à traiter,
3. les outils inadaptés ou trop difficiles à utiliser,
4. les études insuffisantes en nombre, des cas non triviaux,
5. et enfin trop peu de gens disposaient de la formation nécessaire.

Durant la dernière décennie, ces methodes se sont développées :
- dans le domaine du logiciel (ex : langages Z puis B),
- dans le domaine du matériel, par l'adoption de techniques du model-checking [83, 38] et des démonstrateurs de théorème (en complément de la simulation),
- le nombre d'études de cas réels, ainsi que la confiance dans l'utilisation des méthodes formelles.

De nombreuses équipes de vérification sont apparues chez des concepteurs logiciels (Matra) et la plupart des concepteurs matériels (IBM, Intel, SUN, HP, Cadence, Siemens, etc.). De nombreuses sociétés (Chrysalis, Cadence, Synopsis, IBM, etc.) commercialisent des outils, de conception matérielle ou logicielle, basés sur des spécifications formelles.

En effet, la **complexité des conceptions** a augmenté à un point tel que la validation traditionnelle est devenue la **difficulté principale au cours du développement de circuit.** La méthode traditionnelle de vérification passe par un processus de simulation qui peut faire intervenir potentiellement un nombre astronomique de données en entrée. C'est pourquoi, seule une petite partie de l'ensemble des fonctionnalités du circuit peut être réellement vérifiée. La vérification formelle est statique ; elle est capable de travailler **sans devoir énumérer ou simuler tous les états possibles du système**.

Elle peut être comparée à l'analyse statique de synchronisation. **Statique signifiant que cette synchronisation peut être analysée sans appliquer aucun signal à l'entrée.**

En employant ce type de méthode, il devient possible, dans certaines circonstances, de montrer qu'une conception est correcte, sans appliquer des ensembles importants de *jeux de test*, puisque la preuve formelle est établie quelques soient les données en entrée. Ceci est particulièrement important dans le cas de systèmes critiques. La correction n'est plus statistiquement établi avec un taux de 99% mais bien formellement pour toute donnée en entrée.

En cas de **détection d'erreurs**, un diagnostic est souvent plus facile à établir dans le cas formel si on dispose d'outils d'analyse automatique adaptés.

Enfin **les modifications**, qui peuvent intervenir tardivement dans la phase de conception, sont aussi une raison pour laquelle la vérification est devenue un aspect important, pour des technologies submicroniques. A cette échelle d'intégration, certains phénomènes physiques perturbent le modèle, purement logique, utilisé dans tous les outils de conception matériel et obligent à introduire de la redondance pour une meilleure tolérance aux pannes. Ces modifications sont souvent nécessaires avant la production. Celles-ci peuvent inclure aussi des ajustements de synchronisation et des insertions de tableaux de tests, etc. La plupart de ces changements sont faits plus ou moins manuellement, directement dans la *netlist*, au niveau des portes, et peut, potentiellement, changer le comportement du circuit (chip).

Types de vérification formelle

La première, appelée **vérification d'équivalence**, compare une version de la conception avec une autre. Typiquement si on se place dans le cas de modifications tardives, la *netlist* au niveau porte est comparé avec le modèle RTL[31] correspondant. Dans ce cas-ci, le modèle RTL est employé comme « version d'or[32] » qui doit être vérifiée séparément. La vérification est dite formelle, ici, car l'équivalence est démontrée quelques soient les données en entrée. La question à traiter correspond au problème SAT[33].

[31] Voir page 61.
[32] Golden Version.
[33] SATisfiabilité.

Un deuxième type de vérification formelle concerne la **vérification des protocoles**. Il est particulièrement adaptée pour des circuits multiprocesseurs fortement communiquant. Des outils, comme Murphi et SPIN, utilisés plutôt dans le cadre de la vérification des logiciels (architecture logicielle distribuée), ont été expérimentés dans le cadre de la vérification matérielle avec un succès mitigé. A l'opposé du première type de vérification, un des problèmes est que cette technique s'applique uniquement aux niveaux abstraits de la conception, sans relation avec les niveaux plus concrets.

Le troisième type de vérification formelle, plus largement répandu, est le **model-checking symbolique ou property-checking** [27].
Une grande partie du comportement ordinaire du circuit est encore vérifiée par la simulation, mais la plupart pourraient être vérifiées par vérification formelle. En utilisant un langage séparé, qui peut être une extension de celui du RTL, un ensemble de propriétés du circuit peut être défini. Celles-ci sont souvent exprimées en domaine de temps ; elles indiquent ce qui se, produit pendant un certain nombre d'unités de temps, impliquant des variables d'état et la logique combinatoire. En employant la property-checking, une propriété peut être **prouvée toujours vraie**.

La théorie et les algorithmes pour la vérification formelle ont fait l'objet des nombreux travaux de recherches universitaires, et les outils commerciaux sont disponibles comme SMV, FormalCheck, RuleBase. Depuis peu, PSL/Sugar se propose d'être un standard pour la spécification formelle de circuits électroniques. C'est une extension des logiques temporelles CTL et LTL, au sens que toute spécification PSL[34] peut-être compilée en une pure formule CTL ou LTL. Les limitations intrinsèques à ces méthodes sont les mêmes que celles liées à la simulation, à savoir, le nombre prohibitif d'états à tester. Ceci conduit à ne pouvoir vérifier formellement que de petits composants matériels. Cela résulte ici de la prise en compte du *temps* dans la modélisation formelle.
Le dernier type de vérification formelle, que nous allons plus particulièrement développer ici, concerne la **démonstration de théorèmes**[35]. Son utilisation est encore limité malgré quelques beaux succès industriels (vérification du calcul en virgule flottante du AMD K7 par Russinoff). Ses problèmes majeurs sont les suivants :

1. ces assistants prouveurs utilisés nécessitent une interaction humaine importante,

[34]Property Specification Language.
[35]Theorem proving.

2. il manque encore de bonnes stratégies, méthodes et procédures de décision adaptées aux problèmes à traiter.

1.2 Langages d'architecture de logiciel

Une architecture de logiciel [76, 81] décrit la **structure et le comportement d'un système logiciel** et les **interfaces** du système avec son environnement. Dans une architecture de logiciel, un système est représenté comme ensemble de composants de logiciel, de leurs connections, et de leurs principales interactions comportementales.

La création d'une architecture de logiciel permet de mieux comprendre le système, et par la suite de faciliter le processus de conception. Elle fournit également une base pour l'analyse rigoureuse de la conception de système, ce qui rend la détection possible des erreurs plus tôt. Cette détection mène, à son tour, aux améliorations de la qualité du logiciel, diminue les coûts de développements et aident à assurer l'exactitude.

Un langage de description d'architecture(***ADL***) est un langage employé pour décrire une architecture de logiciel. Il peut être un langage descriptif formel ou semi-formel, un language graphique, ou tous les deux. Un *ADL* peut être associé à un ensemble d'outils, pour rendre l'analyse d'architectures plus facile.

Ces dernières années, il y a eu une quantité considérable de recherche autour des *ADLs* [84, 81, 76]. Beaucoup de langages de description d'architectures ont été proposés pour satisfaire aux besoins de différents domaines d'application, par exemple : Aesop, ArTek, C2, Darwin, LILEANNA, MetaH, Rapide, SADL, UniCon, Weaves, Wright,[24, 13], et différents aperçus sur ce sujet ont été publiés. Chaque *ADL* fournit au concepteur, des possibilités complémentaires pour le développement et l'analyse architectural. Ces *ADLs* et leurs outils de support, sont développés indépendamment, ce qui rend difficile d'intégrer ces outils et de partager des descriptions architecturales.

Bien qu'il y ait peu de consensus au sujet de ce qu'est, exactement, un *ADL*, il y a un accord sur ce qu'est l'ensemble des concepts essentiels, au moins, en ce qui concentre sur l'aspect structural de la description architecturale. ***ACME*** [58, 57, 72, 56, 61] a été développé, dans un effort commun

de la communauté de recherches d'architecture de logiciel, **pour standardiser un format commun d'échanges entre des outils différents de conception d'architecture**.

ACME incorpore les concepts essentiels d'un *ADL*. Et pour cela, il peut être considéré comme un *ADL* de seconde génération [53]. Il focalise sur la définition d'un ensemble des concepts principaux qui sont reliés aux aspects structuraux de l'architecture de logiciel. L'objectif, comme on l'a dit, est d'avoir la possibilité d'échanger des descriptions entre plusieurs *ADLs*, qui sont développés indépendamment. *ACME* est un *ADL* générique, qui peut être employé pour décrire des concepts plus spécifiques, en spécialisant les concepts de base de ADLs.

Comme les autres *ADLs*, *ACME* se concentre sur la conception et l'évaluation d'architecture de logiciel. Un problème important, des systèmes basés sur les composants, est de trouver les notations appropriées pour décrire ces systèmes. Des bonnes notations permettent de documenter ces systèmes clairement, de bien préciser leurs propriétés, et d'automatiser leur analyse. La description en *ACME* est faite d'une notation textuelle et graphique. La spécification structurelle de conception est déclaré par les notions de composants, connecteurs, bindings, attachement, .etc., alors que la spécification fonctionnelle est déclarée par les propriétés, les contraintes, les types, etc. [55]. La présentation de *ACME*, dans les paragraphes suivants sert à comprendre les aspects génériques de *ADLs*.

Les composants et les ports

Les Composants, en *ACME*, représentent les éléments de base permettant de stocker des descriptions de calcul et des données d'un système. Il n'y a aucune restriction à la taille d'un composant [53]. Un composant peut être aussi petit qu'un bouton dans une interface graphique utilisateur ou aussi grand qu'un serveur web. Pour les composants complexes, il est essentiel d'avoir la possibilité de cacher leurs structures internes. L'interface d'un composant, c'est-à-dire sa vue externe, est décrite comme un ensemble de ports. **Un port** est un point d'interaction entre ce composant et son environnement.

Les connecteurs et les rôles

Un connecteur représente une interaction entre les composants. Comme les composants, les connecteurs peuvent être de complexité variée. Cette

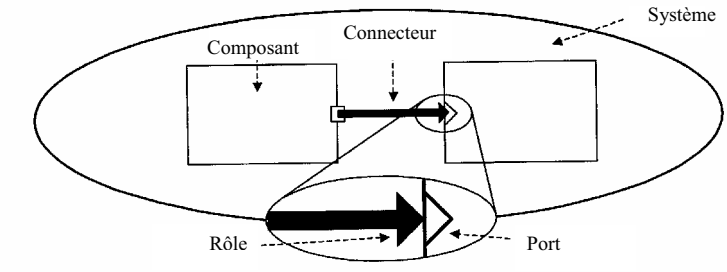

FIG. 1.3 – Composants et ports.

notion permet de représenter, de la même manière, des concepts simples, tels qu'un appel de méthode, et des concepts très sophistiqués et abstraits tels qu'une connexion d'un client SQL avec un serveur de base de données.

En effet, un des aspects importants de beaucoup d'*ADLs* est de déclarer les connecteurs dans l'entité de conception de plus haut niveau. **On peut spécifier une interaction souhaitée entre des composants avant même de déclarer ces composants.** En plus, *ACME* permet de définir un type de connecteur.

Puisqu'un connecteur peut être assez complexe, il est important de pouvoir décrire son interface indépendamment de sa structure interne. L'interface d'un connecteur est définie comme ensemble de rôles. **Un rôle**, pour un connecteur, est comme un port pour son composant, c'est-à-dire, ses **points d'interaction**. Beaucoup de connecteurs simples sont binaires. Par exemple, un appel de méthode est un connecteur avec deux rôles, l'appelant et l'appelé, un pipe a un producteur et un consommateur, un canal d'événement a un émetteur et un récepteur ; et ainsi de suite. Des flèches simples pourraient être une notation graphique appropriée pour les connecteurs binaires.

Les systèmes et les attachements

ACME permet de décrire des composants et des connecteurs séparément, mais le but est de décrire un système complet comme assemblage de composants et de connecteurs. La structure d'un système est indiquée par un ensemble de composants, un ensemble de connecteurs, et un ensemble d'attachements. **Un attachement** lie un port d'un composant à un rôle d'un connecteur.

Les connections entre les composants sont ainsi modélisées comme une séquence de composant-port-rôle-connecteur-rôle-port-composant. Ceci peut sembler inutilement complexe, mais c'est tout à fait nécessaire pour modéliser des systèmes aussi riches que variés. En particulier, ceci nous permet de penser les connecteurs, indépendamment, ce n'est pas possible que si des interfaces de connecteurs sont explicitement définies.

Les représentations et les Bindings

Les concepts qui sont présentés ci-dessus permettent de décrire comment un système est constitué de composants et de connecteurs, en donnant une vue globale sur ce système. La conception de haut en bas (top-down) des architectures des logiciels rend nécessaire la possibilité d'écrire son architecture au niveau le plus élevé, et ensuite de la raffiner au niveau de plus en plus détaillée. A un niveau donné, une interface de composant peut être associée à un ou plusieurs composants. La représentation du composant, dans ce cas, se fait par un *système* et un ensemble de *Binding*. **Chaque *Binding* lie un port interne à un port externe.** Ainsi un composant peut être décomposé récursivement. Les connecteurs peuvent également être décomposés d'une manière semblable. Un aspect important de cette approche est que le même formalisme est employé à chaque niveau de décomposition.

Les propriétés, les contraintes, les types, et les Styles

Les concepts présentés ci-dessus peuvent être employés pour décrire l'aspect structurel de l'architecture du logiciel. Pour améliorer la description des composants, des connecteurs, et des systèmes, chaque *ADL* offre l'information supplémentaire pour définir, par exemple, le comportement du composant, les protocoles de l'interaction, et les propriétés fonctionnelles et extrafonctionelles.

Au lieu de définir un ensemble fixe de caractéristiques, *ACME* permet d'**annoter chaque entité avec un ensemble arbitraire de propriétés**. Chaque propriété a un type et une valeur.

ACME inclut également un langage de contraintes, celles-ci sont basées sur la logique du premier ordre. Des contraintes peuvent être attachées à n'importe quelle entité architecturale. Ceci inclut des contraintes simples, comme celles définissant l'intervalle des valeurs, qu'un attribut peut prendre, ou le nombre de connections, admises par un composant, mais également des contraintes sophistiquées pour contrôler la topologie du système.

ACME fournit également un service pour définir de nouveaux types, y compris des types de propriétés, de composants, de connecteurs, ou de ports. Les types en *ACME* ne sont pas exprimés, seulement, en termes de structure des éléments qu'ils caractérisent, ils incluent, également, les contraintes qui doivent être satisfaites par les éléments. Un dispositif intéressant d'*ACME* est qu'il nous permet de définir le modèle de l'architecture, c'est-à-dire, le type de système.

ACME définit un ensemble relativement petit de concepts clairement identifiés. Ces concepts sont généralement trouvés chez tout autre *ADLs*. C'est pourquoi *ACME* peut être considéré comme un *ADL* générique. Il ne contient pas tous les dispositifs trouvés dans tout l'ensemble des *ADLs*, mais sa généralité, la mise en évidence des aspects structuraux, le rend intéressant, en particulier, quand il s'agit de prendre en compte les descriptions des modèles composants industriels.

En effet, les modèles industriels de composants se concentrent sur un sous-ensemble des concepts, décrits ci-dessus, et ignorent, en grande partie, les dispositifs avancés tels que les spécifications de comportement.

Enfin *ACME*, comme d'autre *ADLs*, facilite la notation et la conception de nouveaux systèmes. Cependant il ne fournit pas l'aide substantielle pour un logiciel d'implémentation ou de traitement de code. C'est pourquoi il est peu probable que *ACME*, comme autre *ADL* existant, puisse être largement utilisé en industrie.

1.2.1 Analyse de Conception en ADL

Les *ADLs* proposent deux critères pour l'analyse [2], qui est utilisée pour la description architecturale. Ces deux critères sont cohérence et complétude[36] Intuitivement, **la cohérence** signifie que la description, de la spécification initiale et pendant le développement de conception, a toujours un sens, et ces différentes parties de la description ne se contredisent pas.
La complétude assure, elle, qu'il ne manque aucune information nécessaire au bon fonctionnement du programme.

[36]Consistency and Completeness. Voir page 51.

Outils des *ADLs*

La conception de multi-niveaux exige plus qu'un outil de représentation. L'utilisateur doit pouvoir construire des architectures, les voir, et les analyser en employant souvent d'autres logiciels. L'outil doit donner au concepteur la possibilité de l'édition graphique, la disposition des vues produites, et l'intégration en d'autres outils d'analyse. A la fin, la vérification de cohérence est également importante. Les *ADLs* existants fournissent des outils de conception et de développement très variées [81].

D'une part, l'outil de SADL consiste principalement en un « consistency checker » de raffinement. Weaves s'est concentré sur des spécifications et la manipulation interactive des architectures. D'autre part, Darwin, Rapide, et UniCon supportent des environnements puissants pour modéliser l'architecture. Actuellement, *C2* et *Darwin* sont les *ADLs* fournissant le plus large spectre, en termes d'outils de conception, et de développement.

Plusieurs outils sont fournis actuellement pour faciliter la tâche du concepteur en *ACME* :

- ***AcmeLib***[37]
  fournit une infrastructure générique et extensible pour décrire, représenter, produire, et analyser des descriptions d'architecture de logiciel [56].
  La bibliothèque du réalisateur d'outil (*AcmeLib*) est un cadre, orienté objet, pour les outils architecturaux, écrite en Java. Cette bibliothèque lit, écrit et manoeuvre des conceptions d'architecture de logiciel d'*ACME*.
  Le cadre d'*AcmeLib* est conçu pour aider au développement rapide de deux classes d'applications :

  1. la traduction de description d'architectures écrites en un code « natif » ADL (tels que Rapide, Wright, UniCon, et Aesop) vers un autre code ADL.

  2. Soit la conception d'architectures ACME proprement dit et des outils d'analyse associés [61].

[37] Acme Tool Developer's Library.

– ***AcmeStudio***
est un outil graphique *ADL* qui est utilisé [17] pour gérer des conceptions architecturales de logiciel basées sur le langage de description architectural d'*ACME*. Avec *AcmeStudio*, on peut définir de nouvelles familles d'*ACME* et adapter l'environnement aux besoins du client. Ces familles peuvent être déclarées comme des modèles de diagramme. *AcmeStudio* est adapté à *Eclipse*[38] et peut être employé dans une variété d'applications de modélisation et d'analyse. *AcmeStudio* est mis en application en tant que plugin pour l'environnement d'*Eclipse*. Eclipse fournit à un plugin environnement permettant des prolongements faciles d'*AcmeStudio* de nouvelles analyses et fonctionnalité, et à la personnalisation de nouveaux environnements architecturaux conçus en fonction une organisation particulière.

Par exemple : Steppe and all. explique dans [111] son expérience de prolonger *AcmeStudio*, pour faire en sorte qu'il supporte les deux modèles architecturaux, ceux qui ont été développés pour « Ford motor company automative control systems ». Il offre en plus, d'abord la capacité de lire les modèles de composants de, Simulink, de Ford, et ensuite, la génération des assemblages des architectures de système, de niveau élevé. Cet outil s'appelle Synergy.

Une future version d'*AcmeStudio* inclura le soutien d'***ARMANI*** [119] pour le type-checking [72]. Bien qu'*ARMANI* ne soit pas officiellement soutenu dans la version publique actuelle d'*AcmeStudio*, on peut coder des constructions d'*ARMANI* en *ACME* et ensuite traduire son descriptions de *ACME* en *ARMANI*, à l'aide des outils fournis dans les librairies d'*ARMANI*. *AcmeStudio* inclut des types de propriété de bui1t-in pour ces trois prolongements d'*ARMANI* : *ARMANI-HeuristicT*, *ARMANI-InvariantT* et *ARMANI-AnalysisT*.

Exemple : Nand3 avec trois entrées en ACME, (en se basant sur des composants à deux entrées).

```
// Définir des composants génériques
// - NAND à 2 entrées.
// - externalconnect pour les connexions externes.
// -line_1_1_s : connexion entrée-sortie.
// -line_1_2_s : connexion à une entrée et deux sorties.
```

[38] un environnement ouvert de développement intégré par Java de source.

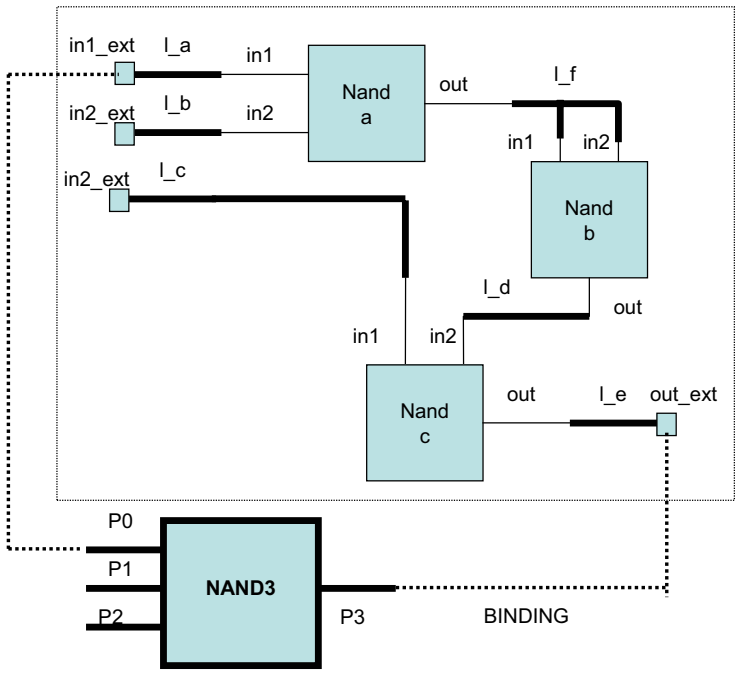

FIG. 1.4 – Nand3 en ACME.

```
Family Circuits = {
    Component Type externalconnect = {
        //sortie de porte
        Port connect = {
            Property val_connect : boolean;
        };


    }


    Component Type NAND = {
        //1er entrée de porte Nand
        Port in1 = {
            Property val_in1 : boolean;
        };
```

```
    //2eme entrée de porte Nand
    Port in2 = {
        Property val_in2 : boolean;
    };


    //sortie de porte Nand
    Port out = {
        Property val_out : boolean;
    };

  // La function logique de Nand
    invariant  (    self.in1.val_in1
                 AND self.in2.val_in2
               )
                  = ! self.out.val_out ;

}


//connextion à une entrée et une sortie
Connector Type line_1_1_s = {
    //sortie de connexion
    Role out = {
        Property val_out : boolean;
    };


    //entrée de connexion
    Role in1 = {
        Property val_in1 : boolean;
    };


    invariant (self.out.val_out) == (self.in1.val_in1);

}


//connexion à une entrée et deux sorties
Connector Type line_1_2_s extends line_1_1_s with {
```

```
        //2eme sortie de connexion
        Role out2 = {
            Property val_out2 : boolean;
        };


        invariant (self.out.val_out) == (self.out2.val_out2);

    }


}


//La création de Nand avec 3 entrées avec 2 niveaux de conceptions

System Nand3 : Circuits = {
    // le niveau abstrait nand3 est une boîte avec 3 entrées
    Component Circuit_3_Nands = {
        Port p0 = {
            Property vis-order : int = 3;
                     // numero de l'instanciation
            Property aProp : int;
        };

        Port p1 = {
            Property vis-order : int = 2;
        };

        Port p2 = {
            Property vis-order : int = 1;
        };

        Port p3 = {
            Property vis-order : int = 0;
        };

// Un autre niveau plus concrète de conception
   Representation rep = {
      System rep = {
         Component in2_ext : externalconnect =
```

```
                        new externalconnect extended with {

  Port connect = {
     Property val_connect : boolean;
     Property vis-order : int = 0;
           };

       };

Component in1_ext : externalconnect =
       new externalconnect extended with {
  Port connect = {
    Property val_connect : boolean;
    Property vis-order : int = 0;
           };

       };

Component nand_a : NAND =
              new NAND extended with {
  Port in1 = {
    Property vis-order : int = 0;
    Property val_in1 : boolean;
           };

  Port out = {
    Property vis-order : int = 1;
    Property val_out : boolean;
           };

  Port in2 = {
    Property vis-order : int = 2;
    Property val_in2 : boolean;
           };

       };

Connector l_d : line_1_1_s =
               new line_1_1_s extended with {
   Role out = {
   Property val_out : boolean;
```

```
            };

    Role in1 = {
    Property val_in1 : boolean;
            };

        };

Connector l_b : line_1_1_s =
             new line_1_1_s extended with {
  Role out = {
    Property val_out : boolean;
            };

  Role in1 = {
    Property val_in1 : boolean;
            };

        };

 Connector l_e : line_1_1_s =
              new line_1_1_s extended with {
    Role out = {
      Property val_out : boolean;
            };

    Role in1 = {
      Property val_in1 : boolean;
            };

        };

 Connector l_f : line_1_2_s =
             new line_1_2_s extended with {
    Role out2 = {
      Property val_out2 : boolean;
            };

    Role out = {
      Property val_out : boolean;
            };
```

```
   Role in1 = {
     Property val_in1 : boolean;
           };

       };

Component out_ext : externalconnect =
                new externalconnect extended with {
  Port connect = {
    Property val_connect : boolean;
    Property vis-order : int = 0;
           };

       };

Component nand_b : NAND =
               new NAND extended with {
  Port in1 = {
    Property val_in1 : boolean;
    Property vis-order : int = 0;
           };

  Port out = {
    Property val_out : boolean;
    Property vis-order : int = 1;
           };

  Port in2 = {
    Property vis-order : int = 2;
    Property val_in2 : boolean;
           };

       };

Connector l_a : line_1_1_s =
                  new line_1_1_s extended with {
  Role out = {
    Property val_out : boolean;
           };
```

```
    Role in1 = {
      Property val_in1 : boolean;
             };

        };

Component nand_c : NAND =  new NAND extended with {
  Port in1 = {
    Property val_in1 : boolean;
    Property vis-order : int = 0;
            };

  Port out = {
    Property val_out : boolean;
    Property vis-order : int = 1;
            };

  Port in2 = {
    Property vis-order : int = 2;
    Property val_in2 : boolean;
           };

   };

   Component in3_ext : externalconnect =
              new externalconnect extended with {
  Port connect = {
    Property val_connect : boolean;
    Property vis-order : int = 0;
           };

   };

   Connector l_c : line_1_1_s =
               new line_1_1_s extended with {
       Role out = {
           Property val_out : boolean;
       };

       Role in1 = {
           Property val_in1 : boolean;
```

```
                };

            };
        // les connexions entre les composants de 2eme niveau
            Attachment in1_ext.connect to l_a.in1;
            Attachment in3_ext.connect to l_c.in1;
            Attachment nand_a.in2 to l_b.out;
            Attachment in2_ext.connect to l_b.in1;
            Attachment nand_c.in2 to l_d.out;
            Attachment nand_c.in1 to l_c.out;
            Attachment nand_b.in1 to l_f.out;
            Attachment nand_b.in2 to l_f.out2;
            Attachment nand_b.out to l_d.in1;
            Attachment nand_a.in1 to l_a.out;
            Attachment nand_a.out to l_f.in1;
            Attachment nand_c.out to l_e.in1;
            Attachment out_ext.connect to l_e.out;
        };
        // les correspondances entre les deux niveaux
        // de conception.
        Bindings {
            in1_ext.connect to Circuit_3_Nands.p0;
            in2_ext.connect to Circuit_3_Nands.p1;
            in3_ext.connect to Circuit_3_Nands.p2;
            out_ext.connect to Circuit_3_Nands.p3;
        }
    };
  };

};
```

Comme on peut le remarquer, les concepts de base en *ACME* peuvent être employées pour décrire l'architecture aussi bien logicielle que matérielle surtout dans les premières étapes de conception.
Même si ces concepts (composant, connecteur, etc.) soient clairement définis en *ACME*, la manière de valider une conception n'est pas encore très claire. L'utilisation d'*ARMANI* pour décrire les propriétés dans les versions futures de *ACME* doit permettre d'étayer plus formellement ces concepts.
Une solution pour verifier , voire prouver, une conception *ACME* peut être aussi la prise en compte du principe de raffinement en couplant ACME à une méthode formelle telle que la méthode *B*.

1.3 La modélisation en utilisant la méthode B

La méthode ***B*** [3, 48] est la parfaite illustration de l'idée selon laquelle vérification formelle et synthèse automatique de l'implémentation, sont les plus sûrs moyens de garantir la sécurité.

La méthode *B* est utilisée, dans l'industrie, pour développer et prouver la conception de logiciels. Récemment des recherches ont été menées pour l'appliquer à la conception de circuit numériques.

La méthode *B* est basée sur un principe de **raffinements successifs** [107, 112] qui, à partir d'une spécification très abstraite, produisent un exécutable « synthétisé et concret ».

La première étape consiste en une spécification formelle du programme, sous forme d'une machine abstraite. Ensuite, après chaque étape de raffinement, l'***AtelierB***[39] génère des obligations de preuves. Là, on a deux hypothèses de travail :

- soit *AtelierB* réussit à exécuter la preuve demandée, et on n'a plus rien à faire,
- soit l'*AtelierB* en est incapable, et il faut, alors, le résoudre soi-même à l'aide de l'assistant prouveur.

Il faut noter que cette incapacité à prouver, peut déceler une impossibilité logique due à une erreur dans le raffinement. Après, et une fois que les obligations de preuves seront résolues, il ne restera plus qu'à raffiner la machine, et recommencer le jeu des preuves pour les nouvelles données. Et ainsi de suite, jusqu'à ce que l'on arrive à l'implémentation et la compilation de ce code. pour générer une application qui pourra être exécutée en toute liberté sans crainte de message d'erreur du genre « segmentation fault ».

Cela dit, il ne faut pas oublier que l'*AtelierB* n'a pas été prouvé. Il a été validé comme tous les autres compilateurs, c'est-à-dire à l'usage et par l'essai. En d'autres termes même si la specification en B a été parfaitement écrite, une erreur générée par le compilateur lui-même reste toujours possible.

[39] Une plateforme pour appliquer la *méthode B*.

Un intérêt majeur de la méthode B est la prise en compte, dans **un seul formalisme**, du développement incrémentiel d'une spécification, de ses raffinements successifs, jusqu'à l'implémentation correspondante.

Récemment plusieurs essais pour modéliser les circuits numériques en utilisant la méthode B ont été faits [70, 5, 2, 34, 43, 28].

En [78] Jean Louis Boulanger et Georges Mariano ont abordé le sujet : ils proposent quelques machines principales, pour représenter les composants, soit au niveau des portes logiques, soit au niveau des transistors. Ces composants sont la base vers laquelle la conception est orientée.

La conception commence par la spécification fonctionnelle du circuit combinatoire, dans une machine abstraite. Dans cette description, on décrit les entrées et les sorties du circuit. Chaque port est représenté par une variable interne, dans la machine abstraite. A cette variable est attachée une opération pour consulter sa valeur (sortie) ou pour modifier cette valeur (entrée).

A ce niveau-là, on déclare le but du circuit, comme relation logique entre les entrées et les sorties, mais les opérations ne précisent pas « comment » les modifications sont effectuées.

Pendant le raffinement, on décrit les composants du circuit en important et en réutilisant les circuits de base, ainsi que les autres circuits qui ont été déjà construits par l'utilisateur. La connexion entre deux ports est représentée par une équation entre les valeurs des variables correspondantes. La propagation d'un signal se fait en appelant successivement les opérations correspondan aux ports qu'il traverse. Le raffinement peut se répéter plusieurs fois pour déclarer tous les composants du circuit dans l'implémentation. Le concepteur peut **prouver l'exactitude de sa conception**, en utilisant un outil de la méthode B. Cet outil peut indiquer les obligations de preuves, et, il peut exécuter une partie des preuves nécessaires, soit automatiquement, soit en coopération avec le concepteur. Boulanger et Mariano ont utilisé le code B classique, accepté par l'*AtelierB* [40], pour évaluer ce travail. D'après leur expérience, presque 56% des preuves sont faites automatiquement, quand la conception était orientée vers les portes logiques.

Plusieurs aspects de conception du matériel, comme les aspects temporels, sont difficiles à traiter avec la notation B classique. J. Plosila, K. Sere et M. Walden ont proposé en [101] **d'enrichir la notation de B classique**,

afin de décrire ces aspects. Dans leur méthodologie la conception de circuit commence par une production d'un programme correct écrit en utilisant un système d'action, le B événementiel. Ensuite ce programme est compilé automatiquement, ou manuellement, en circuit en utilisant des règles de transformation, qui préservent l'exactitude de la conception, afin d'avoir un circuit logiquement correcte.

1.4 Co-design et Langages pour le co-design

En liaison de ce qui vient d'être discuté, sur la conception logicielle et matériel, la méthodologie classique de conception des systèmes hybrides, oblige à séparer la conception matérielle de la conception logicielle dès les premières phases de la spécification.

Les deux parties sont alors développées de manière totalement indépendante et concurrente, empêchant toute vérification de cohérence avant la phase d'intégration.

Par exemple les concepteurs de matériels ne savent pas apprécier les besoins, en ressources matérielles, des logiciels qui tourneront sur les processeurs. De même, les concepteurs de logiciels sous-évaluent, en règle générale, les ressources et la puissance des ressources matérielles, qui leur seront nécessaires. Malheureusement, seule la phase d'intégration est capable de mettre en lumière ces problèmes ou interrogations, Ceci engendre des coûts non négligeables, d'abord, pour la réalisation d'un masque du circuit de test, mais, ensuite en temps de développement si des modifications importantes sont à apporter.

Le co-design a donc été introduit pour atténuer, voire éliminer, cette séparation entre conceptions matérielle et logicielle, ou, tout au moins, retarder cette séparation le plus tard possible. **Le gain en productivité est évident puisque le co-design permet de détecter les erreurs de conception beaucoup plus tôt.**

Cette idée n'est pas nouvelle, par contre la mise en pratique de ce couplage, à chaque niveau de conception, de la spécification abstraite à la synthèse, est un développement encore trop récent, pour en démontrer la pleine puissance. Les deux questions principales sont :

- Comment peut-on intégrer deux disciplines très différentes, non seulement dans leurs techniques, mais aussi dans leurs culture ?

- Peut-on utiliser les résultats et les succès, dans le domaine du hardware, en matière de conception et de vérification, dans le domaine du software et donc pour le co-design ?

Il existe, pour cela, un certain nombre de projets, la plupart aux Etats-Unis. Parmi ces projets, nous citons :

- PTOLEMY (Lee et al., université de Californie à Berkeley)

- SpecSyh à Irvine (Gajski et al., 1994).

- CODES à Siemens (Buchenrieder et al., 1994).

- RASSP (Richards, 1994).

- Approche Thomas à CMU (Thomas et al., 1993).

- Approche Gupta et De Micheli à Stanford (Gupta et De Micheli, 1993).

- Projet européen PUSSEE (Paradigm Unifying System Specification Environments for proven Elctronic design, en cours)

Ce dernier projet n'est pas spécifique au co-design, même si ce domaine d'application est tout à fait évident. Toutes ces approches partagent l'idée de décomposer un système en trois parties :

- une partie **purement matérielle**, généralement implantée sous forme de circuits hardware (circuits programmables comme les FPGAs, circuits dédiés comme les ASICs, coprocesseurs, etc.),

- une seconde partie, **logicielle** : le programme qui sera exécuté sur le microprocesseur intégré au circuit spécifique, et enfin

- une **interface** de communication entre les parties matérielle et logicielle.

Les avantages que représente le co-design sont nombreux et dépendent du domaine d'application. En commençant par **une prise en compte glo-**

bale du système à modéliser, même si le partitionnement conduit à une conception modulaire, on peut mesurer en quoi les décisions, prises sur un module, influent sur le fonctionnement des autres. Ceci induit, aussi, à une **détection plus rapide d'erreurs** de conception, bien avant la phase de test, ou d'intégration, comme pour les méthodes classiques. Il faut noter, aussi, **la flexibilité de la conception**. Les frontières entre le logiciel et le matériel ne sont plus hermétiques, et il devient possible, de faire migrer une partie du code hardware, en un code logiciel, sans avoir à refaire tout le travail de conception. **Le temps de mise sur le marché**[40] **est plus court**, ce qui permet une meilleure réactivité, par rapport à certains enjeux commerciaux. Ceci induit aussi une **baisse du coût** de conception.

On peut distinguer deux types de domaines d'applications :

- Les applications pour lesquelles le hardware et le software travaillent de façon très étroite, avec des contraintes temporelles à respecter, applications pour lesquelles la rapidité d'exécution est primordiale.

- Les applications complexes, avec des critères de sûreté importants, et de traçabilité de ces critères, tout au long de la phase de conception, d'implémentation et même de maintenance, répondant éventuellement à certaines normes, rendues obligatoires dans la communauté européenne, dans des domaines sensibles, comme ceux du transport, de la médecine ou de la banque.

1.5 Co-Vérification formelle

La co-vérification doit permettre de vérifier les parties hardware et software d'un système embarqué [30], bien avant son implémentation physique [47]. L'exécution de la partie logicielle, sur un prototype virtuel de la composante hardware, peut permettre la mise en évidence de bugs et réduit le risque de problèmes d'incompatibilité entre le hardware et le software, dans le cycle de conception. Cette approche traditionnelle de « co-simulation » se heurte à la taille des systèmes à vérifier et notre réelle incapacité à réaliser cette co-simulation de façon efficace. Si on sait exécuter plusieurs milliers de lignes de code par seconde, la plupart du temps, seuls quelques dizaines de cycle sont simulables par seconde, pour la composante matérielle. Des outils,

[40] Time to Market

comme Seamless (MentorGraphics), essaient d'optimiser cette co-vérification, en établissant le dialogue hardware-software le plus haut possible, dans le simulateur, pour permettre l'exécution, asynchrone, de paquets de code logiciel. D'autre solutions prônent l'utilisation de langages semi-formels, comme *UML* suivi d'une compilation logiciel et d'une synthèse matérielle. C'est l'approche la plus étudiée ; elle consiste à adapter les plateformes de développement logicielle, pour permettre la modélisation de la partie matérielle.

Si les méthodes réellement formelles sont utilisées, aussi bien dans le domaine du logiciel que celui du matériel, elles n'occupent pas une place identique dans le domaine du co-design. Le logiciel dit embarqué pose des problèmes, qui ressemblent plus volontiers au matériel qu'au logiciel, à proprement parlé (coût et difficultés de mise à jour, ressources limitées en mémoire, en puissance, souvent pour des applications critiques, etc.). Il n'existe pas à l'heure actuelle de solutions satisfaisantes. Dans le domaine réellement formel, il faut citer le projet PUSSEE, qui propose une spécialisation « formelle » des langages de développement logiciel (*UML*) et matériel (*VHDL*, *SystemC*) vers le langage *B*. Ceci conduit à des restrictions très précises d'écritures, pour garantir que la traduction vers *B* sera « complète » et puisse s'intégrer à un projet de développement plus globale à la *B*. Si cette approche apporte une réponse cohérente à la spécification, au niveau système de systèmes embarqués, elle n'est pas, ou peu, prise en compte dans les problèmes temps réels et nécessite, pour les concepteurs, une très bonne compréhension et maîtrise de la méthode *B*. Nous avons choisi dans la suite de cette thèse une approche intermédiaire, même complémentaire au projet PUSSEE en proposant une traduction partielle et automatique du code *VHDL* vers le langage *B*, pour permettre un contrôle de compatibilité, de la traçabilité des exigences, même après partitionnement du développement. Cette solution n'est bien sûr que partielle, mais présente l'avantage de ne pas exiger une maîtrise, à la fois du *VHDL* et de *B*, à travers les contraintes fortes imposées par la bijection structurelle et sémantique.

Chapitre 2

Objectifs de l'outil

2.1 Introduction

Notre objectif principal pour la définition d'un tel outil de conception est de proposer un cadre commun d'architectures (logicielle, matérielle, etc.) prouvables par raffinement. C'est pourquoi nous nous sommes basés sur une analyse des différents modèles d'architecture utilisés et nous nous sommes placés à l'intersection de ces approches conceptuelles.

Traditionnellement, le concepteur peut intervenir bien avant la production du modèle. Il peut participer avec le client à la formulation des besoins (spécifications). Ensuite, il faut choisir une méthodologie de conception correspondant aux besoins et à la qualité souhaitée du produit.

Les choix méthodologiques de développement ont un impact direct sur la sûreté et sur le coût du produit : coût de développement et surtout coût de maintenance lié au debugging ou à l'évolution nécessaire du produit. Dans le but d'augmenter la sûreté, le développement par raffinement est un bon choix. Et c'est dans l'optique d'une réduction des coûts qu'intervient la notion de composants réutilisables d'une application à une autre.

Un outil de conception doit faciliter le travail d'équipe. Le (voire les) langage de spécification choisi doit être compréhensible et sans ambiguïté pour faciliter la tâche de développement d'un système de manière à vérifier la correspondance entre les besoins et l'implémentation. La description claire des besoins [65], ainsi que leur décomposition rend la coopération dans l'équipe plus facile. Cette décomposition conduit aussi à une implémentation par parties (en plusieurs modules ou composants) ce qui facilite la maintenance et

les modifications éventuelles. La conception générique permet une utilisation multiple du même composant ou module et rend la vérification plus simple ; particulièrement quand le système est de grande taille. Le développement de projet étape par étape, ou la conception par raffinement, augmente la sûreté de l'implémentation.

La méthodologie de conception doit respecter un certain nombre de propriétés :

2.2 Spécification, exigences, cahier des charges

La première étape dans la conception d'un système est de définir le but de sa production [34, 44, 74]. Les spécifications et les exigences ambiguës risquent d'une part, d'affaiblir la base sur laquelle la conception est faite. Et d'autre part, de ne pas permettre de vérifier la fiabilité (exactitude) de conception par toutes les méthodes de test. La responsabilité dans cette étape est partagée entre le client qui définit le cahier des charges [4] et l'équipe de développement qui analyse les informations données. Afin d'obtenir une version claire et compréhensible d'exigences, un dialogue entre le client et le concepteur peut s'avérer important pour plusieurs raisons :

- enlever les contradictions possibles,
- reformuler certaines informations ou d'en ajouter d'autres,
- restructurer l'information.

Le cahier des charges est défini[1] comme étant « un document qui spécifie de manière complète, précise et vérifiable, les besoins, le comportement, ainsi que toute autre caractéristique d'un système ou d'un composant, et aussi souvent, les procédures pour déterminer si ces requêtes ont été satisfaites. »

Ces requêtes peuvent comprendre :

- Les spécifications fonctionnelles qui précisent ce que le système doit faire et les tâches principales qu'il doit pouvoir exécuter. On pourrait indiquer, par exemple, ce qu'une boîte particulière doit pouvoir

[1] Le norme IEEE Std 610.122-1997.

exécuter : « ... allumer la lampe rouge de passage à niveau quand le train est à une distance de 1 km de cette station... » Ceci donne au programmeur une idée sur les entrées et sorties de cette boîte, et ce que le système doit pouvoir exécuter.

- Les spécifications non fonctionnelles s'intéressent aux propriétés qui améliorent le produit, telles que la sécurité et la maintenance

- les contraintes d'implémentation qui sont des conditions sur le produit final comme « la latence de signal dans un circuit doit avoir une certaine valeur, l'eau ne doit pas dépasser un certain niveau .etc. »

- les conditions de conception qui précisent dans quel environnement le concepteur va développer son produit ; par exemple, *LINUX, WINDOWS*, un environnement d'une certaine famille de circuits intégrés, etc.

- l'interface avec l'environnement qui inclut les entrées/sorties principales d'un système voulu : noms, types, domaines .etc.

La qualité du cahier des charges doit montrer clairement ce que le système doit faire, non comment il va le faire. Elle ne doit pas prendre aucune décision de conception. Parmi les éléments de qualité de spécifications on peut citer :

- **Consistance**[2] [25] :
  signifie qu'il n'y a pas de contradictions dans le modèle spécifié.

- **Complétude**[3] :
  Dans le sens qu'il fournit **toutes** les informations nécessaires à la description de l'objet et son bon fonctionnement.

- **Clarté** :
  la spécification doit être facilement compréhensible et non ambiguë pour que les différents utilisateurs ou concepteurs donnent la même interprétation à cette spécification. D'une part, la cause de l'ambiguïté peut être grammaticale, c'est-à-dire que les phrases sont mal construites. D'autre

[2]Consistency, qui est traduit aussi par le mot « Cohérence ».
[3]Completeness.

part, elle peut être une résultante d'un manque de détail impliquant plusieurs interprétations possibles. Si le premier de ces deux aspects peut être corrigé indépendamment du client, ce n'est pas le cas du deuxième aspect.

– **Robustesse** :
la spécification doit prendre en compte des conditions de fonctionnement anormales ou dégradées pour permettre, par exemple, une meilleure tolérance aux pannes.

Les méthodes formelles, reposant sur des bases mathématiques, permettent de maîtriser la sémantique des spécifications. Il existe des méthodes qui permettent de prouver la correction des développements. L'utilisation du langage mathématique augmente la qualité du cahier des charges, car il permet d'éviter l'ambiguïté inhérente aux langages naturels ou semi-formels. De plus, l'utilisation d'une méthode formelle permet la vérification et l'évaluation dès les premières phases de spécification. Dans le cas où le cahier des charges contient non pas seulement des besoins, mais des exigences ; (c'est-à-dire des besoins à satisfaire absolument, par exemple pour les systèmes critiques), on peut percevoir facilement tout l'intérêt de méthodes de vérification induites par l'utilisation de telles méthodes formelles. On pourra parler aussi d'ingénierie des exigences. Nous nous intéresserons plus particulièrement à la méthode *B*, fondée elle sur la logique de premier ordre et la théorie des ensembles.

Une bonne spécification peut aider aussi à la clarification et une meilleure compréhension du problème à traiter. Elle peut donner une indication au concepteur sur les composants d'implémentation pour son modèle. Cela nécessite l'existence des bibliothèques de composants d'implémentation, qui sont à leur tour bien spécifiées. Dans le monde des circuits intégrés, il existe des familles bien connues dont le fonctionnement est fiable. Pour chaque famille, plusieurs technologies d'implémentation sont disponibles. La spécification détaillée des propriétés de l'implémentation de ces circuits est préférable pour son utilisation dans un système donné, même si ces propriétés ne sont pas fonctionnelles, comme l'énergie consommée.

Dans notre projet, on utilise des unités mixte *VHDL/B* pour déclarer le système. La spécification sera définie dans un composant *VHDL* abstrait « *ENTITY* » qui peut avoir plusieurs propriétés décrites formellement en *B*.

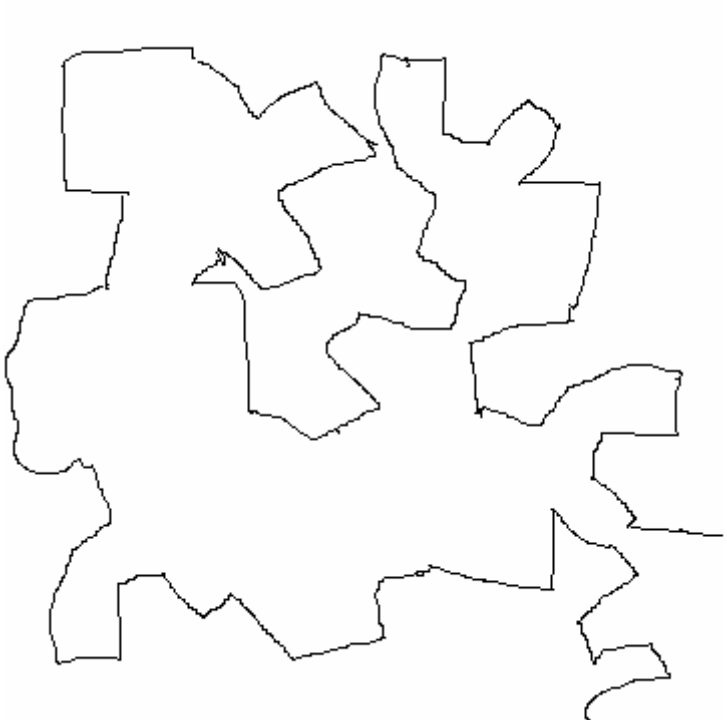

FIG. 2.1 – « Contour » du cahier des charges.

2.3 Décomposition

Pendant les premières étapes de développement on essaie de décomposer le projet en parties plus petites en précisant les spécifications de chaque partie et en définissant les interfaces entre elles. Ce qui permettra de déclarer les composants du projet. Ces composants sont des éléments relativement indépendants qui peuvent communiquer entre eux à travers des interfaces. Ils permettent, par assemblage, de construire une structure. C'est le cas par exemple d'un Objet en *LOO*, un composant logiciel en Wright (*ADL*), une machine abstraite en *B*, device en *VGUI*, etc. Le projet est composé de plusieurs objets. L'objet de base, le composant, a une interface bien définie. Cette interface précise comment les objets peuvent communiquer entre eux. En *VHDL*, on définit les entités dans lesquelles on déclare les spécifications des composants. Les connexions avec les autres entités sont définies en précisant les signaux intermédiaires, leurs types, ainsi que leurs directions.

Dans notre méthodologie, la conception commence par un composant dont les propriétés sont définies formellement. Ensuite, on partitionne ce

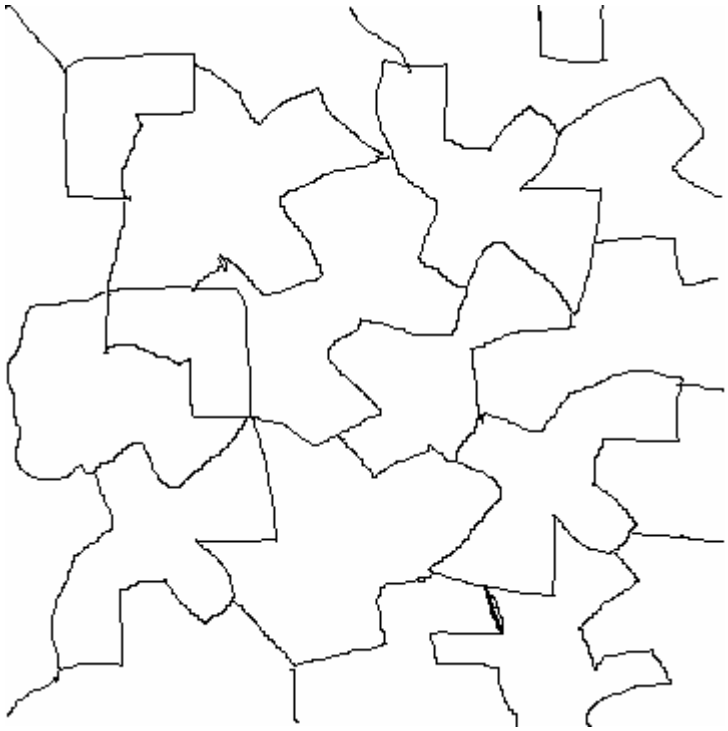

FIG. 2.2 – Décomposition.

composant en plusieurs sous composants. Les interfaces entre les nouveaux composants sont des entités en *VHDL* avec la spécification fonctionnelle en *B*.

2.4 Modularité

Il est honnête de dire que la notion de modularité regroupe, suivant les disciplines ou les sciences, un sens assez différent. Dans le domaine du génie logiciel et tout particulièrement celui des langages orientés objet, le module est une **unité logiciel générique bien identifiée**, une classe par exemple. Par contre les débats sont très animés, par exemple en biologie ou dans les sciences cognitives, sur la définition et l'opérabilité de la modularité, sans parler des mécanismes qui peuvent être liés à ce sujet comme par exemple dans le domaine du raisonnement. Nous différencierons notre interprétation du module en fonction du niveau d'abstraction auquel nous sommes. Au ni-

veau le plus haut, c'est-à-dire l'entrée de conception[4], nous nous attacherons à considérer un module assez informellement comme un ensemble d'objets regroupés par fonction et niveau d'interaction. Même si la décomposition modulaire permet aux concepteurs de faire du management de projet et un partitionnement efficace du travail. Il faut veiller à ne pas séparer les unités qui peuvent être similaires, voire identiques, et veiller dans ce partitionnement à regrouper par modules les éléments ou composants de même nature ou en forte dépendance. En d'autres termes, le module est plutôt le résultat, dans une approche descendante, d'un partitionnement dont les contours peuvent être raffinés pour une meilleure généricité, ou réutilisabilité. Un package ou une librairie de composants apparaît clairement à ce niveau d'entrée de spécification comme un module.

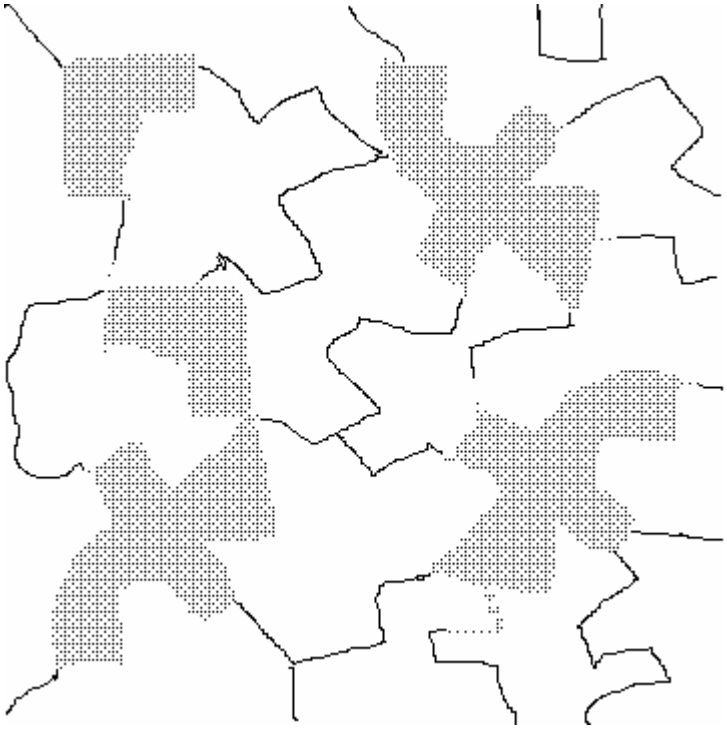

FIG. 2.3 – Modularité.

Les frontières ou le nombre de modules pourront évoluer car, les développements de ces unités peuvent aussi se ressembler, voire partager, une

[4]System Level.

partie importante de leurs comportements, même si chacune a sa spécificité.

A l'opposé au niveau logiciel, le module (qui représente la classe, rappelons-le) est utilisé pour donner des services au monde extérieur par une interface bien définie. Ces dernières années, le développement de langages de programmation orientés objet a considérablement augmenté l'impact du développement de modules et la réutilisation de codes. De tels langages permettent au programmeur de définir les classes (des unités de modularité), des objets qui se comportent d'une façon contrôlée et bien définie. Du point de vue historique, cette réutilisation a toujours existé sous forme de bibliothèques de fonctions externes appelables par le programme principal (bibliothèques mathématique, graphique, etc.). Cette réutilisation de parties de codes peut être qualifiée de marginale au sens propre du terme, car les parties communes à plusieurs applications sont extérieures à la structure du logiciel.

D'une manière générale, dans les *LOOs*, les composants d'un module partagent l'accès aux données communes, et l'interface fournit l'accès contrôlé à ces données. Chaque composant masque son comportement par encapsulation et n'autorise certaines actions ou partages de données qu'explicitement à travers une interface.

FIG. 2.4 – Bibliothèques.

En utilisant la modularité on peut en tirer plusieurs bénéfices [114, 31] :

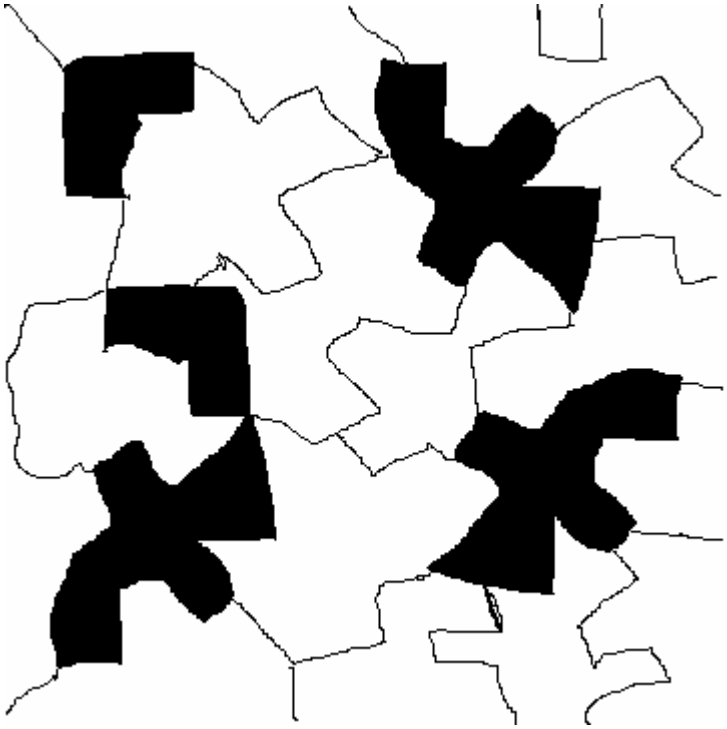

FIG. 2.5 – Implementation.

– il devient possible de définir une application de façon plus incrémentielle. Chaque module peut être utilisé pour construire une unité qui constituera à son tour un autre module.

– On peut réduire le temps de conception en développant un seul module au lieu de plusieurs composants.

– On diminue le temps de test parce qu'on vérifie un seul module à la place de plusieurs pour ses sous-composants, ce qui aide à construire la conception sur une base fiable de code mieux testé.

– La modularité augmente la portabilité, la capacité de prendre une partie de conception déjà produite et le réutiliser dans d'autres environnements. Cette propriété est particulièrement importante quand plusieurs technologies d'implémentation sont utilisées. Dans les systèmes embarqués, par exemple, il se peut que toutes les unités logicielle/matérielle

d'une architecture technique choisie doivent être aménagées sur la nouvelle plateforme. La modularité facilite cette action.

En *VHDL*, l'*ENTITY* spécifie l'interface externe de l'unité. Plusieurs composants peuvent être déclarés dans l'*ARCHITECTURE* pour construire l'implémentation attachée à cette entité. Cette idée ressemble aussi au *BINDING* en *Wright*, le niveau de *DEVICE* en *VGUI* et les opérations en *B*. Dans notre projet, on utilise les unités génériques de *VGUI*. La spécification de ces unités est donnée une seule fois, mais peut être utilisée plusieurs fois comme boîte noire. De l'autre côté, on a défini une bibliothèque utile std_logique_1164 pour montrer la possibilité d'importer les packages qui sont déjà définis en *VHDL* dans notre méthodologie.

Une forme de modularité très populaire est celle de la conception par assemblage de composants COTS[5]. L'utilisation de composants commerciaux disponibles immédiatement comme éléments de plus grands systèmes devient de plus en plus banale. Les budgets craintifs, l'amélioration constante de la qualité des COTS et les conditions d'extension des systèmes conduisent ce processus : le passage du développement classique aux systèmes basés sur les COTS.

2.5 Hardware/Software

Historiquement, la conception de logiciels et la construction de circuits électroniques s'effectuaient de manière totalement indépendante, cette dernière traitant des fonctions relativement simples par rapport aux logiciels, mais de façon notamment concurrente [7].

De plus, les mesures de qualité de conception matériel étaient indépendantes de celles du logiciel [82], comme la fiabilité, la latence, le coût, la taille, l'énergie consommée et la stabilité par rapport au temps.

Avec le développement de conception associant logiciel et matériel des progrès communs sur plusieurs aspects ont été apportés [7, 73].

D'une part, le développement de logiciels a inspiré les notions de réutilisation, de modularité, de maintenance. Et d'autre part, le développement matériel a ainsi rapproché celui relatif au logiciel grâce à plusieurs paramètres :

[5]Commercial Off-The-Shell.

- La qualité de la modélisation logique puis structurelle des circuits électroniques a permis aux concepteurs de matériel de s'affranchir des problèmes de nature physique, la synthèse assurant l'automatisation du modèle physique du circuit en terme de transistor. Suivant cette modélisation logique, les entrés/sorties de circuits sont vues comme variables typées et il est devenu possible de définir le comportement d'un circuit par une fonction logique.

- La création de langages standard de conception de matériel, les *HDLs*, a généré une autre ressemblance de conception dans les deux domaines. Par exemple, un *HDL* a la structure de langage impératif. Ils peuvent contenir des boucles, des conditions, des fonctions, etc.

- Le développement de technologie d'implémentation, adapté à des dizaines des millions de composants de transistors, a obligé les concepteurs, pour prendre en compte cette complexité, à utiliser des différents niveaux [111] d'abstraction, adopter le raffinement au moins de façon informelle et commencer le design au niveau plus haut. Dans le niveau le plus haut, les paramètres de conceptions ne sont pas liées à des technologies particulières.

- La technologie de SoC[6] [35] , a changé la vision traditionnelle de circuit. On développe un système hybride, logiciel/matériel, sur un seul chip. Cela nécessite l'existence d'outils qui peuvent modéliser ces deux technologies conjointement.

Il y a plusieurs aspects communs dans l'abstrait entre la conception hardware/software. Mais souvent les deux domaines sont séparés. Un des objectifs de notre outil est de retarder le plus possible le choix technologique. Si par la suite, on est amené à raffiner la spécification d'un modèle dont une partie de l'implémentation est logicielle et une autre partie est matérielle, il faut donner la possibilité à chaque implémentation de garder une visibilité au moins partielle des autres implémentations concurrentes, ceci afin de pouvoir faire de la détection d'erreurs et aussi pour permettre une migration d'une technologie vers une autre. Cet aspect est très utile dans le domaine du codesign par exemple dans la conception de systèmes embarqués, pour faire migrer des composants logiciel vers des composants $ASIC$[7], pour faire de la copie partielle du matériel vers le logiciel. Le choix matériel se justifie en cas

[6]System On Chip.
[7]Application Specific Integrated Circuits.

de recherches de performances, le choix logiciel pour des raisons de coûts et de mise à jour. Par contre, le choix matériel *et* logiciel permet dans le cas d'exigences de sûreté une la redondance matériel-logiciel est importante pour une meilleure tolérance aux pannes.

2.6 Hiérarchie

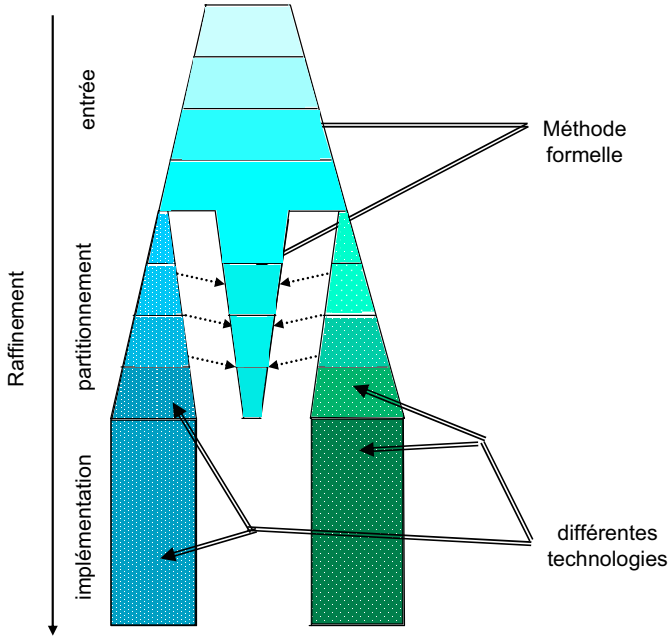

FIG. 2.6 – Hiérarchie.

Le processus de conception modulaire, allant de la spécification abstraite jusqu'à la réalisation, nécessite une hiérarchie claire et précise. Dans le domaine de conception du matériel, quatre niveaux d'abstraction peuvent être distingués [36] :

- **Niveau transistor**,
  La conception est faite à ce niveau pour optimiser la taille du circuit dont le modèle a déjà été établi. On essaie également de diminuer la longueur des fils reliant les différents transistors.

- **Niveau porte**,
  Le but de ce niveau est de vérifier que la consommation d'énergie du circuit ne dépasse pas les limites prévues dans la spécifications . Il est utilisé aussi pour optimiser quelques paramètres comme la latence de transmission de signal, un paramètre qui est inconnue dans les niveaux les plus hauts. A ce niveau on peut traiter des circuits de tailles plus importantes, mais avec moins de précision qu'au niveau transistor.

- **Niveau de transfert entre registres**,
  A ce niveau le circuit représente plusieurs unités opérationnelles gardant ou traitant les données. C'est un niveau structurel dans lequel la conception se concentre sur les interconnexions entre les différentes unités.

- **Niveau algorithmique ou comportemental**,
  C'est le processus de la conception au niveau le plus abstrait où le circuit est spécifié comme étant un algorithme qui sera implémenté dans le matériel. La conception *ASIC* commence à ce niveau d'abstraction.

On remarque que les niveaux de conception ne sont pas liés aux couches de spécifications (les besoins). De telles couches sont importantes pour faciliter les tests de sûreté de conception. En plus, dans les systèmes hétérogènes, ces couches sont nécessaires à l'expression des propriétés communes de conception.

La hiérarchie de *ADL* permet de déclarer les composants, ensuite d'en assembler plusieurs pour obtenir un composant plus complexe qui sera utilisé à un niveau supérieur.

La hiérarchie dans notre méthodologie est principalement composée de trois niveaux d'abstraction [11, 97, 86, 104] :

1. **l'entrée de conception**,
   C'est le premier niveau, et le plus abstrait. Le processus de conception à ce niveau demande l'insertion formelle étape par étape de la spécification fonctionnelle et toutes les propriétés génériques du système. On peut aussi ajouter à ce niveau des spécifications opérationnelles. Il ne doit contenir aucun choix technologique. La propriété de consistance doit être vérifiée pour chaque étape de ce niveau. On essaie de prolonger ce niveau au maximum en essayant d'y ajouter la description la plus complète possible de système. La décomposition modulaire successive

de chaque unité en plusieurs unités combinatoires est l'aspect le plus important dans ce niveau [117, 93].

2. **Le partitionnement**,
   La description à ce niveau là est un raffinement de la description du niveau d'entrée. On peut voir à ce niveau des branches séparées de conception. Chaque branche est liée aux différents aspects de l'implémentation (logiciel, matériel .etc.). Plusieurs aspects communs de description existent encore dans les différentes branches de ces technologies. On doit garder des traces de ces aspects soit en communicant entre les différents processus de développement, soit en collectant toutes ces informations dans un axe commun de conception. Plusieurs bénéfices peuvent être obtenus de cette collection :

   – garder la possibilité de vérifier la compatibilité.

   – possibilité de migration d'une technologie vers une autre

   – possibilité d'exécuter une co-simulation à n'importe quel point du développement.

3. **Niveau d'implémentation**,
   C'est le dernier niveau de développement.

La conception se fait par couche. Chaque composant a une représentation externe, sa spécification attendue et une représentation interne ou architecture ; c'est-à-dire l'assemblage de ses sous composants qui eux aussi peuvent avoir deux représentations, et ainsi de suite. Ceci facilite grandement la décomposition et l'abstraction lors de la conception descendante d'un système.

2.7 Généricité et Spécialisation : Style/architecture/raffinement

Cette notion est présente dans la plupart des environnements de développement et est très directement liée à la notion de modularité et d'abstraction [29, 87]. Proposer des modules plus abstraits, c'est mieux maîtriser le développement par factorisation de notions ou d'actions communes, et c'est assurer une meilleure réutilisation avec comme objectifs la maintenance et la

réutilisabilité [92]. Il faut néanmoins être réaliste sur ce dernier objectif : la réutilisabilité reste assez marginale à l'exception du hardware et des circuits électroniques qui reste un référence difficilement égalable. C'est le cas de la notion de classes abstraites en *LOO*, de *ENTITY* et *ARCHITECTURE* en *VHDL*, Style et Configurations en Wright, et la machine abstraite et son raffinement en *B*.

2.8 Correction

C'est la capacité d'un logiciel à produire les résultats conformes à ses spécifications.
Être capable de montrer qu'un système répond correctement à ses spécifications est une tâche extrêmement difficile si ce n'est impossible même en cas d'utilisation de méthodes de démonstration formelles [55]. Prenons n'importe quelle nouvelle version d'un système d'exploitation ou d'un logiciel d'application, les bugs recensés par le constructeur et signalées lors de la première livraison, se comptent par centaines ! Étant donné les responsabilités croissantes que l'homme confie à des systèmes automatiques (pilote automatique d'avion ou de métro, échanges boursiers, argent électronique, monitoring, .etc.) le critère de correction est de toute première importance. C'est une des bonnes raisons qui militent en faveur de la réutilisation de composants logiciels. On peut espérer diminuer le nombre d'erreurs, en développant peu de nouvelles lignes de code et en réutilisant des composants certifiés.
Dans ce contexte de réutilisation, il est évidemment souhaitable que la spécification d'un composant logiciel fasse partie intégrante de son code afin de ne pas réutiliser de manière aveugle des composants farfelus.

2.9 Développement par raffinement

C'est un développement incrémentiel de spécification qui se concrétise en préservant l'ancien comportement [28, 11, 10, 32]. Il se fait, par exemple, en précisant les domaines des données ou en diminuant le non déterminisme.

Dans le développement par raffinement on distingue les aspects suivants :

– conception descendante, comme *ADL*,

– possibilité d'ajouter des propriétés de sécurité et de sûreté,

– preuves indépendantes pour chaque étape de raffinement,

– validation et vérification de conception.

La conception par raffinement est une approche qui permet de produire graduellement un programme correct. On sépare les spécifications et le code logiciel, c'est-à-dire les spécifications (qui relève d'abord du client) de l'implémentation qui sont décidées par le développeur. Un long processus de développement peut exister entre la spécification obtenue à partir des exigences et une implémentation efficace. Cette opération peut être divisée en plusieurs étapes successives. Dans chaque étape, on développe une nouvelle entité qui doit être un raffinement de l'entité précédente. Les décisions de conception peuvent être introduites une par une dans chaque étape de raffinement. Cette méthodologie rend chaque raffinement petit, compréhensible et « manipulable ». Les exigences peuvent être plus complexes et déclarées totalement dans la spécification initiale. Néanmoins, il arrive que certaines exigences, d'apparence non principales pour le client, soient déclarées pendant les derniers raffinements. A titre d'exemples, citons la connexion de certains ports inutilisés dans le circuit au voltage zéro ou le besoin d'allumer une certaine lampe pour indiquer le fonctionnement d'une partie de circuit.

Le raffinement en B

Dire que Q est un raffinement de P signifie que tout comportement possible de Q est inclus dans ceux de P. Il se peut que le comportement de P soit non déterminisme. Dans ce cas, le comportement de Q va réduire ce non-déterminisme.

Dans la méthode *B*, un projet est constitué de plusieurs composants. Chaque composant contient plusieurs clauses. Les effets de raffinement peuvent être inclus dans plusieurs de ces clauses.

Dans les clauses *INVARIANT* et *INITIALISATION*, par exemple, la première clause permet d'exprimer sous forme d'un prédicat le typage de variable d'état du composant d'une part, et d'autre part, de déclarer des propriétés sur ces variables. Dans la deuxième clause, *INITIALISATION*, toutes les variables de composant doivent être initialisées. Il faudra prouver que l'initialisation vérifie le prédicat « invariant ».

Supposons que P soit une machine et Q un raffinement de cette machine. Dans ce cas, la clause *INITIALISATION* dans Q doit initialiser les variables de P de telle sorte qu'elles prennent des valeurs admissibles dans l'*INITIALISATION* de P.

Exemple,

```
MACHINE
    train
VARIABLES
    vitesse
INVARIANT
    vitesse : N & vitesse < 500
INITIALISATION
    vitesse:(vitesse> -10 & vitesse <+10)



REFINEMENT
    train_ref
VARIABLES
    vitesse
INVARIANT
    vitesse : N & vitesse < 500
INITIALISATION
    vitesse = 0
```

Les *HDLs*, comme les langages impératifs, permettent de déclarer la conception, et quelques fois ils autorisent certains types de raffinement même si ceux-la n'ont pas été bien précisés. En *VHDL*, par exemple, la déclaration de conception d'un circuit se fait en deux étapes : la déclaration de l'*ENTITY*, ensuite la déclaration de l'*ARCHITECTURE*.

Dans la première on définit l'interface externe du circuit, et dans la deuxième on définit sa structure interne. Mais le passage de la première déclaration à la seconde se fait par une seule étape de raffinement et non pas en plusieurs.

La conception de circuits numériques peut contenir deux types distincts de raffinement : le raffinement de l'interface et le raffinement de l'architecture.

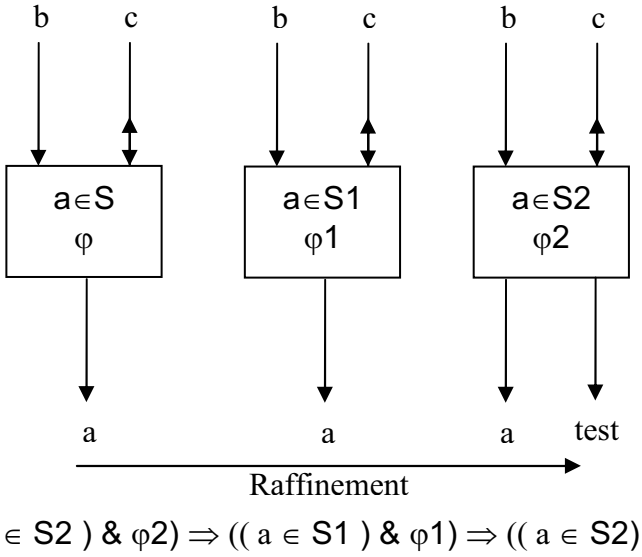

FIG. 2.7 – Raffinement et entité.

Le raffinement de l'interface concerne la connexion du composant avec son environnement Les spécifications logiques qui apparaissent dans chaque boite doivent être interprétées ici comme des propriétés d'invariant. Le domaine de la variable est réduit à chaque raffinement. On peut, par raffinement, ajouter de nouvelles variables. Cette interface est définie de façon incrémentielle, c'est-à-dire que pendant le développement on conserve les ports qui sont déjà déclarés et on peut en ajouter d'autres. On peut déclarer le type de chaque port et la fonction logique du circuit associant . Supposons que le comportement d'une entité raffinée est nommé `Comp`, et le comportement du composant raffinant `CompR`. Le raffinement doit alors satisfaire la condition suivante :

$$(Comp\_R \,\&\, i\_R \;:\; type\_i\_R) \;\Rightarrow\; (\, Comp \,\&\, i \;:\; type\_i)$$

D'un autre coté, le raffinement d'architecture (raffinement interne) se fait par déclaration de la structure interne de chaque composant. Cette structure contient des composants instanciés de modules déjà définis et d'autres, pouvant être déclarées ultérieurement.

On déclare des composants instanciés d'anciens modules et des nouveaux composants pour d'éventuelles nécessités. Pour chaque composant le comportement de composant i est décrit par Comp_i et le comportement total est spécifié par Comp.

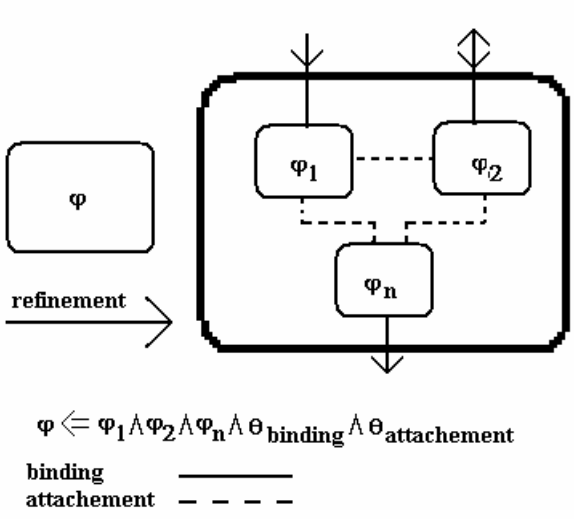

FIG. 2.8 – Raffinement et architecture.

Les *BINDINGS*, en *ADL*, représentent les liaisons entre les variables externes définies dans l'interface et celles internes définies dans l'architecture. Les *Attachements* expriment les liaisons entres les sous-composants de l'architecture. Ces égalités entre variables sont appelées ici *invariants de collage*.

$$\sum Comp_i \; + \; invariants\ de\ collage \; \Rightarrow \; Comp$$

2.10 Preuve automatique de développement

L'utilisation de spécification formelle rend possible la preuve de propriétés dès les premières phases de conception [37]. Il faut pour cela :

- **Générer des obligations de preuves**
  Celles qui sont nécessaires pour vérifier la consistance de chaque unité logique, et pour vérifier la correction du raffinement.

- **Exécution des preuves**
  Les preuves peuvent être assez compliquées, et leur génération manuelle quasi impossible. L'utilisation d'un outil qui sait générer les obligations et les preuves facilite grandement la tache du concepteur. L'*AtelierB* peut exécuter cette tache sur les programmes écrits en *B*. C'est l'outil industriel qui a permis une utilisation opérationnelle de la méthode *B*

pour des développements logiciels prouvés. Il a notamment été utilisé pour le développement des automatismes sécuritaires du métro automatique METEOR par Siemens.

La méthode *B* permet :

- les vérifications syntaxiques des modèles de *B*, la génération automatique des théorèmes à démontrer, la traduction automatique des modèles *B* bas niveau vers les langages *C* et *ADA* et une aide pour démontrer automatiquement les théorèmes,

- une aide au développement : gestion automatique des dépendances entre composants *B*, bibliothèques réutilisables, génération de documentation et génération de métriques.

L'utilisation du raffinement et de son mécanisme de preuves par étape a plusieurs intérêts immédiats :

- avoir un circuit sûr prouvé formellement et non pas des circuits prouvés statiquement correct à 99% par les méthodes de test.
- Identifier lors de la phase de conception les erreurs le plus tôt possible. Plus une erreur est trouvée tôt moins la correction coûte.
- Faciliter les preuves *composables* ou *distribuées*. En fractionnant les preuves à chaque étape de raffinement, on leur rend plus compréhensible et plus facile à vérifier de façon automatique ou semi-automatique.
- Avoir des obligations de preuves beaucoup plus simples que celles nécessaires au niveau de la simulation pour la vérification du système global.

2.11 Aspect graphique

La majorité des langages de conception sont des langages textuels. L'aspect graphique améliore la visualisation du développement et facilite la compréhension d'un modèle. Pour chaque domaine de conception (matériel, logiciel, .etc.) il y a des outils spécifiques. AcmeStudio, par exemple, est un outil de visualisation et édition pour *ACME*, la plus célèbre *ADL*. Avec AcmeStudio, on peut définir des nouvelles familles (bibliothèques) d'*ACME*, qui peuvent être utilisées pour d'autres applications. Il nous permet de créer les unités de conceptions et de les attacher à d'autres unités de deux manières :

– **Connexion interne,**
l'unité est à un niveau inférieur au niveau actuel de conception. Elle constitue avec les composants auxquels elle est connectée une unité plus complexe.

– **Connexion externe**,
l'unité représente une partie des composants de conception actuelle.

Beaucoup d'outils de conception de matériel, comme *OrCad PSpice*, ont des interfaces graphiques qui peuvent avoir :

– Des symboles spéciaux pour les éléments électriques élémentaires.

– Des bibliothèques de circuits standard avec leurs formes propres et leurs entrées/sorties standard.

– La possibilité de connecter les composants par les fils.

– Quelques éléments graphiques pour simuler le comportement de l'implémentation ; sources virtuelles d'énergie, histogrammes, interrupteurs, etc.

– La possibilité de produire un masque de circuit imprimé. Ce masque est utilisé pour produire une plaque de connexion sur laquelle on installe les portes réelles.

La majorité des outils qui ont ces propriétés graphiques, comme *Tkgate*, par exemple, traitent les circuits au niveau porte et transfert de registre, mais pas au niveau transistor et algorithmique. Car **la conception électronique est souvent considérée comme assemblage des composants qui sont déjà fabriqués** et pas une opération de développement à partir de cahier des charges jusque à l'implémentation. Cette idée a influencé les aspects graphiques des outils de conception. Ce qui a produit comme conséquence que beaucoup de ces outils permettent la composition de plusieurs éléments pour construire un élément plus complexe dans un niveau plus haut « *bottom-up* ». Par contre il ne permet pas la déclaration d'un module dont la structure est partiellement spécifiée ce qui est essentielle pour la conception « *top-down* ».

A la recherche d'interface pour notre outil qui facilite la conception spécialement au niveau *entrée de conception*. Notre choix s'est tourné vers *VGUI* pour les raisons suivantes :

- La possibilité de déclarer des unités modulaires. C'est-à-dire ; d'abord on déclare l'interface une par unité, puis ses composants. Il fait partie des outils qui autorisent la déclaration de boîte noire et la précision de ses composants. Ceci s'adapte bien à une approche hiérarchique descendante,

- La possibilité de déclarer des composants modulaires,

- Il est gratuit, et ses concepteurs ont adapté leur outil à certains de nos besoins (la version *VGUI* 2.0 est née des suggestions et souhaits que nous avons formulés),

- sa simplicité d'utilisation et les discussions que nous avons eues avec son développeur, Carl Hein (Lockheed Martin),

- La possibilité de déclarer les propriétés combinatoires du projet et de toutes les unités de conception.

2.12 Modélisation de temps

La représentation de temps est un aspect important dans la conception de circuit au niveau bas. De ce point de vue, il y a deux types principaux de circuits : Synchrone et asynchrone. Cette division est importante pour traiter le temps réel ou conceptuel ainsi pour analyser les boucles, les mémoires, etc.

- **Le temps conceptuel :**
  La propagation de signal ne dépend pas du latence de signal dans chaque composant. Il dépend de la connexion entre les composants : chaque composant donne ses résultats quand toutes ses entrées sont affectées (modèle événementiel, CSP).

- **le temps réel :** Il existe une horloge globale. Les sorties d'un composant dépendent de l'histogramme de ses entrées et de sa spécification.

D'un côté, Boulanger et Mariano ont traité le temps en utilisant une horloge globale et en considérant chaque signal comme étant une matrice de taille [2,n]. Les champs [1,..] représentent le temps. Il contient des valeurs successives du temps dont la distance est fixe. Par contre les champs [2,..] représentent les valeurs de signal. Pour un indice donnée i, Le champ [2,i] contient la valeur de signal au moment [1,i].

De l'autre coté, on a vérifié la possibilité de représenter les circuits asynchrones en enrichissant le domaine des valeurs par une valeur supplémentaire « unknown ». les valeurs initiales de tous les entrées de circuits asynchroné doivent avoir cette valeur. Le déclenchement de pré-conditions de chaque circuit est préconditionné par le changement de toutes les valeurs « unknown » de ces entrées.
Puisque on concentre dans notre thèse sur la conception au niveau système, c'est-à-dire au niveau plus haut de conception, nous ne chercherons pas traiter la modélisation du temps.

Chapitre 3

Un outil de Développement

3.1 Introduction

Dans ce chapitre, nous allons présenter les résultats de nos recherches dont le but principal est de proposer un outil de conception intégrant les principes de plusieurs outils et méthodologies existantes actuellement.

Le standard pour la description de matériel, *VHDL*, permet de décrire les circuits électroniques à plusieurs niveaux d'abstraction. Plus que l'abstraction augmente, plus que les aspects proprement matériels s'estompent et les concepts deviennent utilisables pour décrire des autres types de systèmes. *VHDL* est d'ailleurs présenté aussi comme un langage puissant et général ; Plusieurs aspects de *VHDL* peuvent être considérés comme génériques et peuvent être partagés entre plusieurs types de systèmes :

- La séparation de la conception en plusieurs étapes, *ENTITY*, *ARCHITECTURE* et *CONFIGURATION*,

- la possibilité d'avoir plusieurs versions de descriptions de la même architecture : fonctionnelle et combinatoire.

Les langages de description d'architecture de logiciels, *ADL*s, sont focalisés sur la description de conception au **niveau le plus haut du système** étant vue comme une collection hiérarchisée de **composants**, de connecteurs définissant des relations entre ces composants. Cette **hiérarchie combinatoire** peut être employée pour déclarer des objets de taille très variée des plus simples ou plus complexes dans tout type d'application : systèmes, matériel-logiciel, embarqué, .etc.

Par exemple *VGUI* est un **outil graphique** qui était conçu comme un outil de description matériel dans sa première version. Avec quelques modifications légères, il est employé pour une conception plus générique, c'est à dire tant logicielle que matérielle. Le fait que *VGUI* propose un modèle combinatoire qui n'intègre pas le temps facilite grandement un tel rapprochement.

De son coté, la méthode ***B*** permet d'avoir une **description mathématique non ambiguë**, au niveau système. Elle concrétise aussi les principes de conception par **raffinement**. Ainsi, elle permet de prouver la conception de façon progressive et modulaire. L'idée de prouver le développement est applicable dans le domaine de circuits électroniques.

L'utilisation de composants « corrects » facilite les preuves, et peut diminuer le temps de test spécialement dans les domaines électroniques où la réutilisation est un aspect important et où la méthode traditionnelle de test par la simulation peut prendre un temps prohibitif et donc est inutilisable en pratique pour des circuits complexes.

Les fortes différences en terme d'objectifs et de technologies des langages précédemment cités pèsent sur leurs concepts et leur structures. Malgré ça, la modularité, la décomposition, l'hiérarchie, la réutilisation, la sûreté sont des principes importants dans tous ces langages.

Dans cette thèse, nous allons tenter d'assembler non pas des composants logiciels et matériels, mais les langages eux-même afin d'avoir un outil, ou une méthodologie de conception mixte. Cette conception doit être générique au niveau le plus abstrait, ou « haut », mais pouvoir se traduire en différentes technologies d'implémentation possibles au niveau bas.

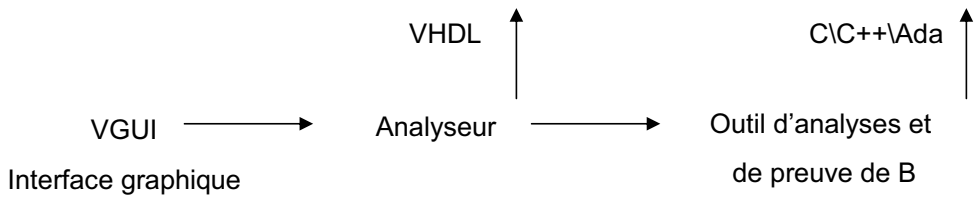

FIG. 3.1 – Schéma global de la plateforme BHDL (1).

La conception commence par une étape générique « l'entrée de conception » le concepteur peut utiliser un outil graphique, *VGUI*, comme interface

pour son développement. Le concepteur peut dessiner un schéma basé sur l'assemblage de composants génériques mais aussi matériels et logiciels.

VGUI produit des fichiers graphiques de diagrammes de conceptions (en particulier les fichiers d'extension *dia*). Dans ces fichiers on garde non seulement les propriétés graphiques, comme la taille de chaque boite et sa position dans le diagramme, mais aussi les propriétés logicielles et matérielles. Ces fichiers peuvent servir comme bibliothèque des projets ; Le concepteur peut réutiliser un projet, pour en inclure une partie dans un nouveau projet.

Nous avons étendu l'utilisation de *VGUI*, en collaboration avec ces créateurs, pour générer du code *BHDL*. C'est à dire du code mixte *VHDL* et *B* [23, 12]. Le concepteur d'un projet peut spécifier directement dans *VGUI* les propriétés qu'il souhaite exprimer, ensuite, *VGUI* les reproduit en *BHDL* (Voir figure 3.2.).
Ce code est ensuite analysé par un traducteur écrit en *ANTLR*[1]. Après plusieurs étapes d'analyse, ce code est analysé et traduit sous deux formes différentes : *VHDL* et *B*. Le code *VHDL* contient la partie purement *VHDL* correspondant aux instructions au modèle standard *VGUI*. *VGUI* n'a pas en effet la même richesse de modélisation de VHDL et ne génére que **la partie structurelle** *VHDL* sans traiter la partie algorithmique de *VHDL*. Nous parlerons de *VHDL-VGUI* dans la suite pour caractériser cette partie de *VHDL*. Il faut noter néanmoins que le traducteur que nous avons développé peut être appliqué sur n'importe quel code *VHDL* aussi riche soit-il et générera deux arbres : la partie du code *VHDL-VGUI* et sa traduction en B. Le code *B* contient les mêmes propriétés communes de *VHDL*, en plus il contient les spécifications qui ont été rajoutées via *VGUI* (voire directement dans le code *VHDL*) sous forme de commentaires. La qualité du code *B* généré et notre capacité à détecter des erreurs de conception sont directement liées à la qualité de ces spécifications elles-mêmes.

Le code *B* est transmis à l'*AtelierB* ou à *BToolkit*[2]. Cet outil est capable d'analyser le code fourni et de vérifier qu il n'y a pas de contradiction dans la conception. En plus il peut prouver la satisfaction de certaines propriétés. Il peut générer les obligations de preuves et en prouver une partie automatiquement.

Certains bibliothèques en *VHDL* sont nécessaires pour la spécification des projets. Afin d'exécuter les preuves de ces projets, on a créé des correspon-

[1] Voir page 95.
[2] Ces outils sont expliqués plus tard dans ce chapitre. Voir page 85.

dants de ces bibliothèques en *B*.

Le concepteur peut décrire le projet en *VGUI*, ensuite il l'analyse afin de prouver ses propriétés. Il peut aussi décrire le projet étape par étape et avec des analyses intermédiaires afin, d'un coté, de détecter les erreurs le plus tôt possible et, de l'autre coté, de diminuer la taille de preuves en remplaçant la nécessité d'une preuve complète d'un système par celle d'une arborescence de preuves modulaires.

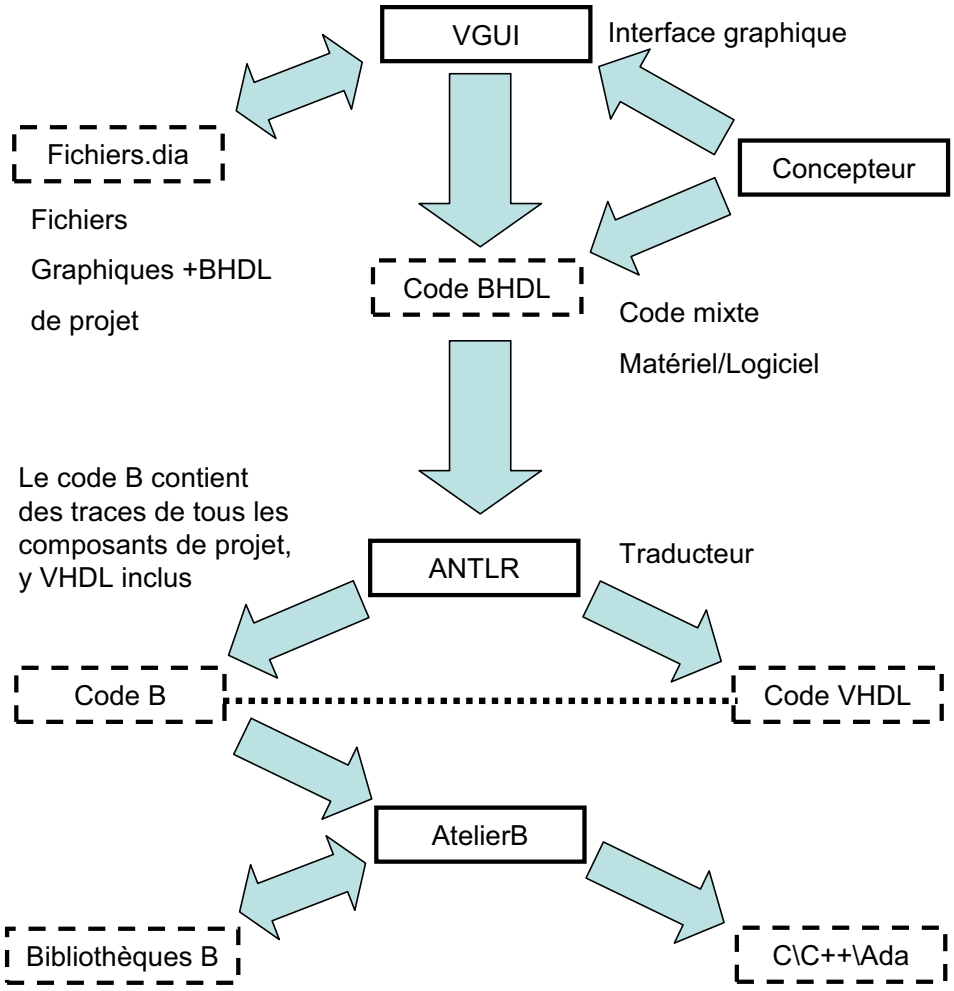

FIG. 3.2 – Schéma global de la plateforme BHDL (2).

3.2 Structure des composants VHDL

L'entité[3]

L'entité contient trois parties :

- La partie déclarative qui commence par le mot clé *ENTITY*, suivi par le nom de l'entité,
- une partie facultative qui commence par le mot clé *GENERIC*, elle permet de déclarer des paramètres et préciser, pour chaque variable, son type, et sa valeur,
- une partie pour la déclaration de l'interface, qui commence par le mot clé *PORT*. Dans cette partie tous les ports sont spécifiés. Pour chaque port, on déclare le nom et le type et on peut déclarer le *mode*. Ceci précise le mode l'entrées\sorties dans le port, ainsi il peut avoir les valeurs : IN, Out, InOut, Buffer.

L'architecture

L'architecture contient aussi trois parties :

- une partie de déclaration qui commence par le mot clé *ARCHITECTURE*. Dans cette partie, on donne un nom à l'architecture et on l'associe à la spécification d'une entité.
- Une partie déclarative. Elle peut contenir la déclarations des variables, signaux .etc. Même elle peut contenir des déclarations de sous-composants qui seront utilisés pour définir l'architecture.
- Le corps : ensemble d'instructions séquentielles et concurrentes. l'instruction la plus importante qu'on utilise dans ce projet est l'appel instanciée d'un composant ou d'une entité.

3.3 V-GUI

VGUI[4] est une interface graphique pour les utilisateurs des modèles de *VHDL*. Il a été créé en décembre 1998. Après sa création et en coopération

[3] Voir page 9.

[4] Graphical User Interface.

avec son concepteur, il a été modifié à deux reprises en 2003.

Actuellement, ***VGUI-2*** est un outil capable de **créer, éditer et visualiser des modèles, des architectures et des structures** de composants électroniques. Il est aussi approprié pour une entrée de conception générique. Il permet d'exprimer des **propriétés logiques** et quantitatives qui sont assez riches pour formaliser les propriétés du système. Il peut générer des boîtes modulaires et il peut aussi générer des copies WYSIWYG[5] de conceptions (c'est un type de fichiers qui sert à imprimer les dessins comme ils apparaissent sur l'écran). Il peut aussi produire des copies structurelles de conception en Verilog et *VHDL*. Prochainement, il devrait aussi capable de générer JHDL[6] [17].

Les fichiers de conceptions en *VGUI* sont codés en texte *ASCII*. Ainsi les mêmes fichiers peuvent être ouverts, sauvés, et déplacés à travers des environnements différents, ils peuvent aussi être édités par toutes les versions de *VGUI*. Ceci facilite la production et la gestion de bibliothèques hétérogènes

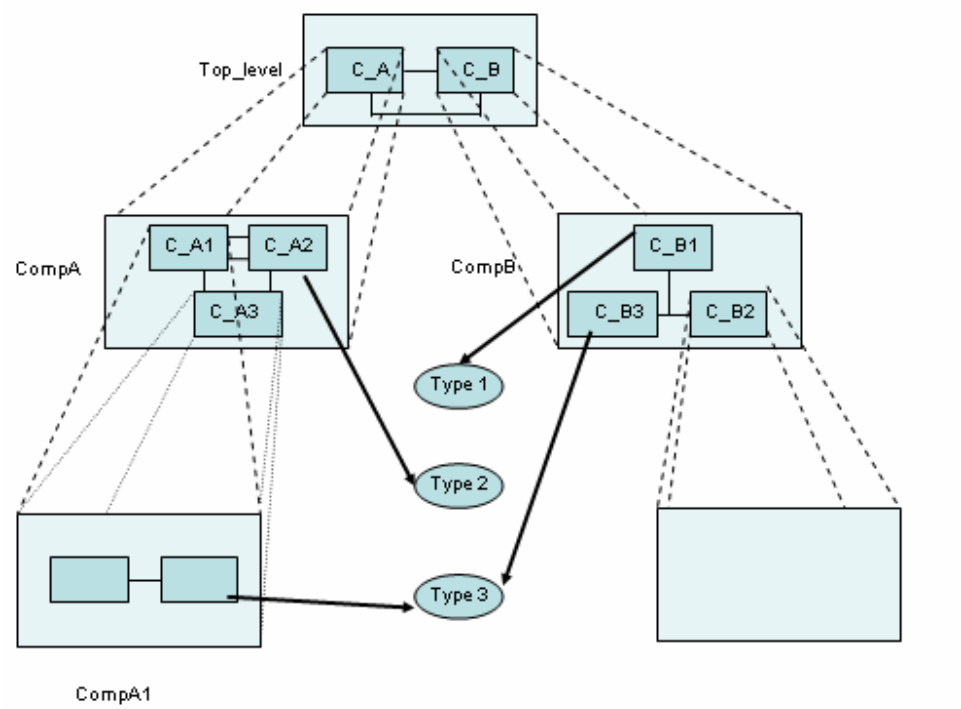

FIG. 3.3 – Conception hierarchique et modulaire en *VGUI*.

[5]What You See Is What You Get.
[6]Java HDL : www.jhdl.org

VGUI permet de déclarer des propriétés compositionnelles, comportementales et hiérarchiques.

D'un côté, La conception en *VGUI* est composée d'une ou plusieurs boîtes dont, au minimum, les interfaces sont définies. On peut ouvrir chaque boîte pour voir, déclarer ou modifier sa structure interne car il peut être composé, à son tour, de plusieurs boîtes. La structure déclarée d'une boîte peut être utilisé comme type générique qui sert à typer des boîtes similaires dans la même conception ou à construire des bibliothèques pour les futures conceptions.

De l'autre coté, *VGUI* permet de spécifier des propriétés comportementales comme le but de circuit « *purpose* », « *cost* », « *weight* », etc. Meme si *VGUI* n'utilise pas les informations qui sont enregistrées comme propriétés, mais il peut être exploiter vers d'autres outils d'analyse. Un autre point important est que les connecteurs en *VGUI* sont typables au *VHDL* et ce type est traduit dans le code *VHDL* généré, ce qui est particulièrement utile pour le debugging dans la phase de conception d'entrée.

Les éléments principaux utilisés dans la conception de *VGUI* sont les suivants :

- **Device/Node**
  il est utilisé pour déclarer les feuilles de la structure hiérarchique de conception.

- **Module/Supmode**
  cet élément sert pour spécifier un composant « non feuille » de conception cadc'est à dire, qui pourra être composé d'autres composants feuille. Le module racine a pour nom par défaut « *TOP_LEVEL* ». Il est créé d'office à la création d'un nouveau fichier *VGUI*.

- **Link/Arc**
  il représente la liaison entre deux composants (*Node*, *Module*, *external port*, etc.).

- **External Port**
  il est employé dans la structure interne d'un module pour indiquer les points de connexion avec la structure externe où le module est utilisé comme composant.

Par défaut, *VGUI* sauve des diagrammes dans un langage structuré d'objets géométriques (boites, liens, commentaires hiérarchiques, etc.), ceci pour préserver la position et la taille de chaque objet. Ce fichier peut être écrit facilement pour générer un dessin ou il peut être traité par des autres outils pour traduire la conception en autre langage.

VGUI ne considère pas le dessin comme un ensemble de pixels, mais de façon vectorielle et traite chaque élément de conception comme un objet indépendant, ce qui donne, comme d'autres outils graphiques à objet, beaucoup de souplesse pour manipulation :

- On peut, par exemple, traiter un élément individuellement même s'il était dans une structure complexe où la distinction d'éléments est difficile,
- on peut aussi déplacer un module ou une partie de dessin sans couper les liaisons logiques qui existent avec les autres parties.
- Plusieurs modes de vues peuvent être utilisés pour mieux comprendre et développer la conception. En contrôlant quelques paramètres individuellement, on peut montrer ou cacher certaines informations sur le diagramme. Cet aspect graphique aide à mieux voir la conception parce que le diagramme complet peut être trop complexe pour permettre un affichage complet. Les attributs qui peuvent être cachés ou montrés sont :
  - les noms de noeuds,
  - les types de noeuds,
  - les noms des ports,
  - les attributs de connexion
  - la grille.

VGUI est un outil graphique essentiellement basé sur la souris. Même si l'utilisateur n'est pas, normalement, concerné par les valeurs des coordonnées précises géométriques de ses éléments de conception, ces informations sont affichées dans la fenêtre de commande et peuvent servir de guide ou de la contrôle. Toutes les actions et événements sont d'ailleurs ainsi tracés ce qui est particulièrement utile en cas de bug dans la conception ou dans l'application elle-même car *VGUI* est encore un prototype dont la stabilité n'est pas totalement garantie.

Afin de protéger la conception, *VGUI* sauvegarde automatiquement une copie de fichier qui contient la description de diagramme qui représente la conception avant la dernière sauvegarde. Il sauvegarde automatiquement aussi des versions intermédiaires de schémas de conception pendant l'édition.

Les schémas intermédiaires sont des fichiers « journaux ». Ces fichiers sont automatiquement nettoyés quand on quitte *VGUI* normalement.

3.4 La Méthode B et ses outils

3.4.1 Présentation de B

B est une méthode formelle définie par Jean-Raymond Abrial [3]. Elle s'applique à la spécification, la conception et la programmation. Elle fait partie de la famille des méthodes de modélisation par machines à état. Elle est fondée sur la logique du premier ordre, la théorie des ensembles et le **principe de raffinement**.
B est supporté par deux outils clones *BToolkit* [16], qui est populaire en Angleterre, et l'*AtelierB* [40], le plus avancé qui a été développé et surtout utilisé en France. Même si ces deux outils sont nées à la même période et partagent la même philosophie, ils ont suivis des évolutions légèrement différentes. Le portage d'un programme de l'un vers l'autre nécessite un certain travail.

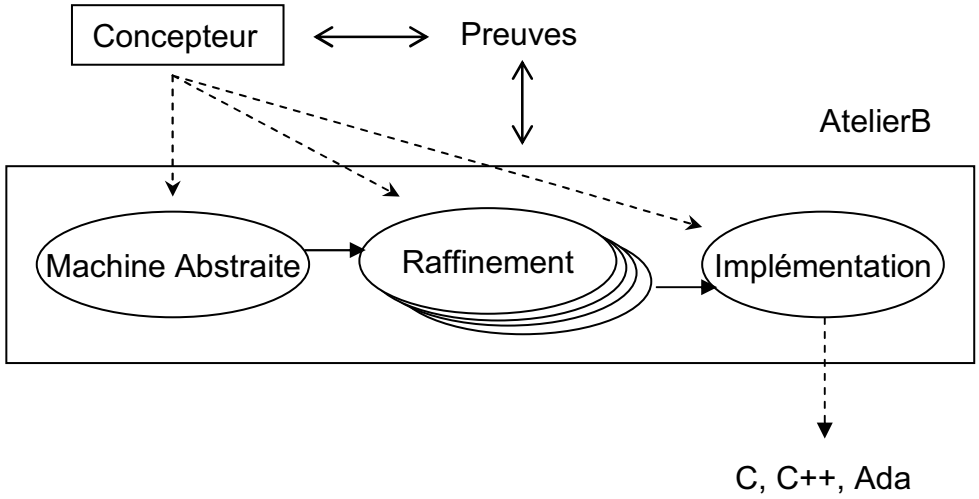

FIG. 3.4 – Un outil B.

Le développement d'un programme en *B* commence par la **modélisation mathématique de sa spécification** (voir figure 3.4). Ce modèle, que l'on appelle **machine abstraite**, décrit les données et les procédures du programme indépendamment de la manière dont ils seront réalisés. Une machine abstraite n'est pas un modèle exécutable, En général elle ne décrit ni les structures ni les algorithmes qui implanteront le programme modélisé.

Des **étapes successives de raffinement** sont nécessaires pour passer progressivement d'un modèle abstrait non exécutable à un modèle concret traduisible dans un langage de programmation exécutable. La dernière phase de raffinement d'une machine abstraite que l'on appelle l'**implémentation** peut être construit avec des machines abstraites qui modélisent d'autres composants du programme ces machines abstraites seront indépendamment raffinées à leur tour.

A chaque étape de développement d'un programme, le concepteur a des obligations de preuve. Il doit prouver que la machine abstraite initiale est mathématiquement **cohérente** et que tous les raffinements (y compris l'implémentation) satisfaisant les spécifications de ses ascendants (y compris la machine abstraite).

Le raffinement définit une **relation transitive et monotone** entre les modèles. Ainsi, par transitivité, le raffinement d'un raffinement d'un modèle est lui-même un raffinement de ce modèle. Et, par monotonie, une machine abstraite peut être remplacée par son implémentation dans un programme qui l'utilise, sans que le comportement de celui-ci soit altéré.

Donc si le traducteur des implémentations préserve la sémantique des implémentations et le code exécutable, obtenue par l'assemblage des traductions des implantations, réalise effectivement la spécification initiale.

La méthode *B* utilise la **même notation dans toutes les étapes de développement**. Chaque composant en *B* est une collection de plusieurs clauses indépendantes et chaque section commence par un mot-clé de type clause.

3.4.2 Les clauses B utilisées dans notre projet

Les clauses les plus importantes sont les suivantes :

- Clause **MACHINE**
  C'est la première clause dans une machine abstraite dans cette clause on accole le nom de machine.

- Clause **REFINEMENT**
  C est la première clause qui apparaît dans un raffinement.

– Clause **REFINES**
On trouve cette clause dans le raffinement ou dans une implémentation. Elle indique le nom de machine abstraite dont le raffinement est dérivé.

– Clause **IMPLEMENTATION**
C'est une clause qui se trouve dans une implémentation pour préciser le nom de ce raffinement.

– Clause **INVARIANT**
Cette clause permet d'exprimer sous la forme d'un prédicat appelé invariant d'une part le typage des variables déclarées dans le composant et d'autre part des propriétés sur ces variables. L'invariant exprime les propriétés invariantes de la machine abstraite portant sur les variables. En effet, il faut prouver que l'initialisation de la machine établit l'invariant et que chaque appel par un composant extérieur à l'une des opérations de la machine préserve l'invariant. Les accès, de l'extérieur du module, directement aux variables de la machine, ne peuvent pas contredire l'invariant puisque en vertu du principe d'encapsulation ; Il est interdit de modifier de l'extérieur les valeurs des variables.

– Clause **DEFINTIONS**
Dans la clause *DEFINITIONS* on introduit des définitions textuelles éventuellement paramétrées, qui sont remplacées dans le texte du composant avant son analyse sémantique.
Les définitions d'un composant doivent avoir des noms deux à deux distincts. Une définition est paramétrée si son nom est suivi par une liste de paramètres formels. Ces paramètres doivent être deux à deux distincts. La partie droite d'une définition représente une expression, un prédicat ou une substitution. Les définitions peuvent dépendre d'autres définitions mais elles ne doivent pas conduire à des références cycliques.

– Clause **VARIABLES**
Elle permet d'introduire les variables d'état de machine. Ces variables doivent être typées dans la clause *INVARIANT* et initialisées dans la clause *INITIALISATION*.

– Clause **INITIALISATION**
Elle initialise toutes les variables du composant. Il faut prouver que l'initialisation établit l'invariant.

- Clause **SEES**
  Une machine M1 peut voir une instance d'une autre machine M2. Le nom de l'instance vue se construit à partir du nom de machine vue, précédé des éventuelles renommages successifs de l'instance de M2.

- Clause **INCLUDES**
  En utilisant cette clause, on peut regrouper dans un composant, les constitutions (ensembles, constantes, et variables) d'instances de machines ainsi que leurs propriétés (clause *PROPERTIES* et *INVARIANT*), afin de décomposer une spécification complexe en plusieurs composants. Cette décomposition permet de simplifier la preuve d'un composant. En effet, la preuve d'un composant et de ses machines incluses (clause INCLUDES) est globalement plus simple que la preuve du composant équivalent.

- Clause **SETS**
  Elle permet de déclarer des ensembles dans un composant. Deux sortes d'ensembles peuvent être définis en B : les ensembles énumérés et les ensembles abstraits. Un ensemble abstrait est défini par son nom. Sa structure sera définie dans un raffinement de machine actuelle. Un ensemble énuméré est défini par son nom et par la liste ordonnée et non vide de ses éléments énumérés. Les éléments d'un ensemble énuméré sont appelés des énumérés littéraux. Ils possèdent la même sémantique que des constantes concrètes dont le type est l'ensemble énuméré.

- Clause **CONSTANTES**
  Définit la liste des constantes concrètes d'un composant. Une constante concrète est une donnée implémentable dans un langage informatique dont la valeur reste constante et qui est implicitement conservée au cours du raffinement jusque l'implémentation. Une constante concrète peut être un entier concrète, un booléen, un élément d'un ensemble abstraie ou énumère, un intervalle fini et non vide d'entiers concrets, un intervalle fini et non vide d'ensemble abstrait, ou un tableau concrèt.

- Clause **PROPERTIES**
  Dans la machine abstraite, cette clause permet d'exprimer sous la forme d'un prédicat d'une part le typage des constantes déclarées dans le composant et de l'autre côté des propriétés sur ces constantes.

3.5 Les outils de La méthode B

Les deux outils principaux qui traitent le code *B* sont *BToolkit* [16] et l'*AtelierB* [40].
B-core, la communitée chargée de la méthode *B* en Angleterre, a réalisé en coopération avec et l'université d'*Oxford* l'outil *BToolkit*. C'est le fruit d'un programme de huit ans de recherche sur les méthodes formelles.
Le deuxième outil, l'*AtelierB*, est réalisé par le *Steria* en France.

Les deux outils partagent des propriétés communes (voir figure 3.4) :

- Ils peuvent analyser syntaxiquement et lexicalement le code *B*. Ceci permet de parser le code *B* et de détecter les erreurs possibles dans le code source *B*. Il donne aussi à l'utilisateur la possibilité de modifier le code *B* en appelant l'éditeur correspondant depuis l'outil,

- Pour un projet *B*, ils génèrent les obligations de preuves nécessaires pour prouver les propriétés de cohérence et raffinement. Ils permettent d'exécuter les preuves soit automatiquement soit en coopération avec l'utilisateur. Si le code source était improuvable, ils aident l'utilisateur à en déterminer la cause,

- Dans le cas de modification dans un projet, qui était déjà prouvé correct, ils sont capables de détecter l'ensemble des preuves qui sont affectées par cette modification. Ceci produit comme résultat la minimisation des preuves nécessaires pour prouver le nouveau projet et la possibilité d'utiliser un composant de bibliothèques qui sont déjà prouvés correctes sans avoir besoin de les reprouver. Ceci limite potentiellement la quantité de preuves,

- A partir d'une implémentation correcte, ils permettent de générer du code *Ada*, *C*, *C++*.

Cependant, plusieurs différences entre les deux outils ont des effets sur la forme du code *B*. Ceci n'a pas un effet énorme sur la méthodologie *B* elle-même, mais nous oblige à préciser la cible de notre travail pendant la traduction automatique. Parmi ces différences on trouve :

- Une bibliothèque standard comme *BOOL*, qui définit les booléens, est vue par défaut dans l'*AtelierB* pas en *BToolkit*. Ceci nécessite de l'ajouter dans la clause *SEES*, quand c'est nécessaire, si on utilise *BToolkit*.

– La clause *INCLUDES* dans *BToolkit* autorise l'utilisation multiple de la même machine en utilisant des copies avec des noms différents. Ceci n'est pas le cas dans *l'AtelierB*.

– BToolkit autorise l'utilisation de ses définitions de l'extérieur de composant en utilisant la clause convenable. Par contre, la clause de définition est locale dans *l'AtelierB*.

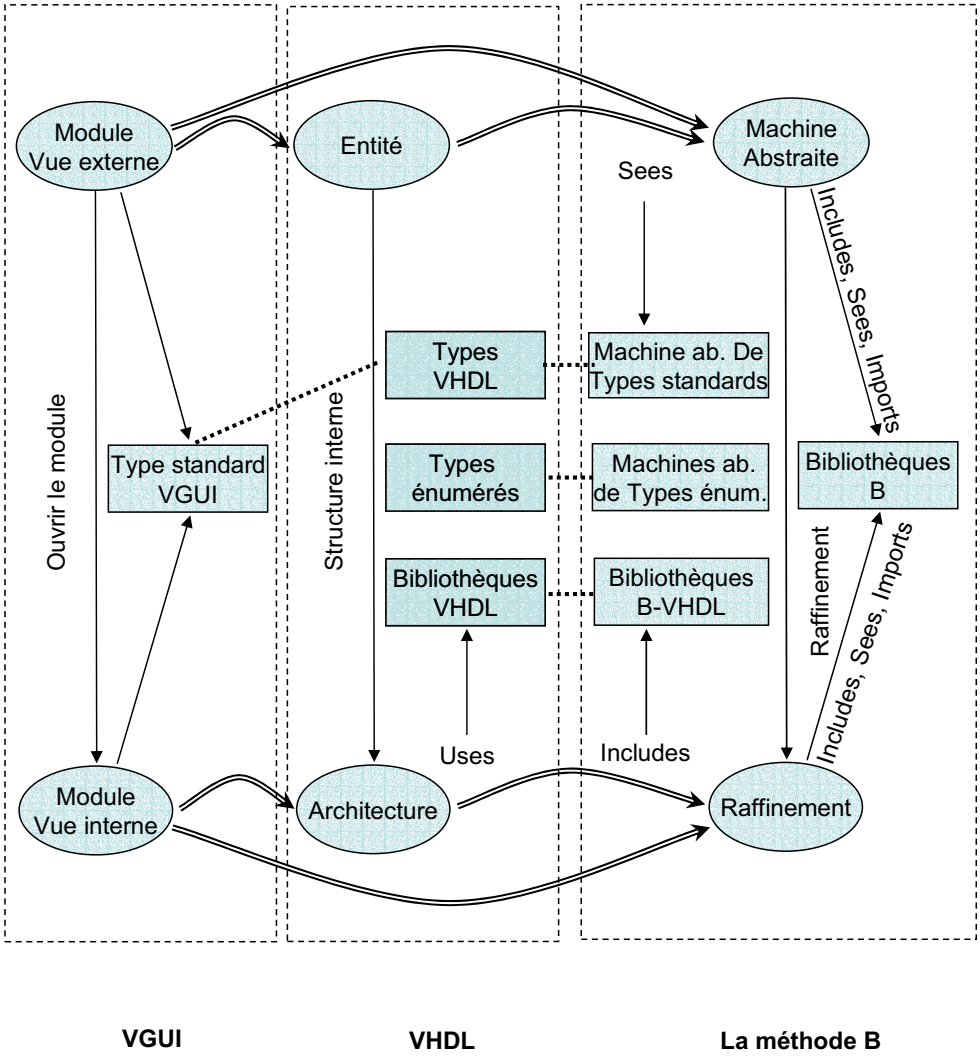

FIG. 3.5 – Schéma de traduction VGUI-VHDL-B.

3.6 BHDL. La correspondance et la traduction

Le squelette de notre application, comme le montre la figure 3.5, décrit la correspondance conceptuelle entre le développement « entité et architecture » du coté *VHDL* et « la machine abstraite » et raffinement du coté *B*.

A cela il faut ajouter l'enrichissement de l'entité avec des propriétés logiques. Ceci permet à l'entité non seulement de préciser l'interface externe de composant mais aussi de spécifier son comportement.

L'entité est la description initiale d'un composant électronique et l'architecture enrichit cette description en ajoutant la description interne soit au niveau algorithmique soit au niveau composition. La déclaration de l'architecture est l'étape qui suit celle de l'entité. La conception des composants doit satisfaire les spécifications initiales, être monotone et consistante. De plus, chaque architecture est attachée à une, et une seule entité. On considère l'architecture comme étant un raffinement de l'entité. Cependant c'est un cas spécial de raffinement car il y a un certain type d'indépendance entre les deux descriptions.

L'addition des propriétés fonctionnelles logiques à l'entité de *VHDL* nous permet de :

- déclarer des **spécifications fonctionnelles** ou des propriétés de sûreté .etc. ceci au niveau de l'entité.

- la composition de plusieurs entités, dans une architecture (de composant plus complexe) permettra de donner une **signification structurelle** à cette architecture par la description de ses composants et la méthode par laquelle ils sont connectés.

- Les descriptions logiques de deux couches « entité, architecture » permettront d'avoir une base formelle pour prouver **la relation de raffinement** entre la spécification fonctionnelle de l'entité et la signification structurelle de l'architecture.

Ce nouveau code (*VHDL* + propriétés logiques) est appelé un code *BHDL*. Une description précise de la syntaxe de ces annotations est décrite à la page 102

Les composants principaux du projet *VGUI* est utilisé comme interface graphique de conception et il est capable par défaut de produire une description *VHDL*. *VGUI* a été modifié pour produire le nouveau code *BHDL*. Les deux vues, externe et interne, de module en *VGUI* vont produire du code *BHDL*, code qui est traduit à la fois en *B* et en *VHDL* ;

- l'image *VHDL* d'une *vue externe* est l'entité et celle de *B* est la machine abstraite,
- et la *vue interne* d'un module est traduite comme étant une architecture en *VHDL* et un raffinement en *B*.

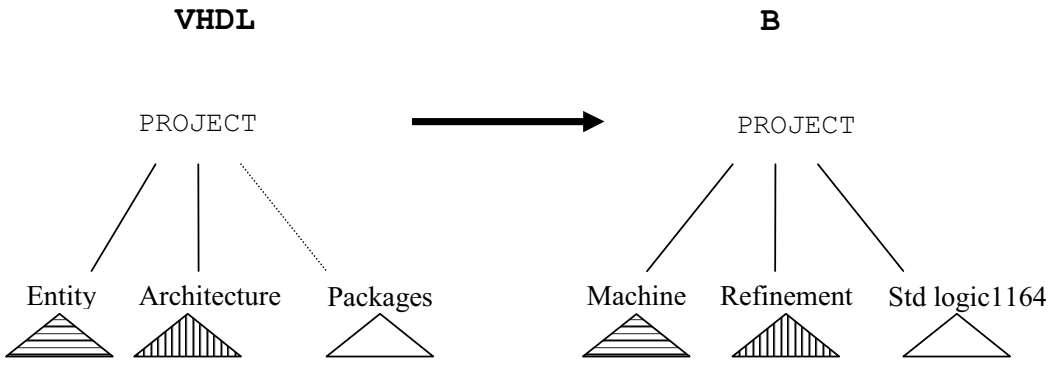

FIG. 3.6 – Correspondence entre les composants de projets BHDL et B.

Le code *B* ne contient pas seulement les propriétés non *VHDL*. Il contient aussi la partie structurelle du code *VHDL*. Alors, comme la figure 3.6le montre, un projet *BHDL*, ou *VHDL*, sera traduit par un projet B dans lequel la machine abstraite et le raffinement correspondent aux entité et architecture respectivement.

Pour analyser et prouver un projet *BHDL*, ou *VHDL* en utilisant un outil *B*, des bibliothèques *B* correspondant aux bibliothèques *VHDL* doivent être créées. Comme exemple, on a traduit manuellement la bibliothèque la plus connue *std_logique* (voir page [111]).

Selon le type de bibliothèques et selon la méthode par laquelle elle est utilisée, elle sera appelée dans les clauses *SEES, IMPORTS, INCLUDES*

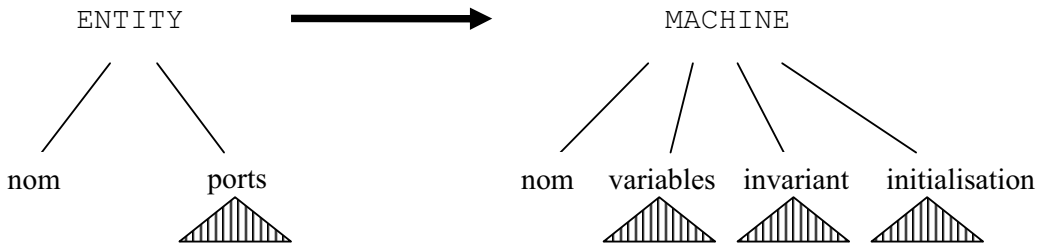

FIG. 3.7 – Correspondence entre une entité et une machine abstraite.

Entité-Machine abstraite

Une entité *VHDL* produit une machine abstraite en *B* (voir figure 3.7). Cette machine contient suffisamment d'information pour connaître l'interface que l'entité définit.

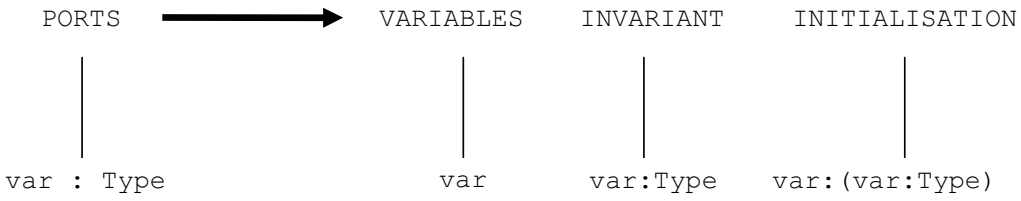

FIG. 3.8 – Un port en B.

Pour chaque port, signal ou variable, de l'interface de l'entité, une variable *B* est définie dans la machine *B* (voir figure 3.10). Cette variable est déclarée dans la clause *VARIABLES*. Elle est typée dans la clause *INVARIANT*. Comme les autres variables en *B*, ce typage doit être initialisé dans la clause *INITIALISATION*.

Les types eux mêmes sont définis de plusieurs manières :

- les types standards de *VGUI*, comme les types *BIT, INT*, etc., sont définis soit par défaut en *B* soit dans une machine spéciale, *VGUI*_Type, qui est vue automatiquement par tous les composants *B* du projet.

- Les types *VHDL* qui ne sont pas connus par défaut en *VGUI*, comme les types énumérés, sont définis dans des machines indépendantes qui sont aussi vues par toutes les machines de projets qui correspondent aux composants *VHDL* où ils sont déclarés. Les noms de ces types doivent

être modifiés parce qu'en *B* ils ne sont pas définis localement, comme en *VHDL*.
Ce type de définition peut créer une ambiguïté : deux types énumérés peuvent partager le même nom et être défini de manière différente localement. On peut résoudre le problème en changeant le nom de type par exemple en y ajoutant le nom de composant *VHDL* où il est défini.

– Les types qui sont définis dans des bibliothèques *VHDL* indépendants, comme le type *STD_ULOGIC* qui est défini dans la bibliothèque std_logic, sont automatiquement vus quand les composants *B* voient ou incluent ou importent les bibliothèques correspondantes en *B*.

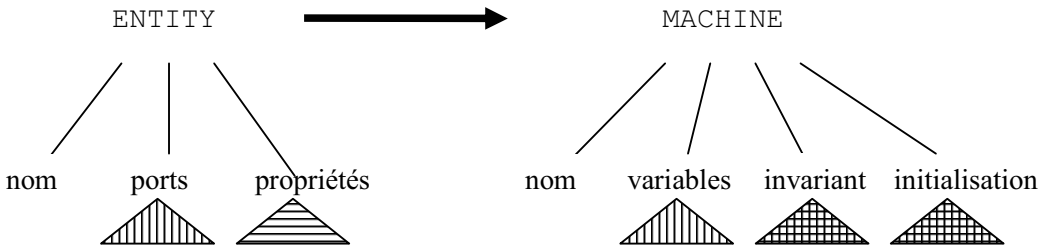

FIG. 3.9 – Une entité en BHDL et la machine correspondante en B.

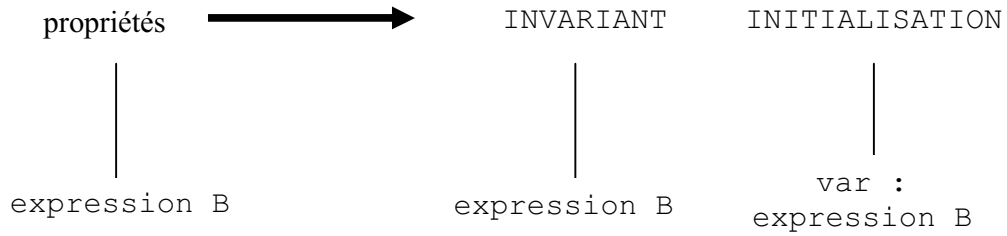

FIG. 3.10 – Transmission de propriété de BHDL en B.

Propriété-Invariant

Le code logique de propriété ne fait pas partie de *VHDL* et est transmis directement en B. Les propriétés fonctionnelles d'une entité sont attachées à l'invariant dans la machine abstraite correspondante et toutes les variables qui ne sont pas définies en *VHDL* doivent être ajoutées dans la clause VA-

RIABLES ce que nécessite leur typage dans la clause invariant (Voir figures 3.9 et 3.10).

Pour des raisons de simplification dans la phase d'implémentation de *BHDL* Tool, nous avons ajouté un codage « standardisé » de l'invariant d'une entité sous la forme d'une définition. Cette opération pourrait être modélisée par la règle de B vers B suivante ; la figure 3.11, pour une entité A.

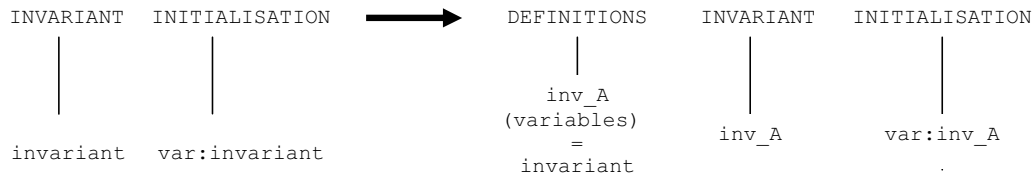

FIG. 3.11 – Définition de l'INVARIANT comme étant une « macro »dans la clause DEFINITION.

Ces définitions (macros) sont recopiées dans la clause *DEFINITIONS* ou dans un fichier indépendant selon la destination de traduction l'*AtelierB* ou *BToolkit*.

Dans la clause INITIALISATION qui est une clause obligatoire dans les composants *B*, où on doit initialiser les variables pour satisfaire l'ivariant.

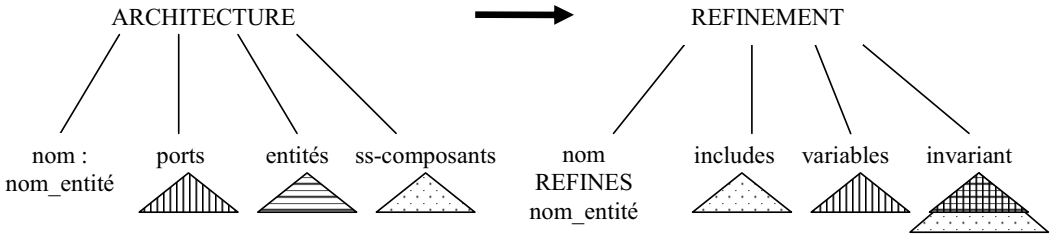

FIG. 3.12 – Correspondence entre une architecture et un raffinement.

Architecture-Raffinement

L'architecture est représentée par un raffinement (voir figure 3.12). Ce raffinement aura le nom de l'architecture[7]. Afin de déclarer la relation de raffinement avec l'entité correspondante, le nom de ce dernière est indiqué dans la clause *REFINES*.

[7]Dans notre exemple, pour des raisons de clarté, le nom de l'architecture est une composition de nom de l'architecture et l'entité correspondante.

Les connexions entre les sous-composants dans l'architecture sont représentées comme les ports dans l'entité, qui sont des connexions externes (voir figure 3.10).

Notons qu'en VHDL il est possible de définir des composants qui ressemblent aux entités dans l'ARCHITECTURE. Dans ce cas, il s'agit de COMPONENT et non pas d'ENTITY dans la terminologie VHDL. Il n'en demeure pas moins que ces deux structures syntaxiques ont exactement la même fonction de notre point de vue. En d'autres termes, les COMPONENTs seront notés et traités dans la suite comme des ENTITYs. On pense que cette définition est bien adaptée à notre méthode, elle n'impose pas de conditions particulières sur l'implémentation et simplifie les obligations de preuve à traiter.

La déclaration d'un sous-composant dans l'architecture en *VHDL* correspond à la création d'une machine abstraite en *B*.

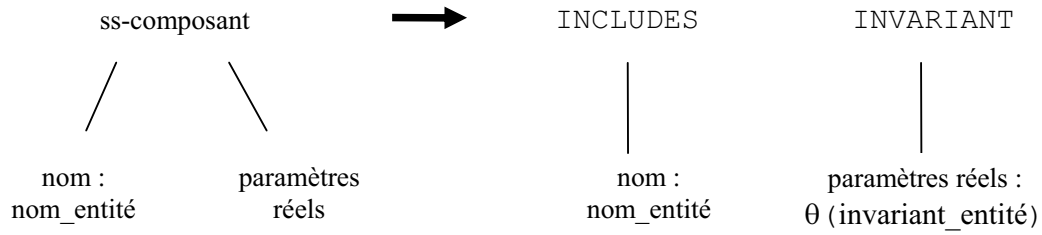

FIG. 3.13 – Instantiation d'un sous-composant.

Et l'instantiation de ce sous composant dans un architecture (voir figure 3.13) est représentée par l'inclusion de la machine abstraite correspondante dans le raffinement. L'invariant de raffinement relie les invariants de différents sous composants. Encore une fois la définition de l'invariant de chaque sous-composant sous la forme d'une « macro » aide à standardiser la compilation. Puisque l'architecture est un raffinement de son entité, son invariant (obtenu par connection de sous-composants) doit satisfaire l'invariant de l'entité (qui représente les propriétés fonctionnelles souhaitées par le client).

Cette méthode permet de profiter de composants qui sont déjà prouvés comme étant sûrs pour construire un nouveau composant sûr. La vérification de conception de plusieurs niveaux de composition est prouvée séparément. Chaque composant est prouvé correcte en se basant sur sa structure interne et sur les spécifications de ses sous-composants sans connaître leurs structures. Ceci permet une conception « top module » sûre et permet aussi le

remplacement d'un composant par un autre avec le minimum de preuves nécessaires sans avoir besoin, par exemple, de réprouver tout le système.

La connection entre les composants peut avoir des effets sur des autres clauses que l'INVARIANT selon le but de traduction :

- **La connexion par l'interface de l'opération**
  Dans cette méthode, chaque variable dans l'entité est attachée à une variable dans la machine abstraite qui est à son tour attachée à une ou deux opérations pour lire ou modifier sa valeur selon son mode dans l'entité :

  - Si le variable est en mode IN en *VHDL*, une opération de modification sera attachée à la variable correspondante en *B*.

  - Si le variable est en mode OUT en *VHDL*, une opération de consultation sera attachée à la variable correspondante en *B*.

  - Si le variable est en mode INOUT en *VHDL*, deux opérations seront attachées à cette variable une pour la consultation et l'autre pour la modification.

  Chaque opération de modification de variable dans le composant complexe va appeler, dans l'ordre, toutes les opérations qui sont dans les sous-composants sur le chemin de signal. Cette méthode peut être utilisée pour représenter la simulation de passage de signal de composants électroniques. Elle est utilisée avec quelques modifications pour représenter les circuits synchrones [78].

- **La connexion en respectant la synchronisation**
  Dans cette méthode, on ajoute à tous les types de variable de l'interface une valeur, *unknown*, qui sera utilisée pour savoir si une entrée d'un circuit a été modifiée ou pas. Si toutes les entrées d'un circuit sont modifiées, elle peut recalculer sa situation interne et elle peut changer les valeurs de ses sorties en réinitialisant ses entrées à *unknown*.

Pour automatiser la traduction on utilise des macros, ou *DEFINITIONS*, pour représenter l'invariant de chaque machine ou raffinement. Ceci sert :

- à mieux comprendre la liaison entre les différentes composants pendant la lecture,

– A ne pas répéter l'invariant (souvent long) plusieurs fois dans la machine dans les clauses *INVARIANT, INITIALISATION* et *OPERATIONS*,

– Il fournit un certain type de portabilité car le nom de l'invariant dans une machine abstraite inclut son nom précédé par les lettres INV. Ceci permet d'utiliser le même invariant à l'extérieur de cette machine.

La méthode par laquelle on utilise l'invariant varie selon l'outil B qui sera utilisé ; ceci change la forme du B produit en particulier selon l'outil B utilisé. Dans l'*AtelierB* on ne peut pas voir les définitions dans les machines vues ou incluses, par contre on peut voir des fichiers externes qui contient seulement des définitions. Ceci nécessite de modifier les règles de traduction.

3.7 ANTLR

3.7.1 Introduction

Créé par Terence Parr [96], ***ANTLR***[8] est un générateur de compilateur. Il intégre :

- Un analyseur lexical (lexeur),
- Un analyseur syntaxique (parseur),
- Un arbre syntaxique abstrait (AST)[9],
- Générateur de code.

A partir d'une description de grammaire enrichie par des actions en *C++* ou Java, ***ANTLR*** permet d'avoir une traduction source à source. En utilisant la stratégie de PRED-LL(k) ; ***ANTLR*** permet une pré-analyse, « *Lookahead* », à profondeur non bornée en cas d'ambiguïté dans la grammaire. ANTLR permet donc d'implanter des grammaires LL(k) mais aussi des grammaires de type LR(k) et LALR(k).

Pourquoi *ANTLR* ?

ANTLR permet de définir à la fois la partie lexeur et la partie parseur pour un langage comme VHDL. A partir d'une telle grammaire écrite en ANTLR, il est capable de créer un programme Java qui vérifie la syntaxe d'un programme VHDL, identifie les erreurs potentielles et construit un arbre abstrait syntaxique (AST). ANTLR fonctionne sur une large variété de systèmes et le compilateur produit est utilisable sur, pratiquement, toute plateforme grâce au langage Java. Cette application est très stable et a déjà été utilisée dans de nombreuses applications. Mais la caractéristique la plus importante pour nous est la possibilité offerte en ANTLR de traduire un arbre AST d'un langage comme VHDL vers un langage cible comme B. Cette traduction peut être définie de façon déclarative par des transformations élémentaires sous la forme de règles de réécriture applicables de façon itérative jusqu'à obtention d'un point fixe. La forme normale produite par ce système de réécriture est la traduction finale en langage cible. La mise au point et la preuve de correction du compilateur en sont grandement simplifiées. Rappelons très brièvement la terminologie et les concepts des grammaires. Nous montrons sur un exemple la puissance en termes de transformations de l'outil que nous avons utilisé.

[8] ***ANTLR*** est l'abréviation des mots anglais : (ANother Tool for Language Recognation).

[9] Abstract Syntax Tree.

1. **Les règles :**
   Chaque règle se compose d'un nom de règle, suivi des deux points « : », et un corps de règle, qui est suivie d'un point-virgule « ; ».
   Le corps de la règle montre les manières possibles par lesquelles un non terminal peut être formé d'autres parties du langage. Elle peut contenir des noms terminaux (c'est-à-dire tokens), des noms de non terminaux (noms de règles), et des meta-symboles qui décrivent la façon par laquelle les éléments du corps se composent pour construire la règle.
   Les Meta-symboles servent à représenter des occurrences multiples d'un sous-partie de règle :

   (a) La barre verticale « | », ou pipe, indique le début d'une alternative dans le corps de règle.
   (b) Les parenthèses « **( )** » délimite une partie du corps de la règle
   (c) Le astérisque « * », ou étoile, pour identifier zéro, une ou plusieurs fois une partie délimitée par des parenthéses. Le symbole plus « + » lui sert à identifier au moins une fois une partie de règle.
   (d) Les accolades « **{ }** » indiquent que la sous règle entourée doit être identifiée une fois, ou pas du tout.

   Une description simple pour un livre :

   ```
   livre : preface introduction (chapitre)+ (annexe)* ;
   ```

   En utilisant cette description, un livre se compose d'une `preface` facultative, suivie d'une « introduction », puis d'un ou plusieurs « chapitres » et enfin zéro « annexe » ou plus.
   La fonction de parseur « livre() » appelle seulement d'autres fonctions d'analyse, et exécute des boucles tandis que sur les tokens actuelles. Dans le premier ensemble des non terminales La structure pour le livre() ressemblera à :

```
livre()
   {
    si (token_suivant == K_PREFACE) alors preface();
    introduction();
    chapitres() ;
    tant que (token_suivant==K_CHAPITRE);
         { chapitre() ; }
    tant que (token suivant==K_ANNEXE)
```

```
        { annexe(); }
}
```

2. **ANTLR : analyse ascendante ou descendante ?**
   ANTLR transgresse cette classification ascendante LR(k) ou descendante LL(k) en proposant des prédicats « syntaxiques ». On peut écrire des règles conditionnelles, la condition correspondant à une pré-analyse à profondeur non bornée.

   ```
   regle : (lookahead-language) => corps de regle
   ```

   L'utilisation de la règle est ici conditionnée à une pré-analyse à profondeur non bornée « lookahead language ». Ceci permet d'établir que la puissance d'expression en ANTLR est strictement supérieure à la puissance des grammaires descendantes LL(k) mais aussi celles ascendantes LR(k) pour lesquelles k est fixé !

3. **Les arbres AST en ANTLR**
   Les arbres en ***ANTLR*** sont importants car leur structure est utilisée pour faciliter l'analyse de code. Le parseur analyse un texte d'entrée, c'est-à-dire une chaîne de caractères. ***ANTLR*** permet de convertir cette chaîne d'abord en une suite de tokens (mots du langage), puis en formes syntaxiques représentables sous forme d'un arbre AST dont la structure met en evidence l'enchainement des regles de grammaire utilisées pour recoonaitre le texte en entrée. La forme de cet arbre est donc directement liée avec la grammaire utilisée.

   Les arbres sont implémentés en interne comme des arbres binaires de façon classique : lien vertical du noeud père au premier noeud fils et lien horizontal du noeud à son noeud frère.

   Cette structure change un peu les algorithmes utilisés pour traiter les arbres : parcourir l'arbre, consulter, ajouter ou détruire un noeud. Pour accéder au noeud à partir de la racine, il faut d'abord passer par tous ces ascendants aux niveaux supérieurs ce qui demande à son tour de passer par tous les successeurs de chaque ascendant à son niveau. Chaque noeud a un type bien identifiable en terme de grammaire à savoir son token et sa valeur, la chaîne de caractères du texte associé. Lors de la construction de l'arbre AST, il est possible d'enrichir les informations associés à un noeud, voire au contraire à les effacer si notre effort de traduction ne portera pas sur cette partie de l'arbre AST.

Voici les operations de base sur les arbres ASTs disponibles en ***ANTLR*** pour faciliter cette tâche :

4. **Les transformations en ANTLR**

   Là, où ANTLR se montre très puissant, surtout pour notre propos, c'est dans sa capacité à transformer les arbres syntaxiques de manière déclarative. Illustrons cela sur un exemple d'expressions mathématiques construites à l'aide des entiers de l'addition et de la multiplication :

```
expr : mexpr (PLUS^ mexpr)* SEMI! ;
mexpr : atom (STAR^ atom)*
atom : INT ;
```

   Le role des annotations « ^ » et« ! » dans la grammaire ANTLR permettent de modifier l'ordre des tokens. Ainsi que le « ^ » met en racine le token et le « ! » ignore l'information associée au noeud. Ceci a dans notre cas pour effet d'obtenir (à l'ordre des paramètres près) la même expression en notation préfixée :

```
expr  :    PLUS ( expr mexpr)
       |  mexpr ;
mexpr :    STAR ( mexpr atom)
       |  atom ;
atom  :    INT ;
```

   Ces deux types d'annotations sont très puissantes pour une construction ensuite complètement automatisée de l'arbre AST.

   Généralisons un peu notre grammaire :

```
expr :       PLUS ( expr expr)
          |  STAR ( expr expr)
          |  INT  ;
```

   Il est possible d'associer une action sémantique à chaque règle ou alternative de règle de la grammaire pour opérer des transformations **locales** de notre arbre sémantique.

```
r:expr      :
              PLUS ( a:expr b:expr)  {r = a + b}
            | star ( a:expr b:expr)  {r = a * b}
            | i:INT                  {r = i};
```

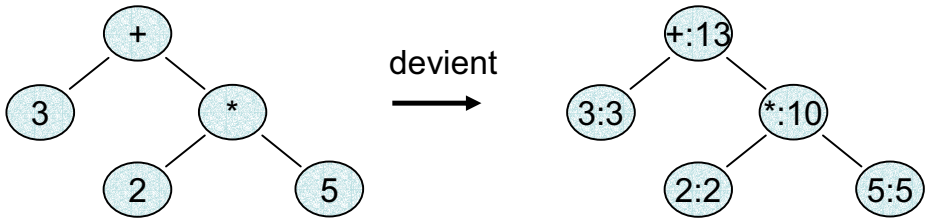

FIG. 3.14 – Analyse d'arbres AST.

Le résultat de l'analyse ANTLR ne sera plus seulement un arbre AST mais un arbre syntaxique , Voir le figure 3.14,dans lequel chaque noeud possède une annotation : la valeur de l'expression mathématique du sous-arbre de racine ce noeud. C'est dans cet esprit que nous appliquerons notre algorithme de traduction de VHDL vers B en calculant pour chaque structure **élémentaire** de notre langage VHDL-VGUI **sa** traduction en B.

3.8 Les étapes de Compilation

3.8.1 Introduction : La Grammaire utilisée

Dans ce chapitre nous allons présenter le travail que l'on a effectué : la réalisation technique de l'outil *BHDL* Tool, outil capable de prendre en compte les propriétés introduites dans le chapitre précédent. Beaucoup d'outils sont capables de traiter la conception de matériel, ou logiciel, au niveau bas. Par contre, rare sont les outils qui traitent la conception au niveau haut.

Notre premier travail a été de décrire une grammaire du langage VHDL en ANTLR. Celles que l'on trouve dans la littérature ne sont pas suffisamment complètes ou présupposent des informations sur la table des symboles. En collaboration avec P.A. Wilsey (Université de Cincinatti), nous avons tra-

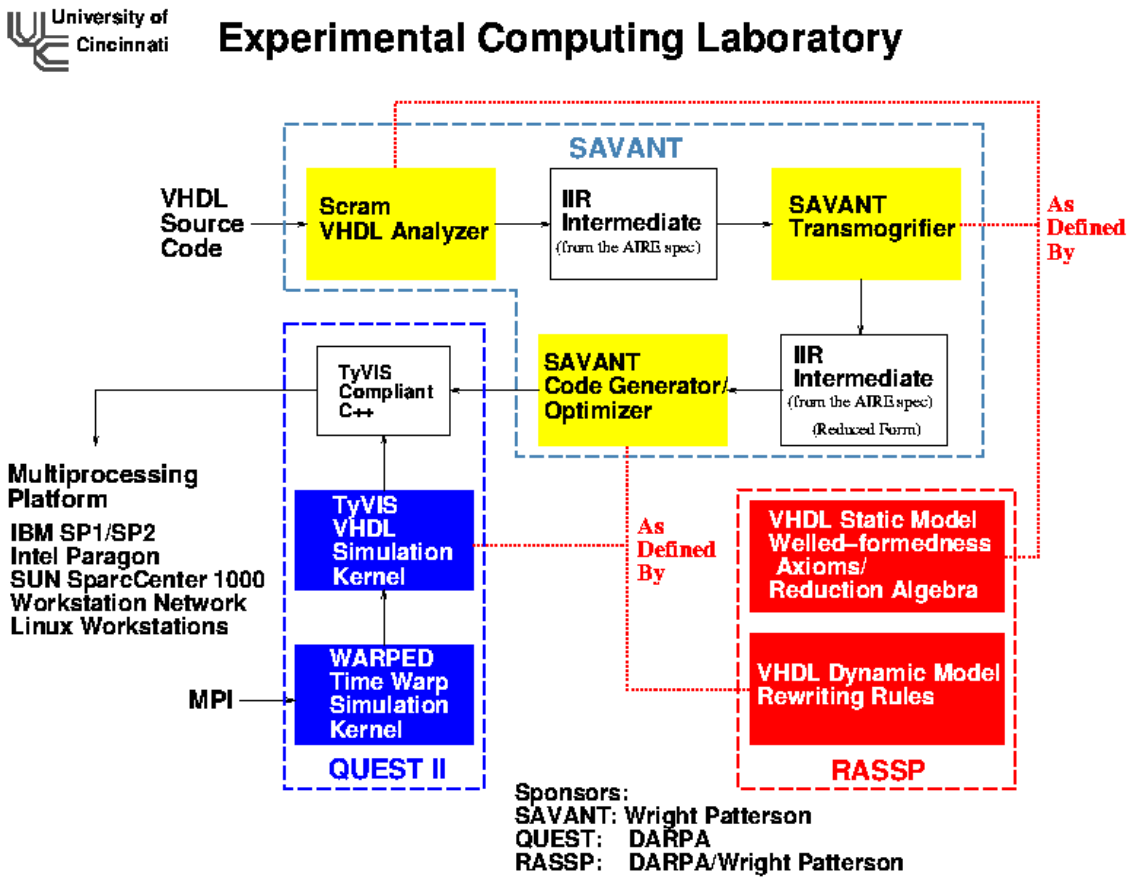

FIG. 3.15 – Projet SAVANT BHDL (1).

Le but est aussi de transformer du code VHDL en un autre langage (IIR) « interne » adapté à la simulation et à l'analyse comportementale. Même si notre propos n'est pas la simulation, nous avons pu transposer leur étude grammaticale et nous disposons maintenant d'une grammaire PRED-LL(2) complète de VHDL-AMS 2003 en ANTLR. Elle est composée de 339 règles, ou plus précisément « 339 + 452 » alternatives si on tient compte des méta-symboles « | » apparaissant en corps de règles.

La deuxième partie de notre étude a été d'extraire de cette grammaire complète, celle utilisée dans le sous-langage générée par VGUI. Contrairement à la grammaire complète, VGUI se concentre sur la description structurelle de VHDL. Ceci représente environ 1/4 des règles de la grammaire

[10]VHDL Analysis Tools.

originale. Enfin pour ces règles, il nous a fallu décrire les actions sémantiques de traduction d'un modèle vers l'autre : 86 actions sémantiques.

3.8.2 lexeur :

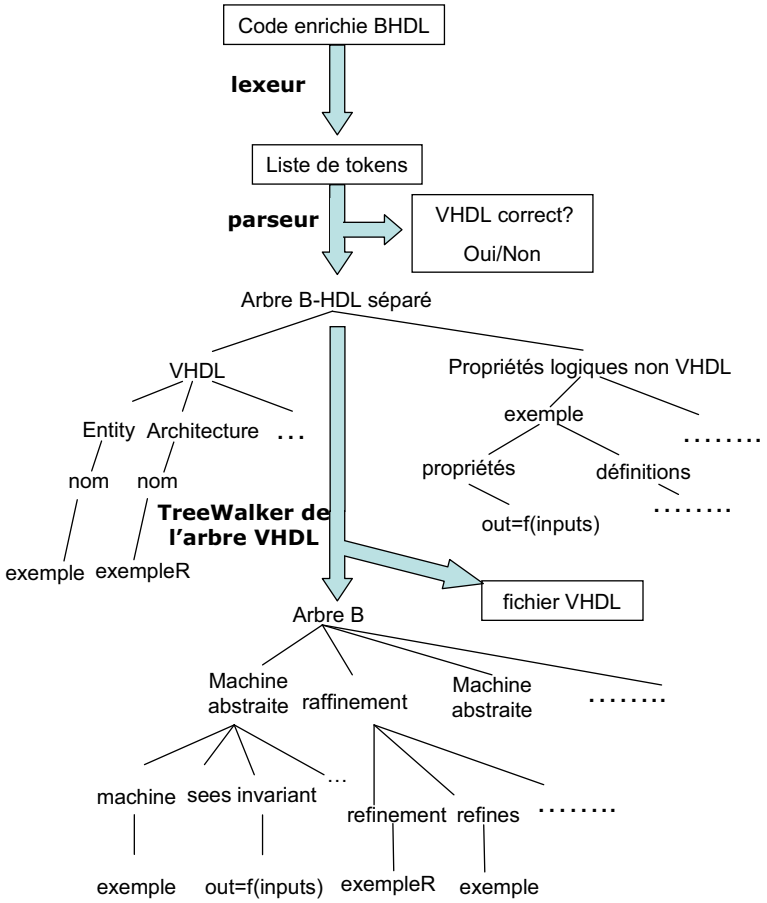

FIG. 3.16 – Analyse syntaxique BHDL.

Le lexeur utilise un lookahead de 9 caractères. Il est capable de lire et détecter sans ambiguïté tous les éléments qui peuvent exister en *VHDL* qui offrent :

- la possibilité de déclarer des valeurs entières (integer) en binaires, octaire, décimal et hexadécimal.
- Toutes les formes possibles des opérations et des opérateurs dans expressions mathématiques de *VHDL*.

- Toutes les formes possibles des opérations et des opérateurs dans expressions logiques de *VHDL*.
- les expressions spéciales pour traiter les signaux.

En plus il peut distinguer entre plusieurs types de chaînes de caractères :

- Des mots clés de *VHDL*. Voir les annexes.
- Des mots clés spéciaux qu'on a ajouté pour contrôler la structure interne des arbres de ***ANTLR***. Voir les annexes.
- Les séries des caractères qui représentent un identifier : le nom de variable, signal, constant, .etc.
- Les séries de caractères qui représentent des chaînes fixe ; les chaînes qui sont entourées par « " " ».
- Le commentaire *VHDL* et les commentaires qu'on utilise pour exprimer des propriétés « à la B » en *VGUI* mais ils n'ont pas de significations en *VHDL*. Par contre, ils sont importants pour construire les éléments des composants *B* plus tard pendant les étapes suivantes.

```
COMMENT :   "-- #i"               { $setType(BCOMMENT_INV); }
          | "-- #d"                {$setType(BCOMMENT_DEF); }
          | "-- #="! (~('\n'|'\r'))* {$setType(BCOMMENT_E); }
          | "--"     (~('\n'|'\r'))* {$setType(Token.SKIP); }
          ;
```

Traditionnellement, les commentaires en *VHDL* commence par « -- ». Quelques commentaires sont importants pour les étapes suivantes de traduction. Les commentaires traduisibles en B sont de trois sortes différentes, mais commencent tous trois par # :

1. **#i** : pour préciser les variables sur lesquelles portent l'invariant (*BCOMMENT_INV*).
2. **#d** : pour préciser la partie gauche de la définition de l'invariant (*BCOMMENT_DEF*).
3. #= : pour préciser la partie droite de la définition de l'invariant (*BCOMMENT_E*).

Tous les autres types de commentaires VHDL seront ignorés en ayant le type *SKIP*. On remarque que l'ordre de description de cette règle est important, seule le dernièr cas sera éliminé après le filtrage des autres alternatives de description.

Considérons un composant additionneur à 2 entrées entières `In1` et `In2` et une sortie entière `SUM`. On souhaite exprimer l'invariant suivant :

`In1`, `In2`, `SUM` de type entier et `SUM = In1 + In2`

Ceci se traduit sous la forme :

```
-- #i (SUM, In1, In2)
-- #d (S, A, B)
-- #= S: NAT & A:NAT & B:NAT & S =A+B
```

La composition des invariants sera beaucoup plus facile par la suite sous la forme de compositions de définitions B. Un identificateur dans une definition ne peut avoir qu'un seul caractère alors qu'au contraire une variable B doit en avoir au moins deux . Ceci explique cette presentation sous forme d'une définition avec des paramètres formels (`S,A,B`)et l'appel de l'invariant sur les paramètres réels (`SUM, In1, In2`). Ce choix a été surtout guidé par la facilite de traduction en B, sachant que la syntaxe de tels commentaires peut être affinée ou modifiée très facilement en ANTLR de manière à produire le même résultat en B.

Pour anticiper la traduction générée par *BHDL* Tool, voila ce que nous obtiendrons dans la machine B de ce composant :

```
DEFINITIONS
     inv_additionneur ==
     (S, A, B)
     = S: NAT & A:NAT & B:NAT & S =A+B
INVARIANT
      inv_additionneur
      (SUM, In1, In2)
```

3.8.3 le parseur

Le parseur, qui s'appelle VHDLparser, analyse la chaîne de tokens que le lexeur génére et il vérifie la syntaxe de ces tokens et il génére à son tour un arbre *VHDL* qui représente le code saisi. On essaie de consacrer cette étape pour la vérification sémantique *VHDL*. Ainsi, on change la forme de code, d'une liste vers un arbre, sans changer l'information incluse. La traduction finale est appliquée le plus tardivement possible pendant les étapes suivantes. Cette séparation rend ce parseur plus générique. Quand *VGUI* génére le code *VHDL*, il utilise une version simplifiée de grammaire *VHDL*. Ce code n'est

pas toujours propre ; il peut contenir des informations qui sont syntaxiquement correctes mais sans valeur sémantique, comme par exemple des entités ou des architectures vides. En plus, il peut générer des autres morceaux de code qui sont considérés comme étant des commentaires en *VHDL* mais importants pour la génération de *B*. En profitant de propriétés de ***ANTLR***, on a réussi à développer un parseur qui utilise un k=2 seulement pour :

- Analyser le code *VHDL* généré par *VGUI* contenant les unités compositionnelles de l'entité et l'architecture.

- Analyser un code *VHDL* complet qui aurait produit non pas par VGUI, mais par programmation ; en VHDL lui-même. Ce code est beaucoup plus riche que celui produit par *VGUI*, comme par exemple la déclaration de nouveaux types énumérés, déclaration de packages, de configuration, etc.

```
library_unit :
               primary_unit
              | secondary_unit
 ;

primary_unit :
 entity_declaration
             | configuration_declaration
             | package_declaration
 ;

secondary_unit :
                   architecture_body
                 | package_body
 ;
```

- Générer un arbre qui contient une structure claire de notre projet :
  Les branches essentielles de code *VHDL* et *B* sont indépendantes. Toute l'information du code source est incluse dans l'arbre généré sauf les commentaires inutiles. La structure d'un arbre est beaucoup plus riche de structure de liste de tokens ; la forme d'entrée du parseur, mais surtout les grammaires (Treewalker et Bgenerator) qui transformeront cet arbre doivent être de type LL(1). Pour cette raison, la grammaire VHDLParser doit produire dans l'arbre AST des annotations abstraites spécifiques capables de lever toutes les ambiguïtés tolérées

par un lookahead égal à 2 et les prédicats syntaxiques. La correction des actions sémantiques associées à une grammaire LL(1) et par voie de conséquence la validité de BHDL Tool seront plus facile à établir.
}

```
design_file! :
           { // This branch create the root of VHDL comments
             // which are useful in B.
             #comments= #([IDENTIFIER,"comments"]);
           }

       df:design_file1

            { #comments.setNextSibling(#df);
              #design_file=#comments;
            }
 ;
```

– Analyser le code supplémentaire fourni par *VGUI* pour spécifier des propriétés non *VHDL*. Le parseur reçoit ces propriétés encapsulées dans les commentaires que le lexeur envoie. Il réanalyse ces commentaires afin d'extraire les informations utiles. Il regroupe les informations séparées sur plusieurs liens, il les réorganise dans l'arbre *VHDL* généré. La racine principale de l'arbre contient le code *VHDL* pur et les informations supplémentaires dans une autre branche.

```
bcomment_e! :
             {  commentcollect="";
             }

 ( bd:BCOMMENT_E
   {commentcollect=commentcollect.concat(bd.getText()+"\\r\\n");}

 )+

             {  #bcomment_e= #([BCOMMENT_E,commentcollect]);
             }
 ;
```

– Supprimer une partie de code de *VGUI* qui est considéré inutile, comme les entités et les architectures vides qui sont corrects syntaxiquement mais incorrecte sémantiquement. Ce point est liée à une particularité

de VGUI, celle de conserver toute ENTITY même non utilisée, voire supprimée, et ceci à la fois dans le fichier représentation interne (`.dia`) mais aussi dans le fichier `VHDL` produit.

```
entity_declaration :
// to ignore empty entities which are generated by VGUI

      !(ENTITY (identifier) IS END)
            => ENTITY (identifier|TOPLEVEL)
               IS END (identifier|TOPLEVEL) SEMI_COLON

    | (ENTITY identifier IS entity_header)
            => ENTITY ^ identifier IS e:entity_header
            entity_declarative_part
           ( entity_statement_part )? END ( ENTITY )?
           ( simple_name )? SEMI_COLON
```

3.8.4 le TreeWalker

Le treewalker parcourt l'arbre généré par le parseur, il essaie d'analyser la sémantique pour reproduire les informations incluses ou une partie de ces informations en utilisant un autre forme. Dans notre application on utilise le treewalker pour :

- Supprimer les parties de *VHDL* qui sont utilisés seulement pour la clarté de texte mais ils n'ont pas d'importance sémantique ou syntaxique. Par exemple, l'entête d'une entité doit inclure son nom, ce nom peut être répété encore une fois à la fin de l'entité pour améliorer l'apparence. Grâce à la structure de l'arbre, où chaque entité occupe une branche indépendante, cette répétition n'est plus nécessaire. Il y a un autre type de symboles qui ont une importance structurelle pour préciser les différentes parties de code dans la forme chaîne de caractères, par exemple les parenthèses, qui encadrent une liste. Ce sont des transformations très locales dont le but est de supprimer de la redondance dans l'arbre AST.

- Il régénère un fichier *VHDL* source de code traduit afin de voir le code *VHDL* structurelle, c'est-à-dire la partie VHDL qui sera réellement traduite en B.

- Il remodifie l'arbre *VHDL* pour faciliter la lecture et la traduction de cet arbre. Par exemple, une liste de variables est représentée dans l'arbre

de parseur comme une branche à plusieurs niveaux, le nombre de ces niveaux correspond au nombre de variables. Dans la nouvelle forme toutes ces variables seront au même niveau.

```
identifier_list !:
    #(   id:IDENT11
         i1:identifier
                  {amfile.print(i1.getText());}
                  {#identifier_list=#(id, #i1);}
         ( co:COMMA {amfile.print(co.getText()+" ");}
           i2:identifier {amfile.print(i2.getText());}
                    {#identifier_list.addChild(#i2);}
         )*
     )
;
```

– Modifier l'ordre dans lequel certaines expressions en *VHDL* sont écrites afin de faciliter l'analyse sémantique et la production de code cible. Par exemple, en *VHDL*, le typage d'une liste de variables de l'interface par une seule déclaration peut être déclarer par la notation :

```
                        x,y : type.
```

Pour faciliter la traduction on a transfomé l'expression pour que chaque variable soit typée individuellement.

```
                    x : type , y : type.
interface_declaration !:
        ( (     CONSTANT!
              | si:SIGNAL {amfile.println(si.getText());}
              | VARIABLE!
          )?
          li:identifier_list
          co:COLON {amfile.print(co.getText()+ " ");}
         ( mo:mode {amfile.print(mo.getText()+ " ");} )?
         id:subtype_indication {amfile.print(id.getText()+ " "
                                             );
                               }
         ( BUS! )?
         ( initialization! )?
         {
          AST t;
          AST tempA;
```

```
                AST tNext;
                AST currentIST;
                #t=#li.getFirstChild();
                #tNext=#t.getNextSibling();
                #t.setNextSibling(null);
                if (mo!= null)
                 {
                  #tempA=#(t,[IDENTIFIER,id.getText()],
                           [mo.getType(),mo.getText()]);

                 } else
                 { #tempA=#(t,[IDENTIFIER,id.getText()]);
                 }
                #interface_declaration=#( [ IDENTIFIER ,
                                            "PortList"
                                          ],
                                          tempA
                                        );
                while (#tNext != null)
                {
                  #t=#tNext;
                  #tNext=#t.getNextSibling();
                  #t.setNextSibling(null);
                  if (mo!= null)
                   {#tempA=#(t,[IDENTIFIER,id.getText()],
                           [mo.getType(),mo.getText()]);
                    } else
                    {#tempA=#(t,[IDENTIFIER,id.getText()]);
                    }
                  #interface_declaration.addChild(#tempA);
                 }
                }
              | FILE identifier_list COLON subtype_indication
              | ( TERMINAL identifier_list COLON subnature_indication
                  | QUANTITY identifier_list COLON ( IN | OUT )?
                    subtype_indication ( initialization )?
                )
          )
    ;
```

– Générer un arbre B :

Cet arbre contient les racines principales d'un projet *B*, les machines abstraites, et les raffinements. Chaque clause de ces éléments de projet peut contenir différentes clauses qui sont représentées par des branches indépendantes. Les clauses principales générées sont *MACHINE, REFINEMENT, SEES, INCLUDES, INVARIANT, VARIABLES, OPERATIONS*.

– Parcourir l'arbre pour trouver la dépendance entre les entités, les architectures et les codes supplémentaires afin de créer les composants *B*.

3.8.5 Le B_générateur

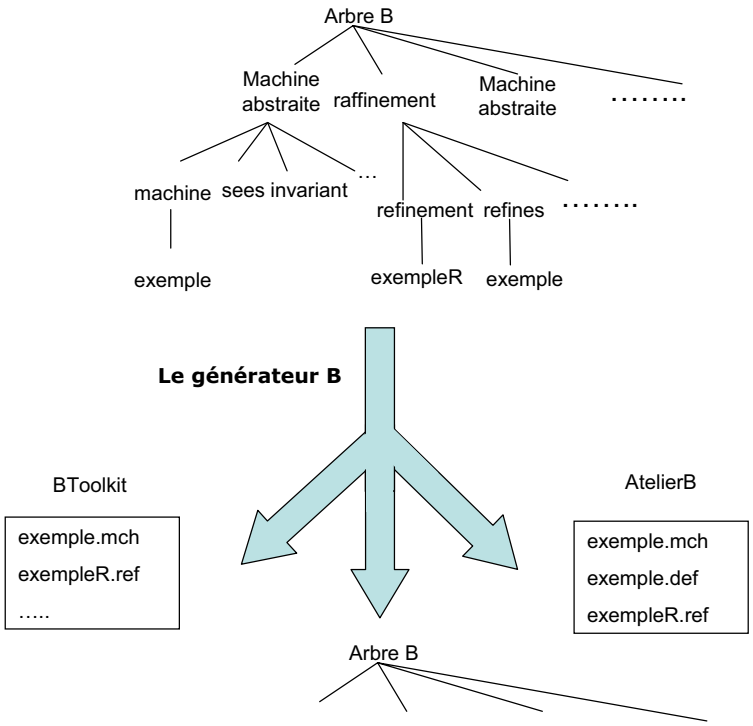

FIG. 3.17 – Sémantique et générateur de code B.

C'est un autre treewalker basé sur une grammmaire LL(1). Comme entrée, il lit l'arbre généré par le treewalker précédent, dont la structure principale correspond déjà au code *B*. il analyse cet arbre pour produire, comme sortie, le code *B*. la séparation entre la génération de l'arbre *B* et le code *B* sert à mieux comprendre l'étape de traduction. Cela sert aussi à garder le même

arbre B pour plusieurs générateurs de B et à valider plus facilement chaque étape de traduction (action sémantique). Dans notre application on a utilisé deux versions de B à cause de la légère différence entre le langage de *BToolkit* et celui de l'*AtelierB*. Le développement de notre outil pourrait produire d'autres versions, comme B événementiel. Ce générateur exécute les tâches suivantes.

1. Créer premières fichiers principaux du projet B en utilisant les actions de traitement de fichiers en *Java*. Chaque machine abstraite et chaque raffinement a un fichier indépendant. En plus, selon la cible de traduction, on peut trouver des fichiers de définitions pour chaque machine abstraite.

   Selon la structure de l'arbre il produit les clauses nécessaires dans les fichiers correspondants.

   Pour chaque raffinement il cherche la machine abstraite correspondante pour formuler correctement les clauses selon les règles de traduction.

   On peut éditer les commentaires dans chaque fichier B pour faciliter sa compréhension, ainsi pour faciliter la tâche de utilisateur de B spécialement dans le cas de preuves interactives.

2. Reformuler les sous branches qui sont encore en *VHDL* pour l'intégrer en code B. par exemple le typage de *VHDL* est reformulé sous forme d'un invariant B en sachant qu'en VHDL on peut typer plusieurs fois la même variable.

3. Tenir compte de la correspondance entre les variables formelles, qui sont utilisées pour définir une fonction, et les paramètres réels, sur lesquelles la fonction est appliquée.

4. Produire un autre arbre B comme résultat. Cet arbre sera une copie identique de l'arbre de l'entrée. Il est utile pour vérifier que la traduction est correcte. Dans le cas d'erreur, il sert à savoir de quelle branche elle vient.

3.8.6 Les bibliothèques BHDL

II- IEEE_STD_LOGIC_1164

Comme il a été présenté au première chapitre, le *STD_LOGIC_1164* définit un paquetage contenant des définitions pour un type de données standard de neuf valeurs (Voir les annexes). Ce type est nommé *STD_ULOGIC* Les neuf valeurs incluent toutes les possibilités que les signaux numériques peuvent avoir. Il définit aussi des sous_types dérivés de ces valeurs accompagnés d'une fonction de résolution. Cette fonction calcule la valeur qu'un signal après un « point de connection » peut avoir quand plusieurs signaux de types, ou de valeurs, différents sont y connectés. Cette fonction est très utile spécialement pour savoir les valeurs des signaux des interfaces entre des circuits hétérogènes.

De ces signaux, STD_LOGIC donne la possibilité d'utiliser des signaux simples ou des structures de données plus compliqués, les matrices.

Puisque, dans cette logique, on utilise neuf valeurs et non pas les deux valeurs de la logique classique, on redéfinit toutes les fonctions logiques ; `not, and, or, xor, nand, nor` sur ces 9 valeurs :

- `U` : Non initialisé
- `X` : Inconnu fort
- `0` : 0 fort
- `1` : 1 fort
- `Z` : haute impédance
- `W` : Inconnu faible
- `L` : 0 faible
- `H` : 1 faible
- `-` : sans importance

On peut distinguer deux groupes différents dans ces valeurs :

- `0, L, 1, H` détermine une valeur connue pour le signal attaché, vraie ou faux, même si cette valeur logique peut avoir une valeur physique qui dépend de la puissance du signal et du système de correspondance « logique/électronique » utilisé.

- Par contre, les valeurs « `U, X, Z, W, -` » sont considérées inconnues. Plusieurs fonctions en STD_LOGIC sont écrites pour détecter les signaux de ce type pour mieux savoir le comportement de système.

A part les fonctions logiques, ***STD_LOGIC*** contient deux fonctions pour détecter la direction de changement de signal ; s'il est descendant, quand sa valeur change de niveau haut au niveau bas, ou s'il est ascendant, quand sa valeur change de bas à haut. Beaucoup de circuits dépendent de changement descendant ou ascendant de signaux de contrôles pour affecter le lecteur, l'écriture. Aussi pour résoudre le problème de l'interfaçage entre plusieurs circuits différents, le paquetage ***STD_LOGIC*** utilise plusieurs types de conversions entre tous les types dérivés de ***STD_ULOGIC*** soit pour des signaux indépendants soit pour des matrices.

Comment il est traduit ?

Tous les éléments de ce package sont traduits en *B*. Principalement, on a deux machines abstraites en *B* qui vont contenir :

1. le type STD_LOGIC et ses dérivés,
2. les matrices construites de ces types,
3. la fonction de résolution,
4. les fonctions logiques,
5. les fonctions de conversion
6. les fonctions de détection de direction de signal
7. les fonctions de détection de signaux inconnus
8. et les attributs attachés aux objets en *VHDL*, comme par exemple l'attribut longueur qui peut être attaché à une matrice.

La première machine *B*, B_STD_LOGIC_1164_0, contient toutes les définitions et les fonctions liées avec les variables simples de types STD_LOGIC et leur dérivées.

L'autre machine, B_STD_LOGIC_1164_V_0, se base sur la première pour traiter des vecteurs de types précédents. On utilise aussi une machine supplémentaire, B_Signal_0, pour modéliser un signal.

1. **Modèle de traitement d'une valeur *Ulogic***

   Les types sont représentés en *B* en utilisant des ensembles. Le type principal *STD_ULOGIC* est défini par un ensemble qui porte le même nom. Ainsi les types dérivés sont représentés comme étant des « sous-ensembles » de celui de *STD_ULOGIC*. *B* permet de définir le type

STD_ULOGIC directement dans la clause SETS, par contre il n'accepte pas de définir les sous-ensembles dans la même clause parce qu'un élément ne peut pas appartenir aux deux ensembles au même temps. Ensuite, les sous types (sous-ensembles) sont déclarés comme étant des *CONSTANTS* qui ont des propriétés (*PROPERTIES*). Les ***CONSTANTS*** en B permet de définir des noms de composants qui ne vont pas être changés jusqu à l'implémentation les attributs de ces constantes doivent être déclarées dans la clause ***PROPERTIES***.

Exemple :

En VHDL

```
Type std_ulogic IS ('U','X','0','1','Z','W','L','H','-')

SUBTYPE std_logic IS resolved std_ulogic
```

En B

```
CONSTANTS STD_LOGIC ...
```

En B

```
PROPERTIES
          STD_LOGIC <: STD_ULOGIC
          STD_LOGIC = ('UU','XX','OO','II','ZZ','WW','LL','HH')
          STD_LOGIC = STD_ULOGIC - DD
 ..
```

En VHDL

```
SUBTYPE
 XO1 IS
resolved std_ulogic RANGE 'X' TO '0'    CONSTANTS UXOI ...
```

En B

```
PROPERTIES UXOI <: STD_ULOGIC UXOI = ('UU','XX','OO','II')
 ..
```

B

```
SETS
  STD_ULOGIC = ('UU','XX','OO','II','ZZ','WW','LL','HH','DD')
...
```

VHDL

```
SUBTYPE std_logic IS resolved std_ulogic
```

B

```
CONSTANTS
  STD_LOGIC
PROPERTIES
  STD_LOGIC <: STD_ULOGIC STD_LOGIC= ('UU','XX','OO','II',
                                      'ZZ','WW','LL','HH')
  STD_LOGIC = STD_ULOGIC- DD
  ..
```

VHDL

```
SUBTYPE XO1 IS resolved std_ulogic RANGE 'X' TO '0'
```

B

```
CONSTANTS
    UXOI
PROPERTIES
    UXOI <: STD_ULOGIC UXOI =('UU','XX','OO','II')
    ..
```

Les appels des fonctions sont définis comme étant des appels des opérations en *B*. Mais les comportements des fonctions sont déclarés comme des tableaux de relations constantes en *B*. Les relations doivent être définies sur tout le domaine « le type » de la fonction. Le typage d'entrée est vérifié avant chaque appel de fonction ; « opération », par une précondition en *B*. Mais le typage de sortie de fonction n'est pas vérifié parce qu'il est garanti par les tableaux des relations constantes. Contrairement à *VHDL*, *B* n'accepte pas le « overloading » ; c'est-à-dire, deux opérations ne peuvent pas avoir le même nom, même si leurs signatures sont différentes. Pour résoudre ce problème, on a choisi d'ajouter une partie de signature ; le type d'entrée au nom de l'opération. Par exemple en *STD_LOGIC_1164* six fonctions différentes sont utilisées pour convertir les valeurs de types différents en type standard X01 ; qui contient les valeurs X, 0 et 1, toutes ces fonctions s'appellent TO_X01. Les opérations correspondantes en *B* vont avoir des noms comme From_std_ulogic_To X01.

En VHDL

```
CONSTANT
```

```
  Not_table : std_table
      : stdlogic_1d :=('U','X','1','0','X','X','1','0','X')
            -- = NOT ('U','X','0','1','Z','W','L','H','-')

Function "NOT" ( l : std_ulogic )
   RETURN UX01 IS BEGIN RETURN (not_table( l ))
END "NOT" ;
```

En B

```
CONSTANTS
     NOT_STD PROPERTIES NOT_STD -> UXOI

OPERATIONS
   Out <- NOT ( A)
    PRE
      A: STD_ULOGIC
    THEN
      Out := NOT_STD (A)
    END
```

2. **La modélisation des vecteurs**
   La structure de vecteurs n'existe pas en *B*. Ceci nécessite la création de ce type en utilisant des expressions mathématiques plus simples. On a choisi de représenter et regrouper tous les éléments de *STD_LOGIC_1164* dans une machine indépendante qui peut, en voyant la machine précédente par la clause *SEES*, gérer les éléments de vecteur un par un. Cette machine, *B_STD_LOGIC_V_0*, contient principalement une définition d'un ensemble d'éléments limités par une taille maximale paramétrée, Max_Element . Chaque élément contient deux champs : un indice et une valeur de type *STD_ULOGIC*. La définition contient aussi la taille réelle de chaque vecteur.

```
CONSTANTS
    ...
 ,   STD_ULOGIC_VECTOR
 ,   Max_Element


PROPERTIES
```

```
    Max_Element :   NAT1
&    STD_ULOGIC_VECTOR   = struct (
                        vector:1.. Max_Element --> STD_LOGIC,
                        vector_size :   NAT1
                        )

&   card(STD_ULOGIC_VECTOR) <   MAX_VECTOR
&   !xx.((xx:STD_ULOGIC_VECTOR)
             => xx'vector_size < Max_Element)
```

L'application d'une opération sur un vecteur dans la machine *STD_LOGIC_V_0* est exécutée en appelant l'opération correspondante dans la machine STD_LOGIC_0 et ceci sur tous les éléments du vecteur un par un. La répétition d'une partie importante de code dans toute les opérations nous a permis d'extraire cette partie dans une macro indépendante, DEFINITION en *B*.

```
DEFINITIONS
  APPLY(op,in1,in2,out)   ==
   ANY
    vv
   WHERE
      vv : STD_LOGIC_VECTOR
    & !xx.( (xx : 1 .. max({in1'vector_size,in2'vector_size}
                          )
           )
           => ((vv'vector)(xx) = op(    (in1'vector)(xx)
                                    , (in2'vector)(xx)
                                   )
               )
          )
   THEN
     out := vv
   END

;   MAX_VECTOR      ==  1000


OPERATIONS

 out <-- And(in1,in2)=
```

```
 PRE
     in1 : STD_LOGIC_VECTOR
 &   in2 : STD_LOGIC_VECTOR
 THEN
 APPLY(AND_STD,in1,in2,out)
END;
```

II- Portes standard

Boulanger et Mariano ont produit une librairie de machines B [78] de composants B pour représenter les portes logique standards utilisés dans la conception électronique, comme les portes logiques `and, or`. Dans notre projet, on peut voir et inclure ces composants.

Chapitre 4

Exemples

Nous allons présenter dans ce chapitre quelques exemples pour illustrer la méthode de conception décrite plus haut.

4.1 Exemple 1 : adder 4Bit

On voudrait créer un additionneur de 2 entiers naturels codés sur 4 bits ; c'est-à-dire compris entre 0 et 15. On va voir comment démarre la conception en ***VGUI*** et comment le code ***VHDL*** et ***B*** se génère et se prouve en utilisant l'*AtelierB*.

4.1.1 Le développement de adder4bit en VGUI :

Au départ, on précise l'interface externe du modèle et ses fonctionnalités. La création d'un nouveau fichier en ***VGUI*** est considérée comme une ouverture d'un module « top_level » ce que donne un tableau noir pour préciser les composants de ce module.
On a employé pour cette déclaration les éléments suivants :

- Trois ports externes,
- Une boite (module),
- Trois connexions orientées entre les ports externes et la boite : deux entrées et une sortie.

Dans les propriétés de la boite, on précise son nom, *AddInteger1*, et son type, *AddInteger*. On précise également, dans le champ « *objectif* », sa spécification fonctionnelle : s=a+b .

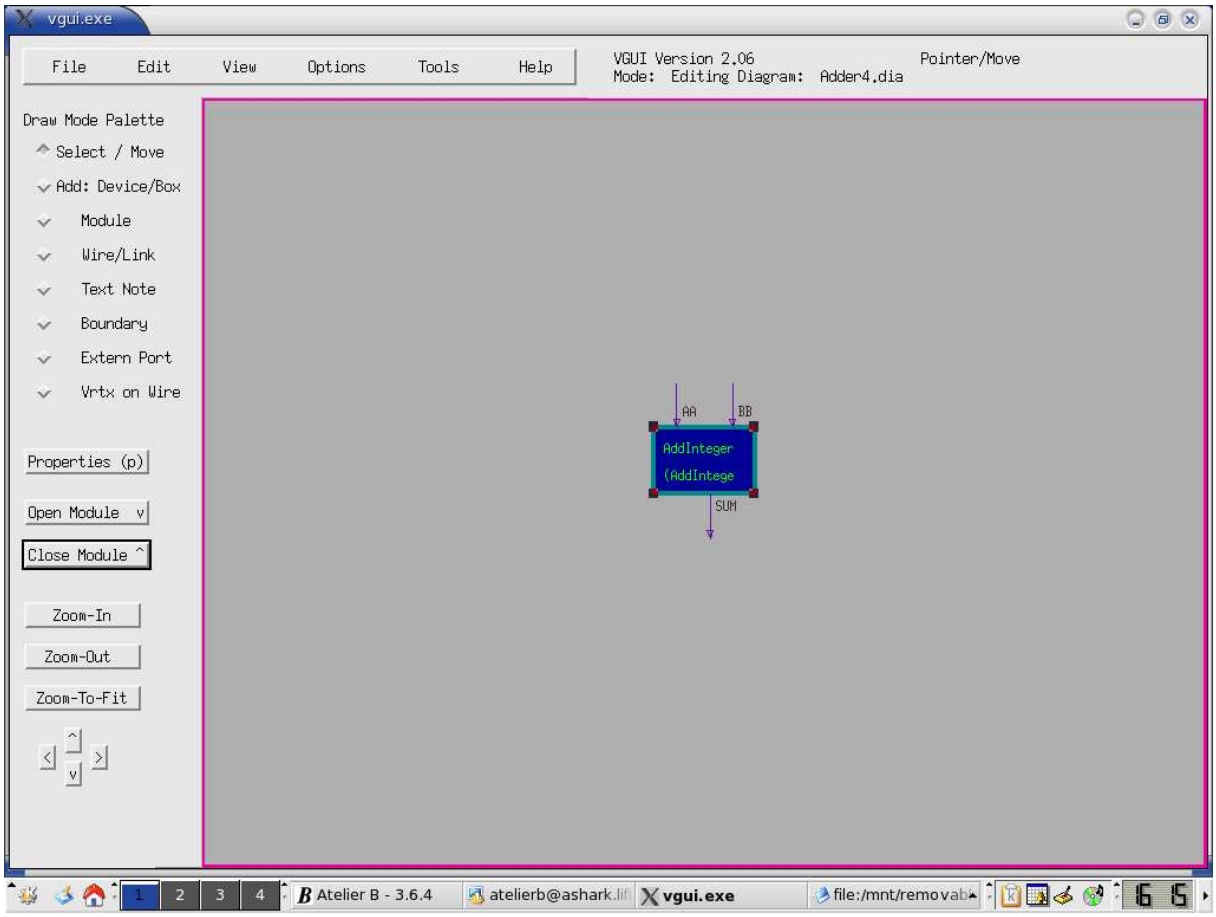

FIG. 4.1 – Adder (toplevel).

Les propriétés de connexion peuvent être employées pour préciser les noms des variables (ou les signaux), les directions des entrées/sorties et leurs types.

Les détails sur le type *AddInteger*, peuvent être vus, créés ou modifiés en ouvrant la boite *AddInteger1*. Puisque ce type n'était pas déclaré auparavant, l'ouverture de cette boite nous permet d'accéder à un tableau noir vide pour déclarer les détails du module. Ainsi, les trois ports externes correspondant aux connexions externes « deux entrées et une sortie » doivent avoir les même noms, les mêmes types, et les mêmes directions de signaux. Pour concrétiser notre conception on utilise un additionneur binaire. On ajoute pour cela des convertisseurs sur les entrées/sorties. Dans notre cas, le tableau noir de AddInteger contient quatre boites :

1. Deux convertisseurs *Integer2Bit_A* et *Integer2Bit_B* de même type, *In-*

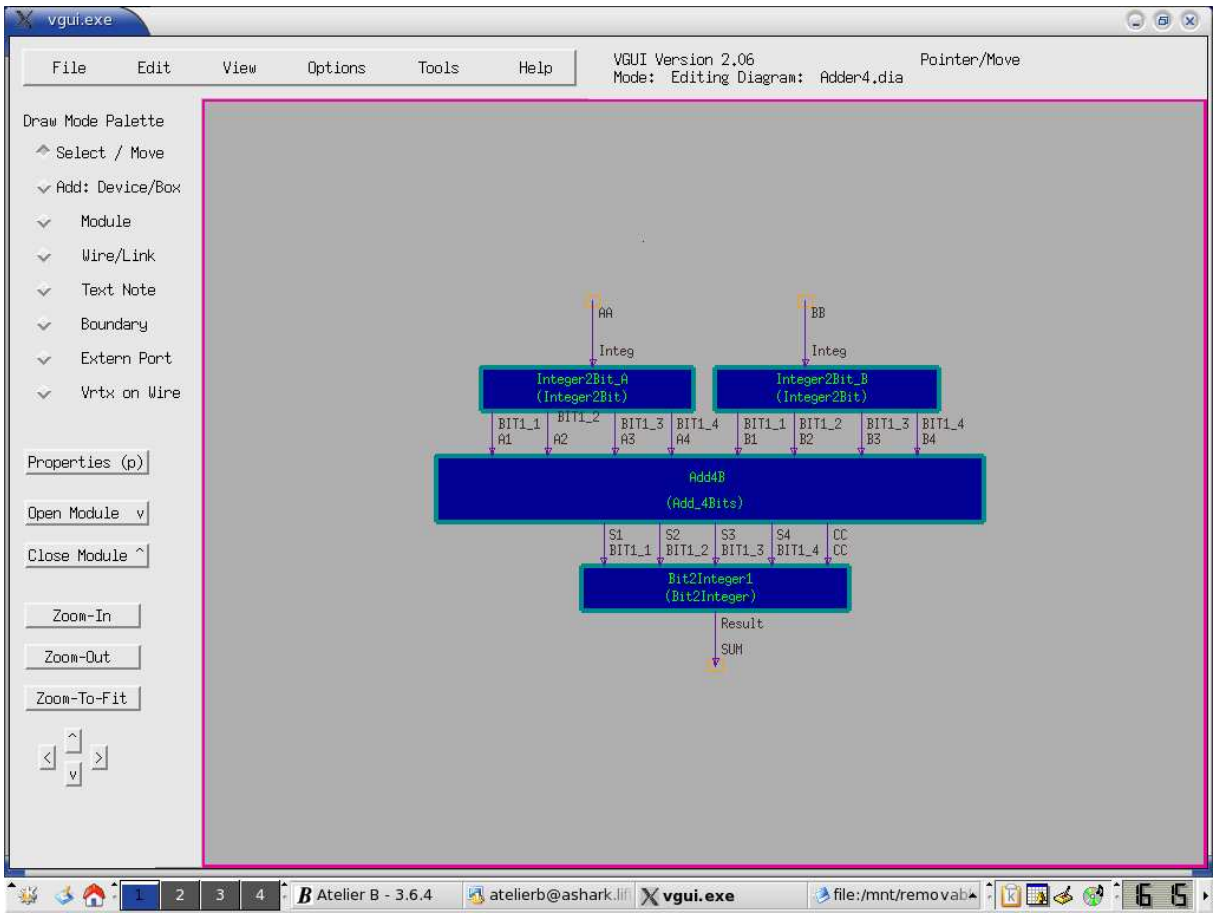

FIG. 4.2 – AddInteger.

teger2Bit, permettant de traduire les entiers en code binaire. Ce type de module a une entrée de type *Bit4* et quatre sorties de type *Bit*. La spécification fonctionnelle de ce type de boite est :

$$i = a + 2 * (b + 2 * (c + 2 * d)).$$

2. Une boite, *Add4*, qui représente un additionneur des variables de types Bit. Les sorties de deux convertisseurs précédents sont les entrées de cette nouvelle boite, qui contient par conséquent 8 entrées de type *Bit*. Le résultat de l'addition est aussi de type binaire ; « *Bit* », ainsi on a quatre sorties de type *Bit* avec une sortie supplémentaire pour représenter, le cas échéant, la retenue. Le type choisi est *Add_4Bits*. La spécification de cette boite est :

$$(a+o+2*(b+p+2*(c+q+2*(d+r))) = v+2*(w+2*(x+2*(y+2*z))))$$

3. Une boite, *Bit2Integer1*, pour représenter un convertisseur qui prend les cinq sorties de type *Bit* de boite précédente pour les convertir en un entier de type *Bit5*. La sortie de cette boite est connectée à la sortie externe qui représente à son tour la sortie de l'additionneur principal au premier niveau. La spécification fonctionnelle de cette boite est :

$$s = a + 2 * (b + 2 * (c + 2 * (d + 2 * e)))$$

Les trois modules sont à leurs tours détaillés en utilisant d'autres composants plus élémentaires.

(a)- Le convertisseur Integer2Bit

Ce module convertit un entier naturel en format binaire. Pour réaliser cette opération, on itère l'algorithme de division par deux : On divise l'entier à convertir « *Integ* » par deux. Le reste de cette division, 0 ou 1, représente le chiffre le plus à droite en format binaire de cet entier. Le quotient de cette première division, s'il est différent de 0 ou 1, est divisé à son tour par deux. Le reste de cette nouvelle division, 0 ou 1, représente la valeur du bit situé immédiatement à gauche du bit précédent. On répète cette dernière opération jusqu'à obtention d'un quotient égal à « 0 ou 1 », dernier bit à placer à gauche du bit précédent.

Finalement, pour appliquer cette conversion à un élément de l'ensemble [0,..,15], trois opérations sont suffisantes, comme le montre le tableau noir de module *Integer2Bit*. Trois boites, *ModDiv2a*, *ModDiv2b* et *ModDiv2c* de même type « *ModDiv2* », sont utilisés pour exécuter l'algorithme. *ModDiv2* représente un module à une entrée de type entier et deux sorties l'un de type entier et l'autre de type binaire. La sortie de type entier représente le quotient et la sortie binaire représente le reste. La spécification fonctionnel de ce module est comme suit :

$$\mathrm{i} = 2^{*}\mathrm{j} + \mathrm{k}$$

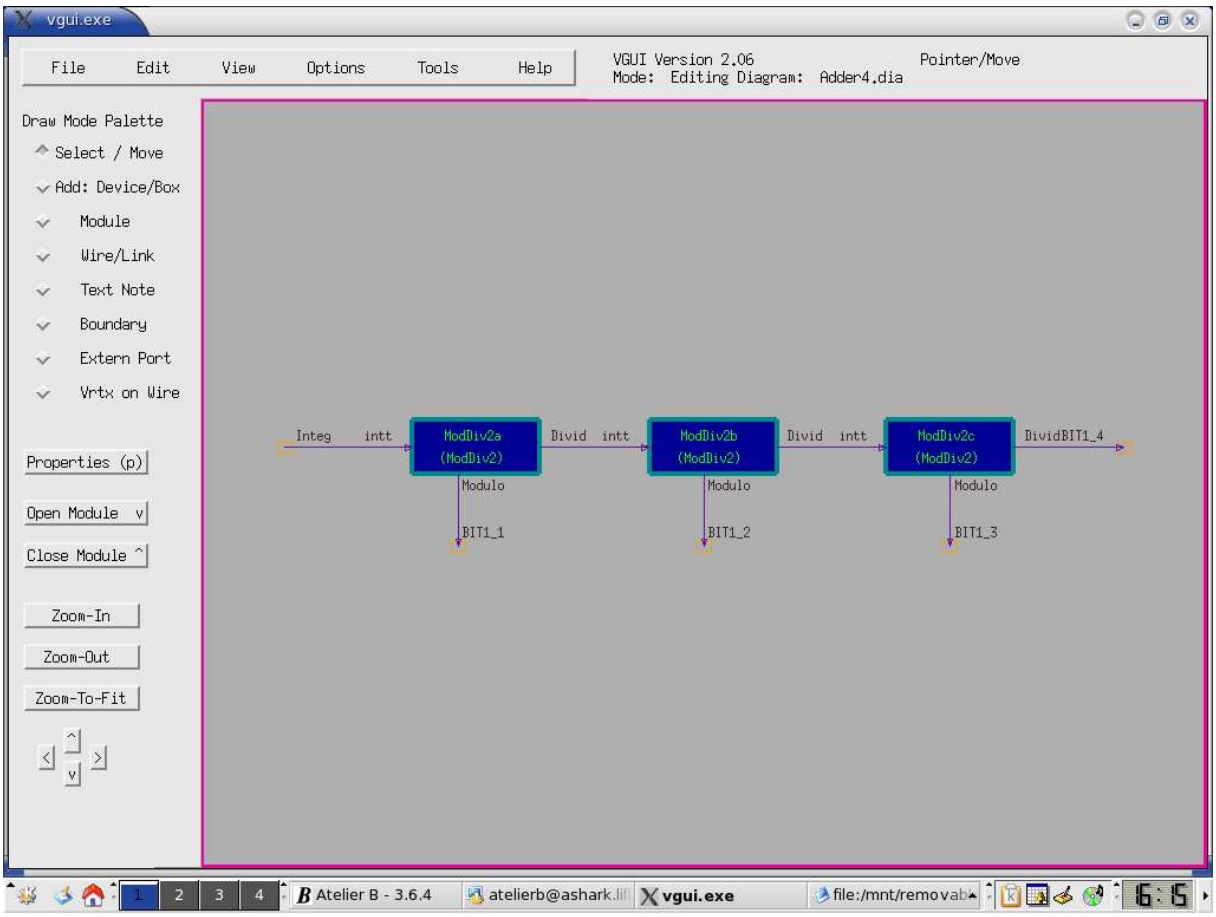

FIG. 4.3 – Integer2Bit.

(b)- Le module Add_4Bits

La tache principale de ce module, qui est l'addition de deux nombres binaires de taille « 4 bits », est décomposée en quatre taches similaires et plus simples : l'addition de chaque deux bits correspondants de chaque nombre en prenant en compte le retenue qui peut résulter de l'addition des bits précédents. Le tableau noir de ce module contient cinq boites.

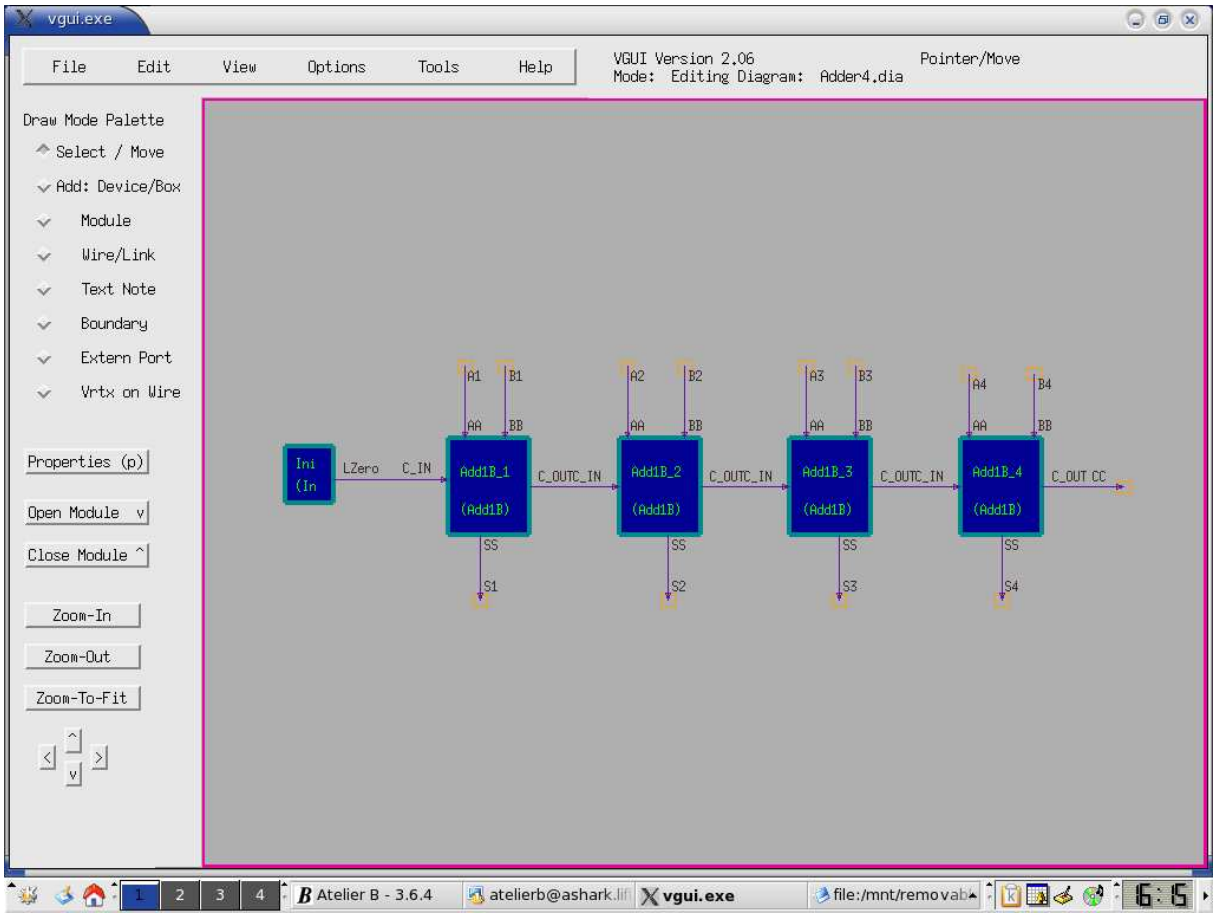

FIG. 4.4 – Add4Bit.

Add1B_1, Add1B_2, Add1B_3 et ***Add1B_4*** sont similaires et ils ont le même type Add1B. Ce type a trois entrées et deux sorties. Les entrées sont de type *Bit* dont deux représentent les bits qui doivent additionner et le retenue possible des bits précédents. Les deux sorties de type *Bit* représentent le résultat et la retenue calculée. La spécification de ces boites est :

$$((a+b+c=3) \Rightarrow (d=1 \;\&\; s=1))$$

$$((a+b+c=2) \Rightarrow (d=1 \;\&\; s=0))$$

$$((a+b+c=1) \Rightarrow (d=0 \;\&\; s=1))$$

$$((a+b+c=0) \Rightarrow (d=0 \;\&\; s=0))$$

La cinquième boite est utilisée pour initialiser la valeur initiale de la retenue à zéro.

(c)- Le module Bit2Integer :

L'algorithme utilisé pour créer ce module est la décomposition directe de l'expression

$$s = a + 2 * (b + 2 * (c + 2 * (d + 2 * e)))$$

On a cinq boites Mul21, Mul2a, Mul2b, Mul2c et Mul2d de type Mul2. Elles reçoivent les sorties de l'additionneur binaire. Chaque boite reçoit un bit de l'additionneur binaire et elle reçoit au même temps le résultat de la boite précédente, elle calcule la somme et elle multiplie le résultat par deux ensuite elle envoie le résultat au boite suivante. Le résultat de la dernière boite est le résultat de la conversion. Une boite d'initialisation à zéro est créée seulement pour préparer l'entrée de première boite car elle n'est pas précède par une boite de multiplication. La spécification fonctionnelle de chaque boite est :

$$r = 2 * i + m$$

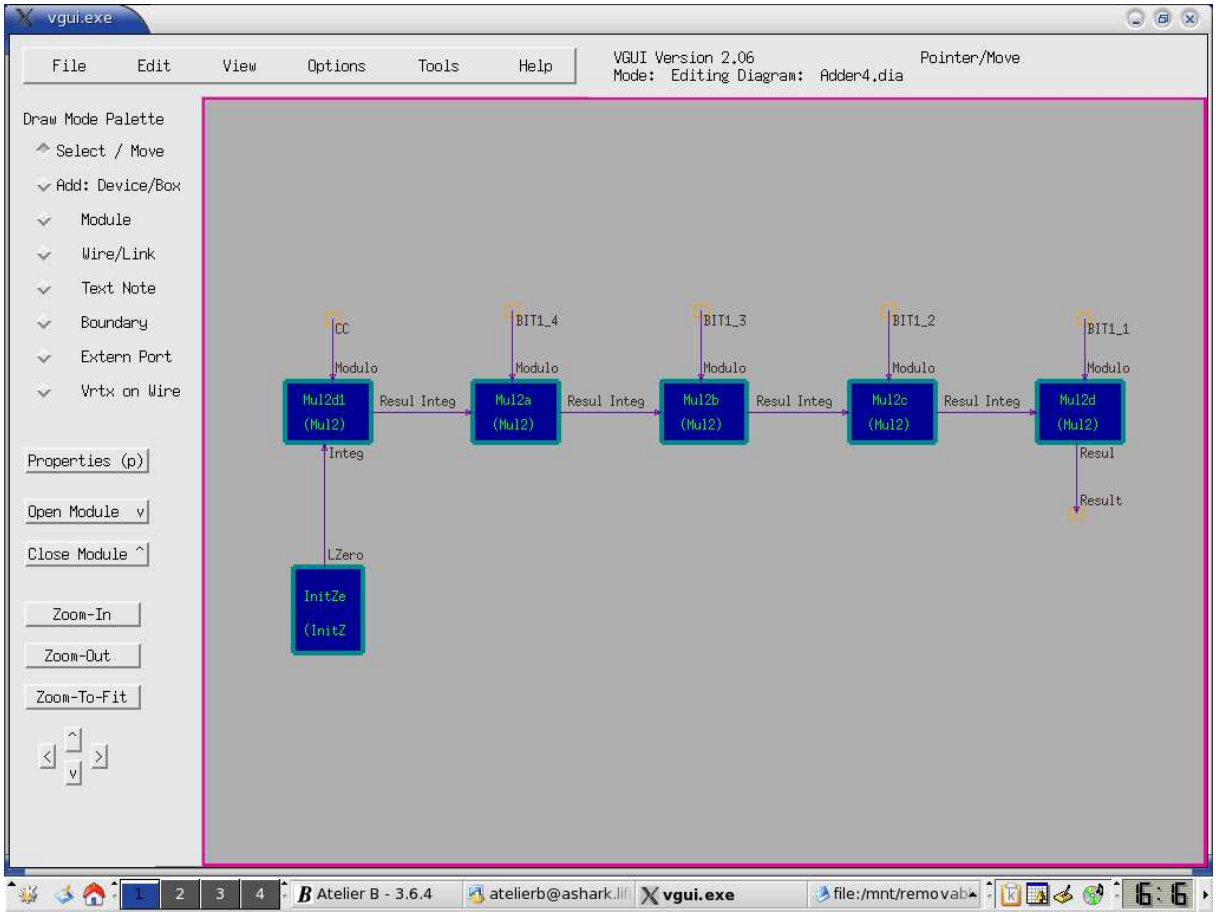

FIG. 4.5 – Bit2Integer.

4.1.2 Le code en VHDL

VGUI a exporté le code ***BHDL***. En utilisant le parseur et le treewalker, ***ANTLR*** a généré les entités et les architectures équivalent à notre conception. La vue externe de chaque module a produit une entité ; la vue interne a produit une architecture. Chaque entité contient des informations sur l'interface : les noms des signaux, leur types et ses directions. L'entité principale est *AddInteger* :

```
ENTITY  AddInteger  IS
  PORT(  SUM    : OUT BIT5;
         BB, AA : IN BIT4
      );
END AddInteger;
```

Le tableau noir de AddInteger a produit l'architecture suivante. ***VGUI*** donne le même nom, structure, pour toutes les architectures. Donc elles seront distinguées à partir de leurs types. Dans cette architecture, on trouve trois composants correspondants aux trois modules, *Integer2Bit_A, Add_4Bits* et *Bit2Integer*, qui existent dans la vue interne de *AddInteger*. Le composant est instancié deux fois et les autres une seule fois. On voit ici des noms de signaux « L0, ..., Ln ». Ils servent à représenter la connexion entre ces trois différentes boites.

1. **L'architecture de l'additionneur qui montre ses différents composants :**

```
ARCHITECTURE  structure  OF  AddInteger  IS

SIGNAL  L0, L1, L2, L3, L4, L5, L6, L7, L8, L9, L10, L14, L15
                                              :  BIT1;

    COMPONENT Add_4Bits
    PORT(
            CC, S1, S4, S3, S2 : OUT BIT1;
            B4,A4, B3, A3, B2, A2, B1, A1  : IN BIT1 );
    END COMPONENT;


    COMPONENT Integer2Bit
    PORT(
        BIT1_3, BIT1_4, BIT1_2, BIT1_1 : OUT BIT1;
        Integ   : IN BIT4 );
    END COMPONENT;


    COMPONENT Bit2Integer
    PORT(
```

```
            CC : IN BIT1; Result  : OUT BIT5;
            BIT1_1, BIT1_2, BIT1_3, BIT1_4  : IN BIT1 );
    END COMPONENT;


BEGIN
 Add4B : Add_4Bits
   PORT MAP( L10, L6, L9, L8, L7, L5,
          L14,L4,L15, L3, L1, L2, L0 );

 Integer2Bit_A : Integer2Bit
   PORT MAP( L15, L14, L1, L0, AA );

 Integer2Bit_B : Integer2Bit
   PORT MAP( L4, L5, L3, L2, BB );

 Bit2Integer1 : Bit2Integer
   PORT MAP( L10, SUM, L6, L7, L8, L9 );

END structure;
```

2. **L'entité de convertisseur d'un variable de type entier en binaire :**

```
ENTITY  Integer2Bit  IS
  PORT(
        BIT1_3, BIT1_4, BIT1_2, BIT1_1 :OUT BIT1;
        Integ  : IN BIT4
       );
END Integer2Bit;
```

3. **L'architecture de convertisseur d'un variable de type entier en binaire :**

```
ARCHITECTURE  structure  OF  Integer2Bit  IS
  SIGNAL  L2    : BIT3;
  SIGNAL  L3  : BIT2;

  COMPONENT ModDiv2
    PORT(
         intt   : IN Integer;
         Divid  : OUT Integer;
```

```
          Modulo : OUT BIT1
        );
   END COMPONENT;

BEGIN
  ModDiv2a : ModDiv2
    PORT MAP( Integ, L2, BIT1_1 );

  ModDiv2b : ModDiv2
    PORT MAP( L2, L3, BIT1_2 );

  ModDiv2c : ModDiv2
    PORT MAP( L3, BIT1_4, BIT1_3 );

END structure;
```

4. **L'entité de unité qui initialise un variable de type entier par la valeur zero :**

```
ENTITY  InitZero  IS
  PORT(  LZero  : OUT Integer
      );
END InitZero;
```

5. **L'entité de l'additionneur de deux nombres composées de quatre bits chaque un :**

```
ENTITY  Add_4Bits  IS
  PORT(  CC, S1, S4, S3, S2 : OUT BIT1;
     B4, A4, B3, A3, B2, A2, B1, A1 : IN BIT1
      );
END Add_4Bits;
```

6. **additionneur de deux variables de type bit :**

```
COMPONENT Add1B
    PORT(    C_IN   : IN BIT1;
             C_OUT  : OUT BIT1;
             BB, AA : IN BIT1;
             SS     : OUT BIT1
        );
```

7. **L'entité de multiplicateur d'un entier par deux :**

```
ENTITY  Mul2  IS
  PORT(  Modulo : IN BIT1;
     Integ  : IN Integer;
     Resul  : OUT Integer
      );
END Mul2;
```

8. **l'entité de diviseur d'un entier par deux :**

```
ENTITY  ModDiv2  IS
  PORT(  intt   : IN Integer;
         Divid  : OUT Integer;
         Modulo : OUT BIT1
      );
END ModDiv2;
```

9. **L'entité pour convertir un bit en entier :**

```
ENTITY  Bit2Integer  IS
  PORT(  CC : IN BIT1;
         Result : OUT BIT5;
         BIT1_1, BIT1_2, BIT1_3, BIT1_4 : IN BIT1
      );
END Bit2Integer;
```

10. **L'entité d'additionneur de deux variables de type bit :**

```
ENTITY  Add1B  IS
  PORT(  C_IN   : IN BIT1;
         C_OUT  : OUT BIT1;
         BB, AA : IN BIT1;
         SS     : OUT BIT1
      );
END Add1B;
```

11. **L'architecture de convertisseur d'une variable de type bit en entier** :

```
ARCHITECTURE  structure  OF  Bit2Integer  IS
  SIGNAL  L7    : BIT1;
  SIGNAL  L9    : BIT2;
  SIGNAL  L11   : BIT3;
  SIGNAL  L13   : BIT4;
  SIGNAL  L16  : BIT1;
```

```
  COMPONENT Mul2
    PORT(    Modulo : IN BIT1;
         Integ  : IN Integer;
         Resul  : OUT Integer
        );
   END COMPONENT;

  COMPONENT InitZero
    PORT(    LZero  : OUT Integer
        );
   END COMPONENT;

BEGIN
  Mul2a : Mul2
    PORT MAP( BIT1_4, L16, L9 );

  Mul2b : Mul2
    PORT MAP( BIT1_3, L9, L11 );

  Mul2c : Mul2
    PORT MAP( BIT1_2, L11, L13 );

  Mul2d : Mul2
    PORT MAP( BIT1_1, L13, Result );

  InitZero2 : InitZero
    PORT MAP( L7 );

  Mul2d1 : Mul2
    PORT MAP( CC, L7, L16 );

END structure;
```

4.1.3 Le code B

La génération automatique de code B de ce projet a produit des composants B. Chaque composant B correspond à un seul composant ***VHDL***.

1. **La machine Add1B.** Voir l'entité *Add1B*, page 129 :

```
MACHINE
    Add1B

SEES
       Signal_type


VARIABLES
       C_IN
     , C_OUT
     , BB
     , AA
     , SS


DEFINITIONS
    inv_Add1B ( c, d, b, a, s )
 ==    (a:BIT1 & b:BIT1 & c:BIT1 & d:BIT1 & s:BIT1 &
       ((a+b+c=3) => (d=1 & s=1)) &
       ((a+b+c=2) => (d=1 & s=0)) &
       ((a+b+c=1) => (d=0 & s=1)) &
       ((a+b+c=0) => (d=0 & s=0)) &
       )


INVARIANT
    inv_Add1B ( C_IN, C_OUT, BB, AA, SS )


INITIALISATION
     C_IN, C_OUT, BB, AA, SS:(
              inv_Add1B ( C_IN, C_OUT, BB, AA, SS )
                        )
```

```
OPERATIONS
    type_Add1B =
    PRE true
    THEN  C_IN
         ,C_OUT
         ,BB
         ,AA
         ,SS
          :( inv_Add1B ( C_IN, C_OUT, BB, AA, SS )
)
    END


END
```

2. **La machine *Add_4Bits***. Voir l'entité *Add_4Bits*, page 129 :

```
MACHINE
    Add_4Bits

SEES
       Signal_type

VARIABLES
       CC
     , S1
     , S4
     , S3
     , S2
     , B4
     , A4
     , B3
     , A3
     , B2
     , A2
     , B1
     , A1


DEFINITIONS
    inv_Add_4Bits ( z, v, y, x, w, r, d, q, c, p, b, o, a )
 == (a:BIT1 & b:BIT1 & c:BIT1 & d:BIT1 &
    o:BIT1 & p:BIT1 & q:BIT1 & r:BIT1 &
    v:BIT1 & w:BIT1 & x:BIT1 & y:BIT1 & z:BIT1 &
    (a+o+2*(b+p+2*(c+q+2*(d+r)))=v+2*(w+2*(x+2*(y+2*z)))))


INVARIANT
    inv_Add_4Bits ( CC, S1, S4, S3, S2, B4,
                    A4, B3, A3, B2, A2, B1, A1
                  )


INITIALISATION
     CC, S1, S4, S3, S2, B4, A4, B3, A3, B2, A2, B1, A1
            :( inv_Add_4Bits ( CC, S1, S4, S3, S2, B4, A4,
```

```
                                    B3, A3, B2, A2, B1, A1
                                   )
                )

    OPERATIONS
        type_Add_4Bits =
        PRE true
        THEN  CC
              ,S1
              ,S4
              ,S3
              ,S2
              ,B4
              ,A4
              ,B3
              ,A3
              ,B2
              ,A2
              ,B1
              ,A1
              :( inv_Add_4Bits ( CC, S1, S4, S3, S2, B4, A4, B3,
                                                    A3, B2, A2, B1,A1 ) )
        END


    END
```

3. **La machine *AddInteger***. Voir l'entité *AddInteger*, page 127 :

```
MACHINE
    AddInteger

SEES
       Signal_type


VARIABLES
       SUM
     , BB
     , AA


DEFINITIONS
    inv_AddInteger ( s, b, a )
 == (a:BIT4 & b:BIT4 & s:BIT5 & s=a+b)


INVARIANT
    inv_AddInteger ( SUM, BB, AA )


INITIALISATION
     SUM, BB, AA:( inv_AddInteger ( SUM, BB, AA )
)

OPERATIONS
    type_AddInteger =
    PRE true
    THEN  SUM
          ,BB
          ,AA
           :( inv_AddInteger ( SUM, BB, AA )
)
    END


END
```

4. **La machine *Bit2Integer***. Voir l'entité Bit2Integer, page 130 :

```
MACHINE
    Bit2Integer

SEES
       Signal_type


VARIABLES
       CC
     , Result
     , BIT1_1
     , BIT1_2
     , BIT1_3
     , BIT1_4


DEFINITIONS
    inv_Bit2Integer ( e, s, b, c, d, a )
 == (a:BIT1 & b:BIT1 & c:BIT1 & d:BIT1 & e:BIT1 &
                  s:BIT5 & s=a+2*(b+2*(c+2*(d+2*e))))


INVARIANT
    inv_Bit2Integer ( CC, Result, B2, B3,
                                  B4, B1 )


INITIALISATION
     CC, Result, BIT1_1, BIT1_2, BIT1_3, BIT1_4:(
                      inv_Bit2Integer ( CC, Result, B2, B3, B4, B1 )
                                                    )

OPERATIONS
    type_Bit2Integer =
    PRE true
    THEN  CC
         ,Result
         ,BIT1_1
         ,BIT1_2
```

```
          ,BIT1_3
          ,BIT1_4
           :( inv_Bit2Integer ( CC, Result, B2, B3, B4, B1 )
)
    END
END
```

5. **La machine *InitZero***. Voir l'entité *InitZero*, page 129 :

```
MACHINE
    InitZero

SEES
       Signal_type


VARIABLES
       LZero


DEFINITIONS
    inv_InitZero ( a )
 == (a=0)


INVARIANT
    inv_InitZero ( LZero )


INITIALISATION
     LZero:( inv_InitZero ( LZero )
)

OPERATIONS
    type_InitZero =
    PRE true
    THEN  LZero
          :( inv_InitZero ( LZero )
)
    END
END
```

6. **La machine *Integer2Bit*.** Voir l'entité *Integer2Bit*, page 128 :

```
MACHINE
    Integer2Bit

SEES
       Signal_type

VARIABLES
       BIT1_3
     , BIT1_4
     , BIT1_2
     , BIT1_1
     , Integ

DEFINITIONS
    inv_Integer2Bit ( c, d, b, a, i )
 == (i:BIT4 & a:BIT1 & b:BIT1 & c:BIT1 & d:BIT1 &
                i=a+2*(b+2*(c+2*d)))

INVARIANT
    inv_Integer2Bit ( BIT13, BIT14, BIT12, BIT11, Integ )

INITIALISATION
     BIT1_3, BIT1_4, BIT1_2, BIT1_1, Integ:
     ( inv_Integer2Bit ( BIT13, BIT14, BIT12, BIT11, Integ )
)

OPERATIONS
    type_Integer2Bit =
    PRE true
    THEN  BIT1_3
         ,BIT1_4
         ,BIT1_2
         ,BIT1_1
         ,Integ
          :( inv_Integer2Bit ( BIT13, BIT14, BIT12, BIT11, Integ )
)
    END
END
```

7. **La machine *Mul2***. Voir l'entité *Mul2*, 129, et le composant *Mul2* dans l'architecture *Bit2Integer*, page 130 :

```
MACHINE
    Mul2

SEES
       Signal_type


VARIABLES
       Modulo
     , Integ
     , Resul


DEFINITIONS
    inv_Mul2 ( m, i, r )
 == (m:BIT1 & i:BIT4 & r:BIT5 & r=2*i+m)


INVARIANT
    inv_Mul2 ( Modulo, Integ, Resul )


INITIALISATION
     Modulo, Integ, Resul:( inv_Mul2 ( Modulo, Integ, Resul )
)

OPERATIONS
    type_Mul2 =
    PRE true
    THEN  Modulo
         ,Integ
         ,Resul
          :( inv_Mul2 ( Modulo, Integ, Resul )
)
    END


END
```

8. **La machine *ModDiv2***. Voir l'entité *ModDiv2*, page 130 :

```
MACHINE
    ModDiv2

SEES
       Signal_type


VARIABLES
       intt
     , Divid
     , Modulo


DEFINITIONS
    inv_ModDiv2 ( i, j, k )
 == (i:BIT4 & j:BIT4 & k:BIT1 &
             i=2*j+k)


INVARIANT
    inv_ModDiv2 ( intt, Divid, Modulo )


INITIALISATION
     intt, Divid, Modulo:( inv_ModDiv2 ( intt, Divid, Modulo )
)

OPERATIONS
    type_ModDiv2 =
    PRE true
    THEN  intt
         ,Divid
         ,Modulo
          :( inv_ModDiv2 ( intt, Divid, Modulo )
)
    END


END
```

9. **Le Raffinement *structure_Add_4Bits*** :

```
REFINEMENT
    structure_Add_4Bits

REFINES
    Add_4Bits

SEES
    Signal_type

INCLUDES
    Add1B_1.Add1B,
    Add1B_2.Add1B,
    Add1B_3.Add1B,
    Add1B_4.Add1B,
    InitZero1.InitZero

VARIABLES
    LZero, L12, B1, A1, S1, L13, B2, A2, S2,
     L14, B3, A3, S3, CC, B4, A4, S4

DEFINITIONS
    inv_structure_Add_4Bits ==
       (
       inv_Add1B( LZero, L12, B1, A1, S1  )
       & inv_Add1B( L12, L13, B2, A2, S2  )
       & inv_Add1B( L13, L14, B3, A3, S3  )
       & inv_Add1B( L14, CC, B4, A4, S4  )
       & inv_InitZero( LZero  )
        )

INVARIANT
     inv_structure_Add_4Bits

INITIALISATION
     LZero, L12, B1, A1, S1, L13, B2, A2, S2, L14,
     B3, A3, S3, CC, B4, A4, S4
     : ( inv_structure_Add_4Bits )

OPERATIONS
```

```
    type_Add_4Bits =
    PRE true
    THEN  LZero, L12, B1, A1, S1, L13, B2, A2,
          S2, L14, B3, A3, S3, CC, B4, A4, S4
          : ( inv_structure_Add_4Bits )
    END

END
```

10. **Le raffinement *AddInteger***. Voir l'architecture correspondante page 127 :

```
REFINEMENT
    structure_AddInteger

REFINES
    AddInteger

SEES
    Signal_type

INCLUDES
    Add4B.Add_4Bits,
    Integer2Bit_A.Integer2Bit,
    Integer2Bit_B.Integer2Bit,
    Bit2Integer1.Bit2Integer

VARIABLES
    L10, L6, L9, L8, L7, L5, L14, L4,
    L15, L3, L1, L2, L0, AA, BB, SUM

DEFINITIONS
    inv_structure_AddInteger ==
       (
       inv_Add_4Bits( L10, L6, L9, L8, L7, L5, L14,
                      L4, L15, L3, L1, L2, L0  )
       & inv_Integer2Bit( L15, L14, L1, L0, AA  )
       & inv_Integer2Bit( L4, L5, L3, L2, BB  )
       & inv_Bit2Integer( L10, SUM, L6, L7, L8, L9  )
        )

INVARIANT
     inv_structure_AddInteger

INITIALISATION
     L10, L6, L9, L8, L7, L5, L14, L4,
     L15, L3, L1, L2, L0, AA, BB, SUM
     : ( inv_structure_AddInteger )

OPERATIONS
```

```
        type_AddInteger =
        PRE true
        THEN  L10, L6, L9, L8, L7, L5, L14, L4,
        L15, L3, L1, L2, L0, AA, BB, SUM
        : ( inv_structure_AddInteger )
        END

    END
```

11. **Le raffinement *structure_Bit2Integer***. Voir l'architecture correspondante page 130 :

```
REFINEMENT
    structure_Bit2Integer

REFINES
    Bit2Integer

SEES
    Signal_type

INCLUDES
    Mul2a.Mul2, Mul2b.Mul2, Mul2c.Mul2,
    Mul2d.Mul2, InitZero2.InitZero, Mul2d1.Mul2

VARIABLES
    BIT1_4, L16, L9, BIT1_3, L11, BIT1_2, L13,
    BIT1_1, Result, L7, CC

DEFINITIONS
    inv_structure_Bit2Integer ==
       (
       inv_Mul2( BIT1_4, L16, L9  )
       & inv_Mul2( BIT1_3, L9, L11  )
       & inv_Mul2( BIT1_2, L11, L13  )
       & inv_Mul2( BIT1_1, L13, Result  )
       & inv_InitZero( L7  )
       & inv_Mul2( CC, L7, L16  )
        )

INVARIANT
     inv_structure_Bit2Integer

INITIALISATION
     BIT1_4, L16, L9, BIT1_3, L11, BIT1_2,
      L13, BIT1_1, Result, L7, CC
      : ( inv_structure_Bit2Integer )

OPERATIONS
    type_Bit2Integer =
```

```
        PRE true
        THEN  BIT1_4, L16, L9, BIT1_3, L11,
        BIT1_2, L13, BIT1_1, Result, L7, CC
          : ( inv_structure_Bit2Integer )
        END

    END
```

12. **le raffinement *structure_Integer2Bit***. Voir l'architecture correspondante page 128 :

```
REFINEMENT
    structure_Integer2Bit

REFINES
    Integer2Bit

SEES
    Signal_type

INCLUDES
    ModDiv2a.ModDiv2,
    ModDiv2b.ModDiv2,
    ModDiv2c.ModDiv2

VARIABLES
    Integ, L2, BIT1_1, L3, BIT1_2, BIT1_4, BIT1_3

DEFINITIONS
    inv_structure_Integer2Bit ==
       (
       inv_ModDiv2( Integ, L2, BIT1_1  )
       & inv_ModDiv2( L2, L3, BIT1_2  )
       & inv_ModDiv2( L3, BIT1_4, BIT1_3  )
        )

INVARIANT
     inv_structure_Integer2Bit

INITIALISATION
     Integ, L2, BIT1_1, L3, BIT1_2, BIT1_4, BIT1_3
                       : ( inv_structure_Integer2Bit )
OPERATIONS
    type_Integer2Bit =
    PRE true
    THEN  Integ, L2, BIT1_1, L3, BIT1_2, BIT1_4, BIT1_3
                                : ( inv_structure_Integer2Bit )
    END
END
```

13. **Le raffinement *top_level***, qui contient la conception au niveau plus abstrait :

```
REFINEMENT
    structure_top_level

REFINES
    top_level

SEES
    Signal_type

INCLUDES
    AddInteger1.AddInteger

VARIABLES
    BBL0

DEFINITIONS
    inv_structure_Bit2Integer ==
       (
       inv_AddInteger( BBL0, BBL0, BBL0  )
        )

INVARIANT
     inv_structure_Bit2Integer

INITIALISATION
     BBL0 : ( inv_structure_Bit2Integer )

OPERATIONS
    type_Bit2Integer =
    PRE true
    THEN  BBL0 : ( inv_structure_Bit2Integer )
    END

END
```

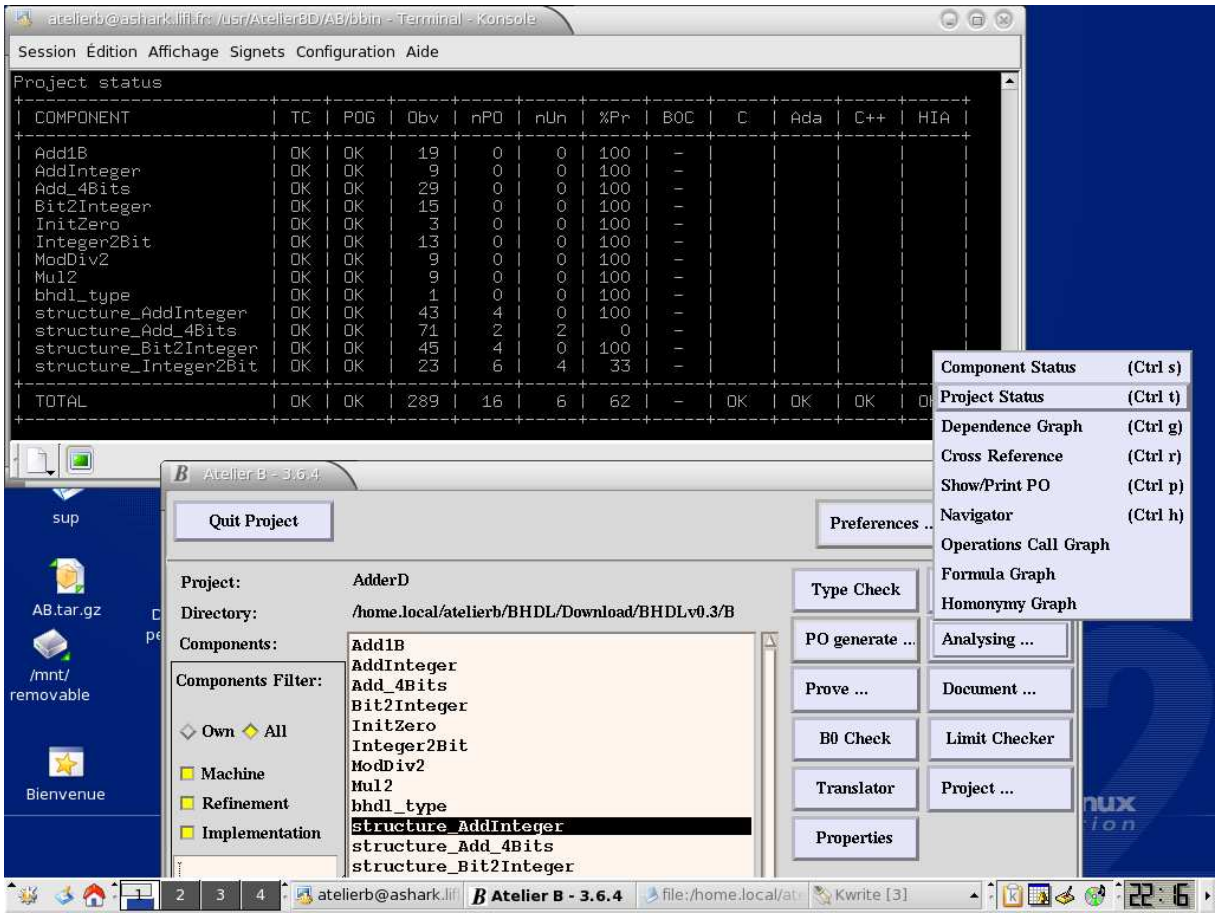

FIG. 4.6 – Projet Adder4 dans l'AtelierB.

4.1.4 Résultat de L'analyse de code

L'analyse complète du projet a montré que :

1. le code généré est correct. Il est complètement coherent.
2. un nombre total des 305 obligations de preuves nécessaires pour prouver notre conception, dont
   - 209 sont triviales ou facilement prouvées par l'*AtelierB*,
   - l'*AtelierB* était capable d'exécuter 16 preuves supplémentaires automatiquement,
   - et seules 6 ont besoin de l'intervention de l'utilisateur.

4.2 Exemple 2, Canal de communication sécurisé :

Dans cet exemple on crée un modèle de canal de communication sécurisé. Le niveau le plus haut de cette conception est simplement une boite dont les valeurs des entrées sont équivalents à celles des sorties. Dans le deuxième niveau, on construit cette conception en utilisant un code supplémentaire correcteur « Hamming ». Le principe des codes correcteurs d'erreurs est de rajouter une information supplémentaire redondante de manière à détecter et éventuellement corriger de possibles erreurs de transmission. La forme la plus simple de détection d'erreur est l'adjonction au mot du message d'un bit de parité.

Par exemple pour un message comportant sept données binaires, on compte le nombre de bits égaux à un. Si ce nombre est pair, le bit de parité rajouté vaudra 0, si ce nombre est impair le bit de parité vaudra 1. De cette façon le message émis de longueur huit aura toujours un nombre de bits égaux à un qui sera pair (parité égale à zéro). Si le message reçu a une parité égale à zéro, on considérera que le message a été correctement transmis (dans le cas ou un bit seulement de message peut être affecté). Mais il peut y avoir deux erreurs de transmission, ce bit de parité ne permettra pas de détecter cette forme d'erreur. Si la parité du message est égale à un, on sait qu'il y a certainement une erreur de transmission. Mais il n'est pas possible de retrouver la donnée erronée.

Dans notre exemple on utilise le code de Hamming pour détecter et corriger « **une** » erreur de transmission de message qui contient des informations de longueur **quatre**. On utilise dans cet exemple un code [7,4,3]. Il prend donc en entrée des mots de quatre bits de données, et ajoute **trois** bits de contrôle pour donner des mots de **sept** bits. Plus précisément :

$$(u1, u2, u3, u4) \rightarrow (u1, u2, u3, u4, u5, u6, u7)$$

où :

$$u5 = u1 \oplus u2 \oplus u3$$

$$u6 = u2 \oplus u3 \oplus u4$$

$$u7 = u1 \oplus u2 \oplus u4$$

Les bits de donnée numéro 1, 3 et 4 apparaissent dans deux équations ; le bit de donnée numéro 2 dans trois ; et les bits de contrôle numéro 5, 6 et 7 dans une seule. On peut alors vérifier que l'on a bien la taille de code supplémentaire « d = 3 » : en modifiant un bit de donnée pendant la transmission dans le canal de communication, intervenant dans deux ou trois équations quand on recalcule en se basant sur les nouvelles valeurs reçues.

En supposant que le canal de communication peut affecter **un bit au maximum du message transmis**, on est confronté à plusieurs cas :

- Aucune équation n'est fausse : pas d'erreur ;
- Une seule équation est fausse : Il s'agit alors forcément d'un bit de contrôle, car seuls les bits de contrôle n'interviennent que dans une seule équation. Les bits de données sont corrects ;
- Deux équations sont fausses : Il s'agit alors forcément d'un bit de donnée (1, 3 ou 4), car seules ces bits de données apparaissent dans deux équations. C'est donc le bit de donnée commun aux deux équations fausses qui est erroné ;
- Trois équations fausses : Il s'agit alors forcément du bit de donnée 2, car seul ce bit de donnée apparaît dans les trois équations, c'est lui qui est erroné.

4.2.1 La conception en VGUI

Le niveau le plus haut d'un canal sécurisé de communication est considéré, dans notre présentation, comme étant une boite avec quatre entrées et quatre sorties principales :

$$y1 = x1 ;$$

$$y2 = x2 ;$$

$$y3 = x3 ;$$

$$y4 = x4$$

```
ENTITY  corrcanal  IS
  PORT(  finderr, gout4, gout3, gout2, gout1    : OUT bit;
     gin4, gin3, gin2, gin1 : IN bit
      );
END corrcanal ;
```

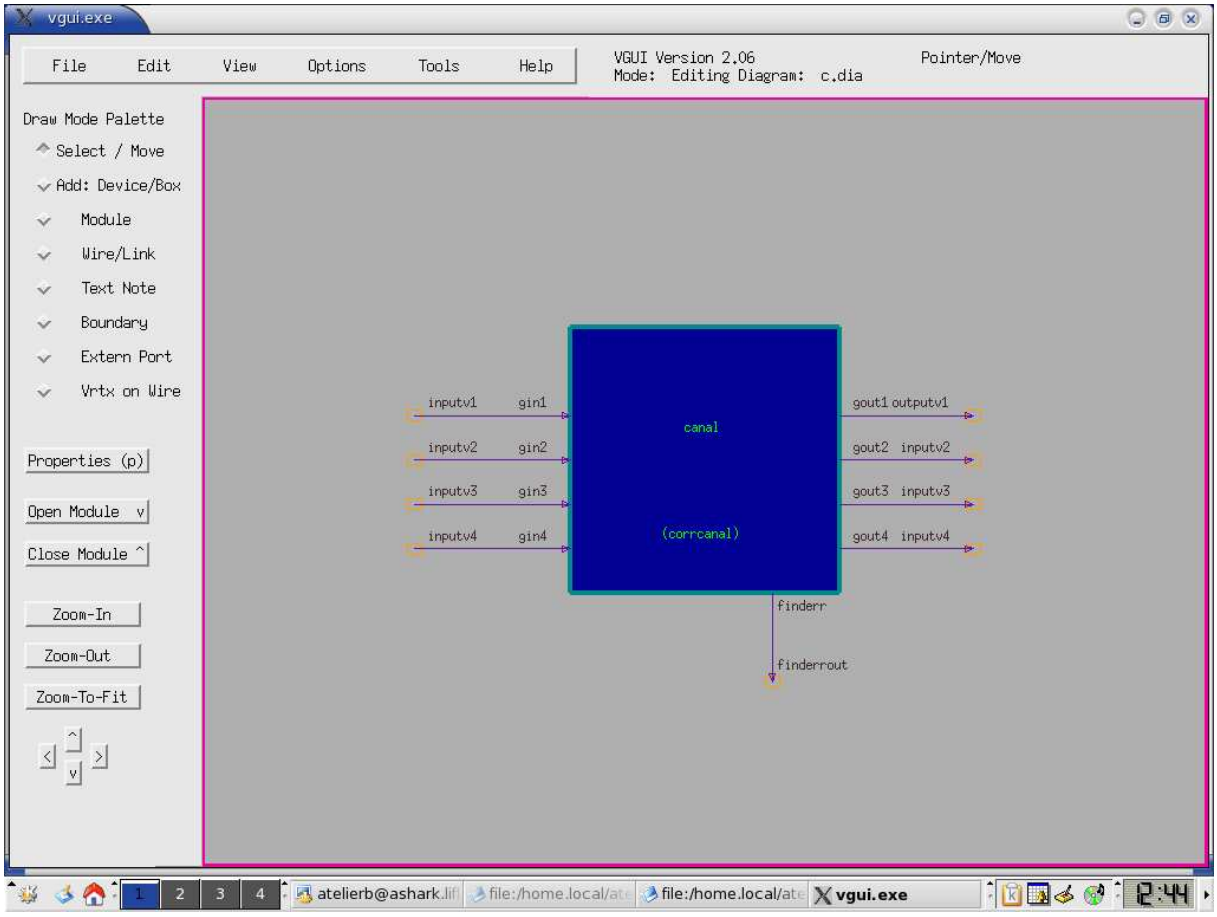

FIG. 4.7 – Hamming - (top-level).

On a ajouté une sortie pour indiquer si une erreur est détectée est corrigée pendant l'analyse ou pas. Le tableau noir de cette boite contient trois modules :

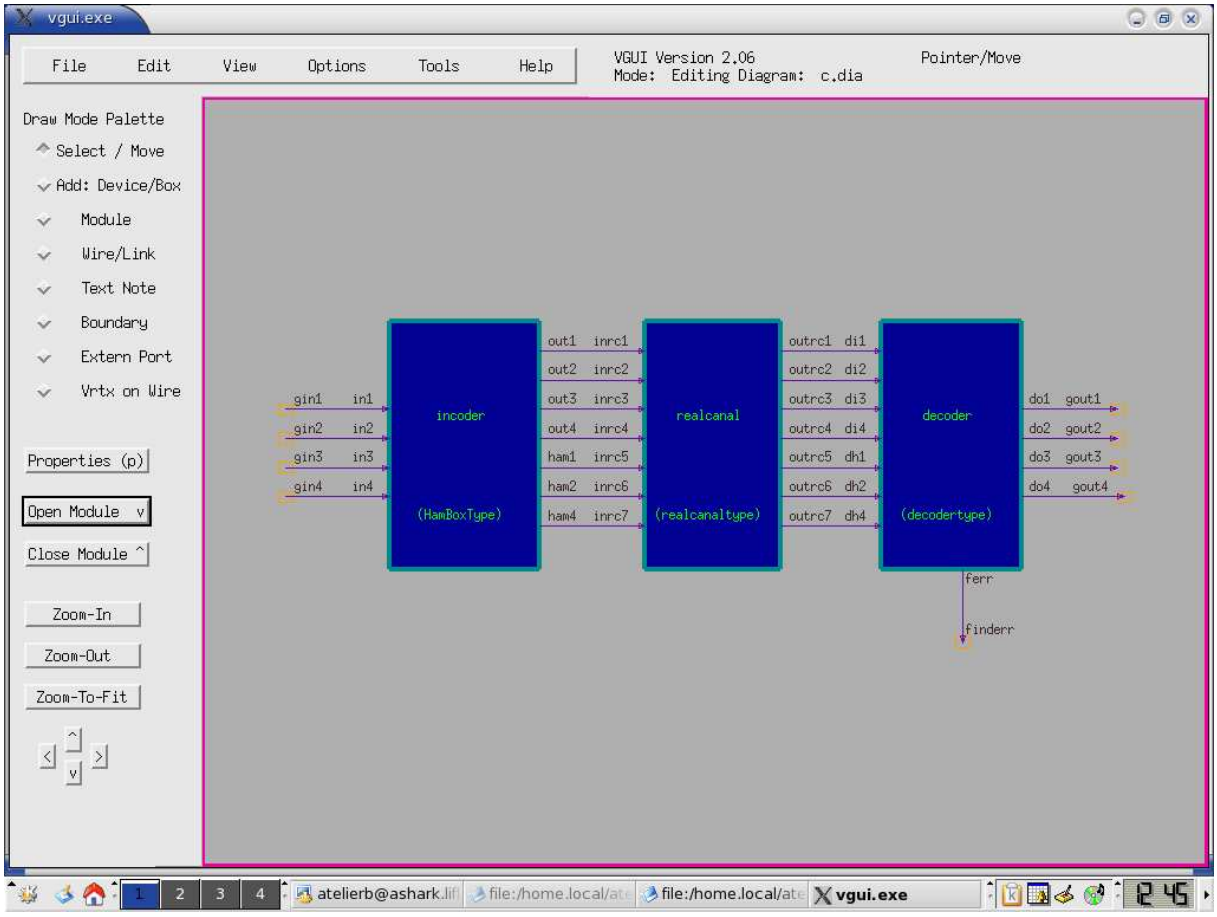

FIG. 4.8 – Hamming - structure.

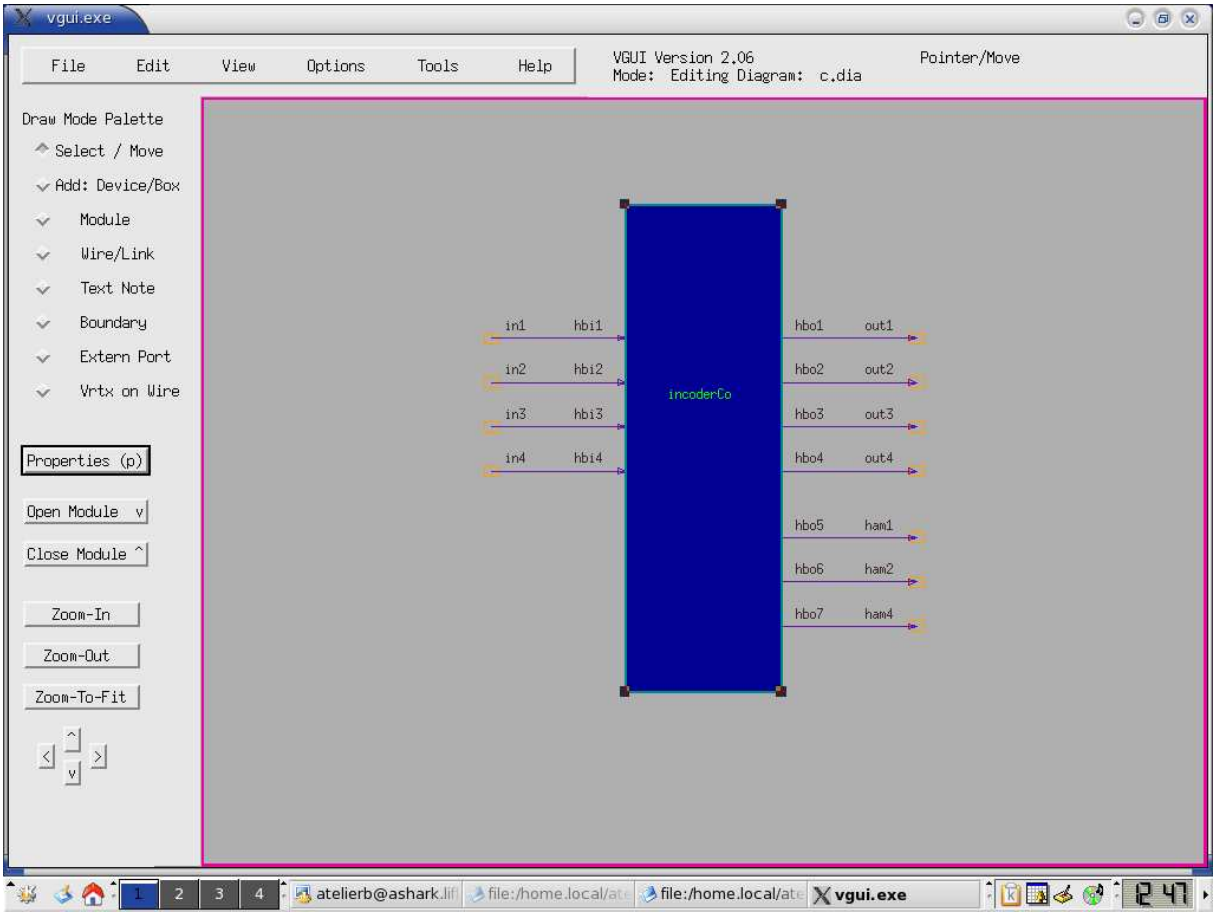

FIG. 4.9 – Hamming - Codeur.

1. **Le codeur de Hamming**
   Les entrées de cette boite sont les quatre entrées principales du canal sécurisé du niveau plus haut. Son but est de générer les trois bits supplémentaires du code Hamming :

```
y1=x1 &
y2=x2 &
y3=x3 &
y4=x4 &
h1=xor(x1, xor(x2,x4)) &
h2=xor(x1, xor(x3,x4)) &
h4=xor(x2, xor(x3,x4))
```

```
ENTITY  decodertype  IS
```

```
  PORT(  ferr, do4, do3, do2, do1   : OUT bit;
     dh4, dh2, dh1, di4, di3, di2, di1  : IN bit
      );
END decodertype;
```

2. **Le canal réel**
   Une boite qui représente le canal réel de communication dont les entrées sont reçues de première boite. Donc il a sept entrées et sept sorties. Cette boite peut générer une erreur au maximum. Autrement dit, elle transmet correctement six bits au minimum. La spécification fonctionnelle de cette boite est :

```
    (o1=i1 & o2=i2 & o3=i3 & o4=i4 & o5=i5 & o6=i6)
or  (o1=i1 & o2=i2 & o3=i3 & o4=i4 & o5=i5 & o7=i7)
or  (o1=i1 & o2=i2 & o3=i3 & o4=i4 & o6=i6 & o7=i7)
or  (o1=i1 & o2=i2 & o3=i3 & o5=i5 & o6=i6 & o7=i7)
or  (o1=i1 & o2=i2 & o4=i4 & o5=i5 & o6=i6 & o7=i7)
or  (o1=i1 & o3=i3 & o4=i4 & o5=i5 & o6=i6 & o7=i7)
or  (o2=i2 & o3=i3 & o4=i4 & o5=i5 & o6=i6 & o7=i7)
```

```
ENTITY  realcanaltype  IS
  PORT(  outrc7,
         outrc6,
         outrc5,
         outrc4,
         outrc3,
         outrc2,
         outrc1
                 : OUT bit;

          inrc7,
          inrc6,
          inrc5,
          inrc4,
          inrc3,
          inrc2,
          inrc1
                 :  IN bit
      );
```

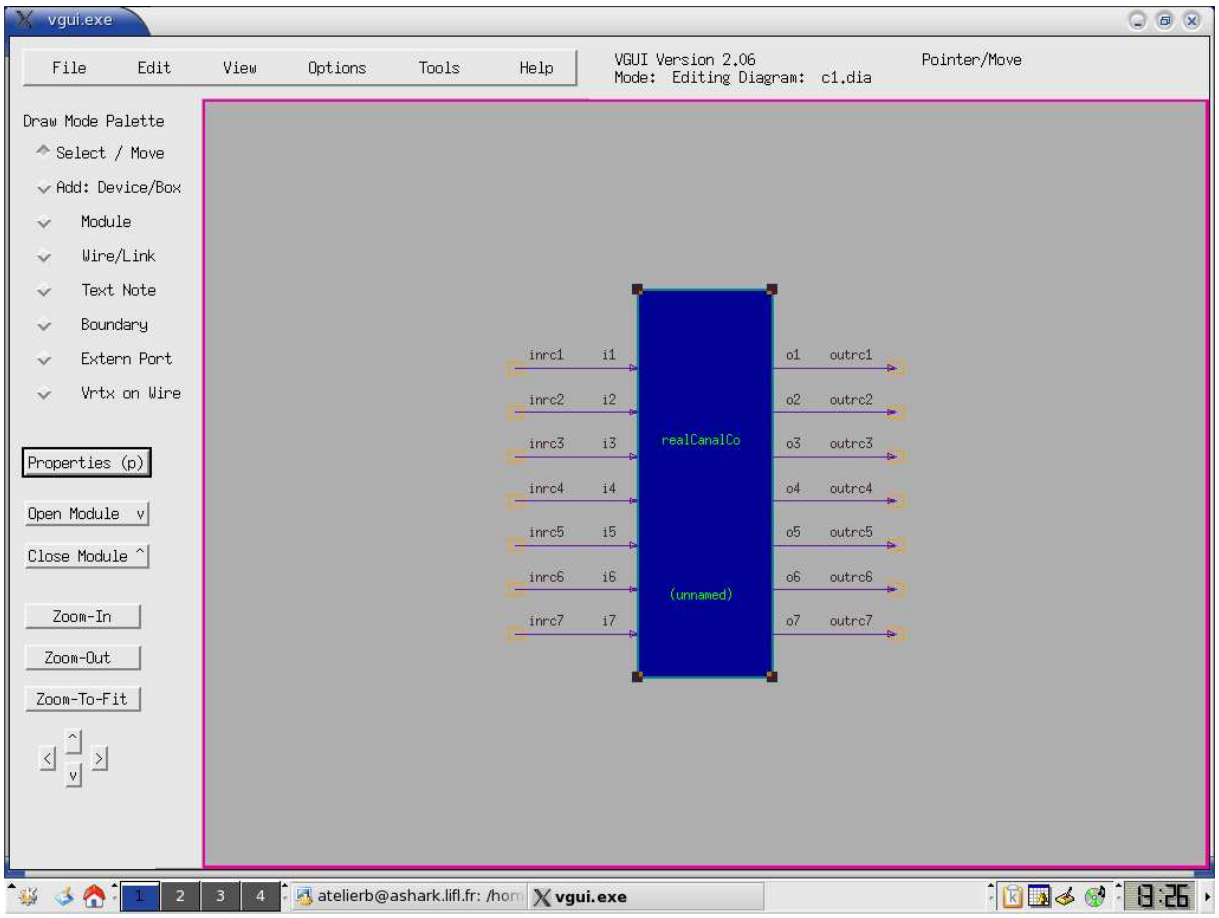

FIG. 4.10 – Hamming - canal réel.

3. **Le décodeur**
   La troisième boite a pour entrée les sorties du canal réel. Elle analyse les bits d'information et les bits de contrôle afin de :
   (a) détecter une erreur,
   (b) identifier le bit erroné place,
   (c) le corriger.
   On recalcule donc le code de Hamming à partir de valeurs reçues. Une erreur est détectée si le nouveau calcul ne correspond pas les valeurs de bits de contrôle reçues. La place de l'erreur, ainsi la correction, est précisée en analysant les équations fausses. Le bit qui apparaît dans toutes ces équations fausses, et uniquement dans ces équations, est le bit recherché. La fonction suivante est la principale dans le décodeur. La fonction `bool` en B est utilisé seulement pour calculer la valeur d'une expression logique.

```
fe=bool(     (h1=nott(xor(i1,xor(i2,i4))))
         or  (h2=nott(xor(i1,xor(i3,i4))))
         or  (h4=nott(xor(i2,xor(i3,i4))))
     )
&  o1=xor(i1 , (bool(     (h1=nott(xor(i1,xor(i2,i4))))
                & (h2=nott(xor(i1,xor(i3,i4))))
                & (h4=xor(i2,xor(i3,i4))) )))

&  o2=xor(i2 , (bool(     (h1=nott(xor(i1,xor(i2,i4))))
                        & (h2=xor(i1,xor(i3,i4)))
                        & (h4=nott(xor(i2,xor(i3,i4)))) )))

&  o3=xor(i3 , (bool(     (h1=xor(i1,xor(i2,i4)))
                        & (h2=nott(xor(i1,xor(i3,i4))))
                        & (h4=nott(xor(i2,xor(i3,i4)))) )))

&  o4=xor(i4 , (bool(     (h1=nott(xor(i1,xor(i2,i4))))
                        & (h2=nott(xor(i1,xor(i3,i4))))
                        & (h4=nott(xor(i2,xor(i3,i4)))) )))
 )
```

```
ENTITY  decodertype  IS
  PORT(  ferr, do4, do3, do2, do1   : OUT bit;
     dh4, dh2, dh1, di4, di3, di2, di1  : IN bit
      );
END decodertype;
```

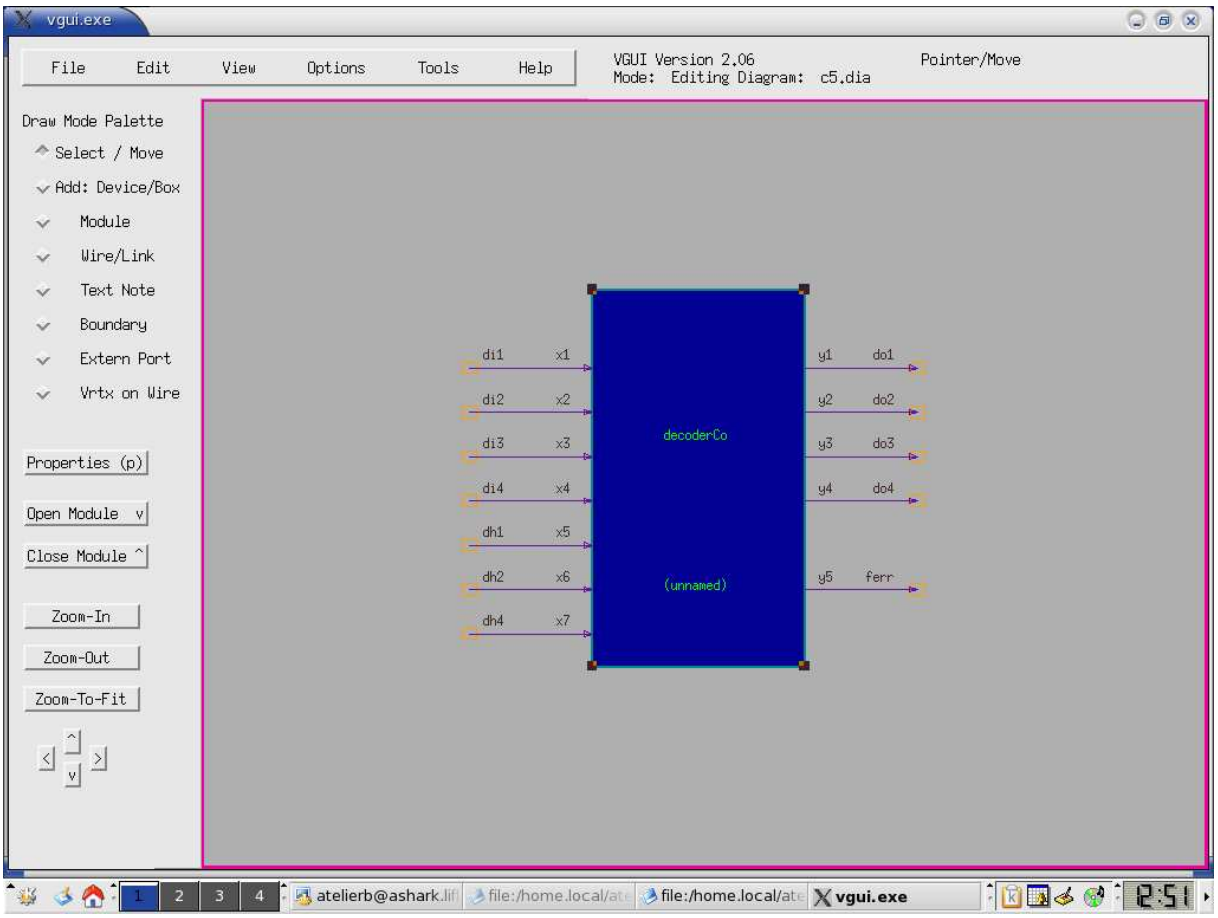

FIG. 4.11 – Hamming - Décodeur.

4.2.2 Le code B produit par l'outil BHDL

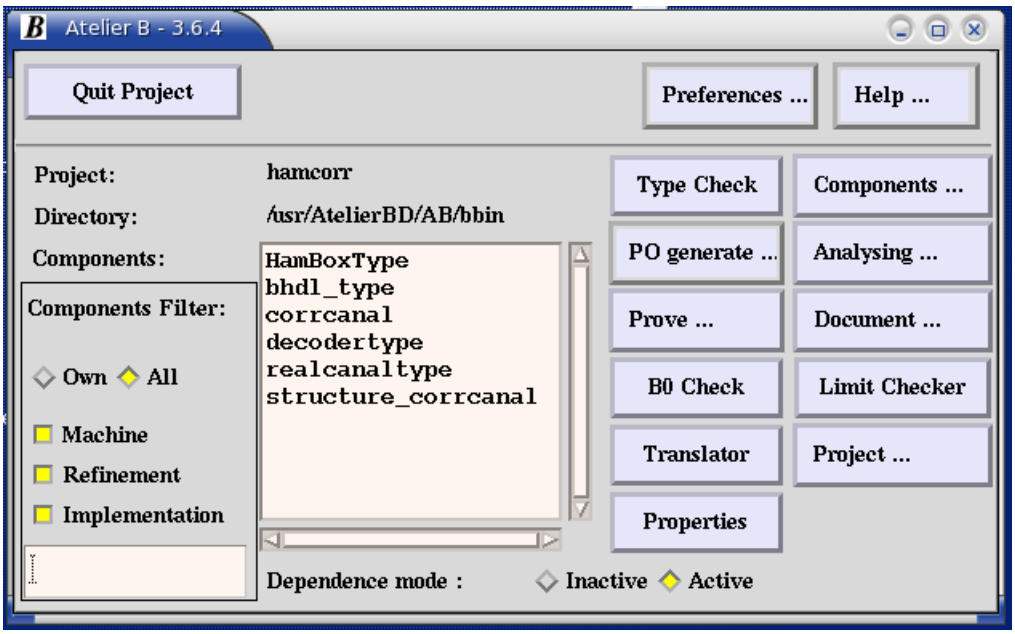

FIG. 4.12 – Projet Hamming dans l'AtelierB.

1. **l'architecture « corrcanal » en BHDL**. Voir l'entité corrcanal, page 152 :

```
ARCHITECTURE structure OF corrcanal IS
  SIGNAL  L30, L31, L32, L33, L34, L35, L36, L37, L38, L39,
      L40, L41, L42, L43  : bit;
  COMPONENT HamBoxType
    PORT(    ham4, ham2, ham1, out4, out3, out2, out1   : OUT bit;
         in4, in3, in2, in1 : IN bit
        ); END COMPONENT;
  COMPONENT realcanaltype
    PORT(    outrc7, outrc6, outrc5, outrc4, outrc3, outrc2, outrc1
       : OUT bit;
         inrc7, inrc6, inrc5, inrc4, inrc3, inrc2, inrc1
      : IN bit
        ); END COMPONENT;
  COMPONENT decodertype
    PORT(    ferr, do4, do3, do2, do1   : OUT bit;
         dh4, dh2, dh1, di4, di3, di2, di1  : IN bit
        ); END COMPONENT;
BEGIN
```

```
incoder : HamBoxType
  PORT MAP( L36, L35, L34, L33, L30, L32,
            L31, gin4, gin3, gin2, gin1
          );
realcanal : realcanaltype
  -- Purpose:
  -- #d (i1,i2,i3,i4,i5,i6,i7,o1,o2,o3,o4,o5,o6,o7)
  -- #= ==( (o1=i1 & o2=i2 & o3=i3
  -- #=            & o4=i4 & o5=i5 & o6=i6)
  -- #= or  (o1=i1 & o2=i2 & o3=i3
  -- #=            & o4=i4 & o5=i5 & o7=i7)
  -- #= or  (o1=i1 & o2=i2 & o3=i3
  -- #=            & o4=i4 & o6=i6 & o7=i7)
  -- #= or  (o1=i1 & o2=i2 & o3=i3
  -- #=            & o5=i5 & o6=i6 & o7=i7)
  -- #= or  (o1=i1 & o2=i2 & o4=i4
  -- #=            & o5=i5 & o6=i6 & o7=i7)
  -- #= or  (o1=i1 & o3=i3 & o4=i4
  -- #=            & o5=i5 & o6=i6 & o7=i7)
  -- #= or  (o2=i2 & o3=i3 & o4=i4
  -- #=            & o5=i5 & o6=i6 & o7=i7) )
  -- #i (inrc1,inrc2,inrc3,inrc4,inrc5,
        inrc6,inrc7,outrc1,outrc2,outrc3,
        outrc4,outrc5,outrc6,outrc7)
  PORT MAP( L43, L42, L41, L40, L39, L38,
            L37, L36, L35, L34, L33, L30, L32, L31
          );
decoder : decodertype
  -- Purpose:
  -- #d ( i1 , i2 , i3 , i4 , h1
        , h2 , h4 , o1 , o2 , o3
        , o4 , fe )
  -- #= ==( o1: BOOL & o2:BOOL & o3:BOOL & o4:BOOL
  -- #= &  i1: BOOL & i2:BOOL & i3:BOOL & i4:BOOL
  -- #= &  h1: BOOL & h2:BOOL & h4:BOOL
  -- #= &  fe: BOOL
  -- #= &  fe=(    (h1=not(xor(o1,xor(o2,o4))))
  -- #=         or (h2=not(xor(o1,xor(o3,o4))))
  -- #=         or (h4=not(xor(o2,xor(o3,o4))) ))

  -- #= &    o1=xor(i1,(h1=not(xor(o1,xor(o2,o4)))))
```

```
    -- #=      or (h2=not(xor(o1,xor(o3,o4))))
    -- #=      or (h4=(xor(o2,xor(o3,o4)))))

    -- #= &    o2=xor(i1,(h1=not(xor(o1,xor(o2,o4)))))
    -- #=      or (h2=(xor(o1,xor(o3,o4))))
    -- #=      or (h4=not(xor(o2,xor(o3,o4)))))

    -- #= &    o3=xor(i1,(h1=(xor(o1,xor(o2,o4)))))
    -- #=      or (h2=not(xor(o1,xor(o3,o4))))
    -- #=      or (h4=not(xor(o2,xor(o3,o4)))))

    -- #= &    o4=xor(i1,(h1=not((o1,xor(o2,o4)))))
    -- #=      or (h2=not(xor(o1,xor(o3,o4))))
    -- #=      or (h4=not(xor(o2,xor(o3,o4)))))
    -- #= & )

-- #i (di1,di2,di3,di4,dh1,dh2,dh4,do1,do2,do3,do4,ferr)
    PORT MAP( finderr, gout4, gout3, gout2,
              gout1, L43, L42, L41, L40,
              L39, L38, L37
            );
END structure;
```

2. **La machine *decodertype***. Voir le composant *decodertype* et son instantiation et ses propriétés dans l'architecture précédente :

```
MACHINE
    decodertype

SEES
       bhdl_type

VARIABLES
       ferr
     , do4
     , do3
     , do2
     , do1
     , dh4
     , dh2
     , dh1
     , di4
     , di3
     , di2
     , di1


DEFINITIONS
    "bhdl_type.def";
    "decodertype.def"


INVARIANT
    inv_decodertype ( ferr, do4, do3, do2,
                      do1, dh4, dh2, dh1,
                      di4, di3, di2, di1
                    )


INITIALISATION
     ferr, do4, do3, do2, do1, dh4, dh2, dh1, di4,
     di3, di2, di1
     :( inv_decodertype ( ferr, do4, do3, do2, do1,
                          dh4, dh2, dh1, di4, di3,
```

```
                              di2, di1
                            )
        )

/* OPERATIONS
    type_decodertype =
    BEGIN
          ferr
         ,do4
         ,do3
         ,do2
         ,do1
         ,dh4
         ,dh2
         ,dh1
         ,di4
         ,di3
         ,di2
         ,di1
          :( inv_decodertype ( ferr, do4, do3, do2, do1, dh4, dh2,
                               dh1, di4, di3, di2, di1
                             )
          )
    END

*/

END
```

3. **La machine corrcanal**. Voir l'entité corrcanal, page 152 :

```
MACHINE
    corrcanal

SEES
       bhdl_type


VARIABLES
       finderr
     , gout4
     , gout3
     , gout2
     , gout1
     , gin4
     , gin3
     , gin2
     , gin1


DEFINITIONS
    "bhdl_type.def";
    "corrcanal.def"


INVARIANT
    inv_corrcanal

INITIALISATION
     finderr, gout4, gout3, gout2, gout1,
     gin4, gin3, gin2, gin1:( inv_corrcanal)

/* OPERATIONS
    type_corrcanal =
    BEGIN
          finderr
         ,gout4
         ,gout3
         ,gout2
         ,gout1
```

```
            ,gin4
            ,gin3
            ,gin2
            ,gin1
             :( inv_corrcanal)
        END
    */

    END
```

4. **La machine HamBoxType**. Voir le composant *HamBoxType* dans l'architecture *corrcanal*, page 161 :

```
MACHINE
    HamBoxType

SEES
       bhdl_type


VARIABLES
       ham4
     , ham2
     , ham1
     , out4
     , out3
     , out2
     , out1
     , in4
     , in3
     , in2
     , in1


DEFINITIONS
    "bhdl_type.def";
    "HamBoxType.def"


INVARIANT
    inv_HamBoxType ( ham4, ham2, ham1, out4, out3, out2,
                             out1, in4, in3, in2, in1
                   )


INITIALISATION
     ham4, ham2, ham1, out4, out3, out2, out1,
     in4, in3, in2, in1
     :( inv_HamBoxType ( ham4, ham2, ham1, out4, out3,
                         out2, out1, in4, in3, in2, in1
                         )
```

```
)

/* OPERATIONS
    type_HamBoxType =
    BEGIN
          ham4
         ,ham2
         ,ham1
         ,out4
         ,out3
         ,out2
         ,out1
         ,in4
         ,in3
         ,in2
         ,in1
          :( inv_HamBoxType ( ham4, ham2, ham1, out4, out3,
                              out2, out1, in4, in3, in2, in1
                              )
)
    END

*/

END
```

5. **Le raffinement structure_corrcanal**. Voir l'architecture *corrcanal*, page 161 :

```
REFINEMENT
    structure_corrcanal

REFINES
    corrcanal

SEES
    bhdl_type

INCLUDES
    incoder.HamBoxType,
    realcanal.realcanaltype,
    decoder.decodertype

VARIABLES
    L36, L35, L34, L33, L30, L32, L31, gin4, gin3, gin2,
    gin1, L43, L42, L41, L40, L39, L38, L37, finderr,
    gout4, gout3, gout2, gout1

DEFINITIONS
     "bhdl_type.def" ;
    inv_structure_corrcanal ==
       (
       inv_HamBoxType( L36, L35, L34, L33, L30, L32,
                           L31, gin4, gin3, gin2, gin1
                     )
       & inv_realcanaltype( L43, L42, L41, L40, L39, L38,
                   L37, L36, L35, L34, L33, L30, L32, L31
                    )
       & inv_decodertype( finderr, gout4, gout3, gout2,
                        gout1, L43, L42, L41, L40, L39,
                        L38, L37
                       )
        )

;   "HamBoxType.def" ;
   "realcanaltype.def" ;
   "decodertype.def"
```

```
INVARIANT
     inv_structure_corrcanal

INITIALISATION
     L36, L35, L34, L33, L30, L32, L31, gin4, gin3, gin2,
      gin1, L43, L42, L41, L40, L39, L38, L37, finderr,
      gout4, gout3, gout2, gout1
                 : ( inv_structure_corrcanal )

/* OPERATIONS
    type_corrcanal =
    BEGIN
L36, L35, L34, L33, L30, L32, L31, gin4, gin3, gin2, gin1, L43,
L42, L41, L40, L39, L38, L37, finderr, gout4, gout3, gout2, gout1
: ( inv_structure_corrcanal )
    END

    */

END
```

6. **La machine realcanaltype**. Voir le composant *realcanaltype* dans l'architecture *corrcanal*, page 161 :

```
MACHINE
    realcanaltype

SEES
       bhdl_type


VARIABLES
       outrc7
     , outrc6
     , outrc5
     , outrc4
     , outrc3
     , outrc2
     , outrc1
     , inrc7
     , inrc6
     , inrc5
     , inrc4
     , inrc3
     , inrc2
     , inrc1


DEFINITIONS
    "bhdl_type.def";
    "realcanaltype.def"


INVARIANT
    inv_realcanaltype ( outrc7, outrc6, outrc5, outrc4,
     outrc3, outrc2, outrc1, inrc7, inrc6, inrc5, inrc4,
      inrc3, inrc2, inrc1
                     )


INITIALISATION
     outrc7, outrc6, outrc5, outrc4, outrc3, outrc2,
```

```
      outrc1, inrc7, inrc6, inrc5, inrc4, inrc3, inrc2,
      inrc1:( inv_realcanaltype ( outrc7, outrc6, outrc5,
                                  outrc4, outrc3, outrc2,
                                  outrc1, inrc7, inrc6,
                                  inrc5, inrc4, inrc3,
                                  inrc2, inrc1 )
            )

/* OPERATIONS
    type_realcanaltype =
    BEGIN
          outrc7
         ,outrc6
         ,outrc5
         ,outrc4
         ,outrc3
         ,outrc2
         ,outrc1
         ,inrc7
         ,inrc6
         ,inrc5
         ,inrc4
         ,inrc3
         ,inrc2
         ,inrc1
          :( inv_realcanaltype ( outrc7, outrc6, outrc5, outrc4,
                                 outrc3, outrc2, outrc1, inrc7,
                                 inrc6, inrc5, inrc4, inrc3,
                                 inrc2, inrc1
                                 )
)

    END
*/

END
```

Chapitre 5

Perspectives

Ce chapitre présente plusieurs axes pour la continuation de ce travail.

5.1 Enrichir la traduction de *VHDL* vers *B* :

Le parseur dans ce travail est capable d'analyser un programme VHDL dans sa totalité. La grammaire qui a été implémentée dans ce projet est la dernière version *VHDL-AMS*. Le treewalker, par contre, ne permet de traduire qu'une partie du programme VHDL en *B*.

Du coté VHDL, on utilise dans notre traduction très exactement le sous-ensemble de la grammaire suffisante pour analyser le code VHDL produit par *VGUI*. Ce sous-ensemble contient une partie de l'entité et une partie de l'architecture. De plus, la version actuelle de notre traducteur est enrichie par la traduction manuelle des packages de VHDL. Le logiciel utilisé, *ANTLR*, permet l'ajout facile de nouvelles règles de traduction de VHDL vers *B*. Ces ajouts se font de façon déclarative sous la forme de règles de réécriture d'arbres. Ceci permet une extension facile du noyau actuel, mais aussi une meilleure confiance dans le code *B* produit puisque la correction de la traduction totale est acquise à partir de la correction de chaque règle de traduction. Du coté *B*, on n'utilise qu'une partie de composants de ce langage. Ces composants sont la machine abstraite et le raffinement. Pour rendre ce projet plus efficace, il est préférable de trouver des traductions pour tous les composants VHDL en *B*. Par exemple, on pourrait trouver des correspondances entre la configuration en VHDL et l'implémentation en *B*.

5.2 Tolérance aux pannes

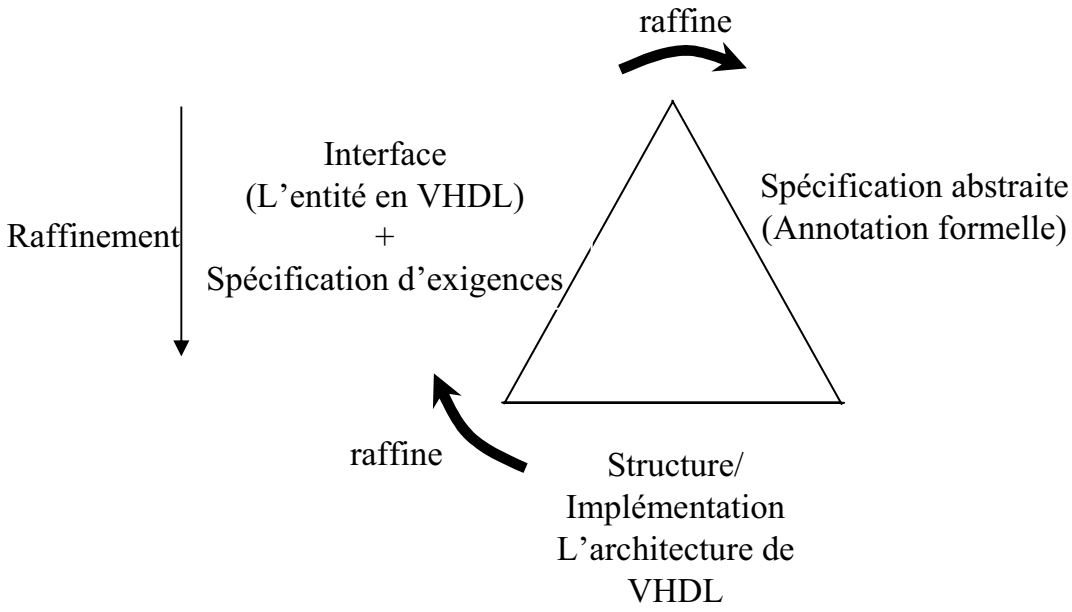

FIG. 5.1 – Modèle de conception à trois facettes

La description d'un projet aussi bien en *B* qu'en VHDL repose sur deux niveaux d'abstraction :

En VHDL, la vue externe est représentée par l'entité qui contient tous les détails sur l'interface du circuit ; alors que l'architecture décrit la structure interne du circuit, ce qui permet de donner une image complète sur le comportement ou la composition du circuit.

En *B*, les deux composants extrêmes dans la conception sont la machine abstraite et l'implémentation. La première déclare la description la plus faible de l'objectif, et la seconde met en exergue une description plus concrète. Les étapes de raffinement entre les deux extrêmes de *B* garantissent l'exactitude de conception du projet.

En anticipant sur *B*, HDL et *ADL*, trois facettes du projet ressortent. Elles peuvent orienter le développement de conception et faciliter la vérification :

1. La spécification abstraite, fonctionnelle et formelle de projet qui contient toutes les exigences et les propriétés de projet attendues.

2. les propriétés de sûreté, qui peuvent être considérées comme étant la spécification la plus abstraite (ou la plus faible du point de vue logique) dans la conception du projet. Elles sont utilisées généralement pour vérifier la conception finale. Ce sont celles aussi que l'on pourrait vérifier dynamiquement.

3. L'implémentation, ou l'architecture, qui contient la description déterminante du projet.

Les relations d'implication notées « raffine » doivent être respectées.
Par exemple, dans un système de contrôle de feux multicolores d'un carrefour, la propriété à respecter peut être la non-existence de deux feux verts simultanément.
Un autre exemple peut être la preuve par neuf. Cette technique est employé pour vérifier que le résultat de la multiplication de deux entiers x et y est correcte : Supposant que X, Y et Z soient les sommes des termes des chiffres des nombres x, y et z respectivement. La règle utilisée pour vérifier la multiplication est comme suit :

$$Prop : (Z mod 9) = (((X mod 9) * (Y mod 9)) mod 9)$$

L'exemple du chapitre précédent « code Hamming » peut être présenté dans ce diagramme aussi.

5.3 Gestion du temps :

La notion de temps intervient au niveau de la conception des portes en HDL pour vérifier la synchronisation entre les composants ainsi que leur latence. En *ADL*, le paramètre temps peut être important dans l'analyse des structures, et plus particulièrement en présence de boucles.

Dans notre projet, plusieurs techniques sont envisagées pour représenter le temps.

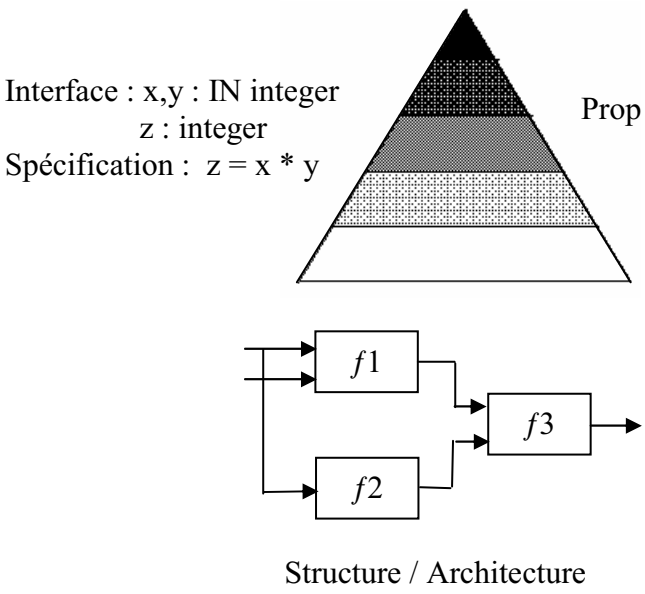

FIG. 5.2 – Exemple - Preuve par neuf

Une des techniques permettant de représenter le temps global est la création d'un type « signal ». Un signal est représenté par une chaîne de couples (valeur, date). La valeur d'un signal à un moment donné serait la valeur du premier champ du couple correspondant à l'instant considéré. Une horloge globale synchronisant les signaux est nécessaire pour ce faire. Cette méthode est utilisée par J.L.Boulanger et G. Mariano. On a utilisé cette technique dans la conception du projet HighWay, qui s'intéresse au contrôle des feux d'accès à la voie rapide. Bien que très peu d'informations soient disponibles sur ce travail, nous avons découvert dans la dernière version de la *B-Toolkit* qu'un embryon d'un tel développement avait été réalisé dans le cadre d'un programme de recherche de l'armée britannique.

Une autre technique pour représenter la conception asynchrone consiste à utiliser des types enrichis par une valeur supplémentaire « unknown » : On ajoute à tous les types utilisés dans le projet une valeur nommée *unknown* qui indique qu'aucune valeur n'a été affectée à la variable considérée dans un composant de projet. Au départ, toutes les entrées/sorties sont initialisées à la valeur *unknown*. Un composant ne fait rien tant qu'une de ses entrées contient la valeur *unknown*. Dès que toutes les entrées du composant sont **réellement** affectées, celui-ci exécute sa tâche. Les résultats de

l'opération obtenus sont alors affectés aux sorties. Cette valeur *unknown* est d'ailleurs incluse dans la logique à neuf valeurs utilisée dans la bibliothèque *std_logic_1164*. L'inconvénient majeur de cette approche est la nécessité d'un « opérateur de point fixe » sur chaque composant pour la mise à jour des sorties. En concordance avec les entrées des composants suivants.

Ceci revient à décrire un simulateur soit manuellement soit en utilisant le moteur *B événementiel*. Il est à craindre que les obligations de preuve soient beaucoup plus complexes et nombreuses que dans la solution que nous avons développé à savoir la seule modélisation combinatoire. Ceci est à rapprocher du travail réalisé dans la cadre du projet européen *PUSSEE* autour d'une traduction des principaux de concepts *VHDL* en machine abstraite *B*. L'objectif ici est de favoriser la migration pour les électroniciens concepteurs de circuits électroniques de *VHDL* vers *B*, cadre plus riche du point de vue formel.

Ces méthodes restent à notre avis insuffisantes pour implémenter correctement les aspects du temps de *VHDL* surtout pour des composants électroniques d'une taille réaliste. L'utilisation des autres notations comme la logique temporelle peut être utile dans ce cas là.

Il est à noter que la partie « spécification abstraite » décrite dans ce schéma pourrait être obtenue de manière plus ou moins automatique. Il existe en effet des outils d'analyse abstraite qui ont été développés pour le langage *VHDL*. Les travaux les plus présents ont été réalisés par Charles Hymans (thèse LIX, 9 Sept 2004).

5.4 Prouveur *B* ou autres ?

L'*AtelierB* est capable de prouver la consistance de chaque composant de *B*. Cet outil peut aussi prouver les relations entre ces composants : raffinement, inclusion, vision, etc., Mais la puissance du prouveur de l'*AtelierB* est limitée. Et son implémentation est totalement masquée.

On peut légitimement se poser la question de l'utilisation d'un autre théorème prouveur. C'est la raison pour laquelle. Jean Louis Boulanger a réalisé un parseur du code *B* en utilisant le compilateur *ANTLR*, ceci dans le but premier pour pouvoir exporter les obligations de preuve *B* dans un format « open source » utilisable dans un autre démonstrateur de théorèmes. Ce travail est

encore en cours de développement. : il permettra à terme d'exporter toutes les obligations de preuve de l'*AtelierB* vers *PHoX* [103].

PHoX[1] est un assistant-prouveur assez comparable à *COQ*. Il est basé sur la logique d'ordre supérieur et est « extensible" ». Un de points importants de *PHoX* est qu'il est aussi facile à utiliser comme possible et nécessite qu'un temps d'étude minimal. La version actuelle est encore expérimentale mais néanmoins utilisable. Des recherches à l'*INRETS*[2] sont orientées pour utiliser *PHoX* en lieu et place de l'*AtelierB*. L'utilisation d'un autre prouveur peut avoir plusieurs avantages.

- Il permet d'enrichir les propriétés logiques de composants par l'introduction d'autres expressions plus puissantes ; la méthode *B* n'acceptant que des propriétés du premier ordre.

- L'utilisation d'un autre démonstrateur ouvert (non fermé comme celui de l'*AtelierB*) permet, en plus, de connaître son fonctionnement, ce qui donne plus de confiance aux utilisateurs, car le code du démonstrateur de l'*AtelierB* lui-même n'est pas un open source.

- L' utilisation d'un nouveau démonstrateur nécessite de porter les expressions logiques vers le langage de ce démonstrateur, ce que nécessite à son tour l'existence de sémantique claire de *B*.

5.5 Sémantique *VHDL* et Sémantique *B*

Le choix des langages de spécification formelle pour le développement et la vérification de composants électroniques ou systèmes hybrides est avant tout argumenter dans le cadre de systèmes dits critiques, systèmes pour lesquels le développement zéro défaut est un horizon idéal. Si les langages *(V)HDLs* fournissent un moyen simple de description de composants adaptés à la simulation et la synthèse, plus précisément, ils sont suffisamment proches du logiciel pour la simulation et suffisamment proches du matériel pour la réalisation matérielle, ils ne sont malheureusement pas suffisamment proches d'un modèle mathématique pour en permettre une utilisation formelle. Le travail de traduction de *VHDL* vers *B* que nous avons réalisé ne doit pas faire oublier que nous n'avons pas ici apporté la preuve formelle de la correction de

[1] http ://www.lama.univ-savoie.fr/ÑRAFFALLI/phox.html
[2] Institut National de Recherche sur les Transports et leur Sécurité.

notre plateforme. Une telle correction n'est possible que si elle s'appuie sur la sémantique mathématique de *VHDL*. Une sémantique dénotationnelle de *VHDL* consiste à donner une signification à chaque structure de ce langage, y compris ses instructions. La dénotation d'une instruction consiste à préciser la fonction de transition qui exprime comment l'état est modifié quand on exécute cette instruction.

Il existe des travaux conséquents sur cette question : [22], [105], [91], [113], [116] .etc. Citons d'abord Félix Nicoli qui a décrit dans sa thèse [91], une sémantique dénotationnelle pour un sous-ensemble du langage *VHDL* et ceci pour simuler le comportement du circuit. Il est spécialement utilisé pour modéliser le temps et permettre de vérifier certaines propriétés dans la conception avant la production. Le démonstrateur utilisé et celui de *Boyer Moore (NQTHM)* qui repose sur une logique de premier ordre sans quantification avec égalité. Hormis les théorèmes prouvables dans cette logique, il est possible d'étendre la théorie par l'introduction de types de données inductifs et la définition de fonctions récursives. Cet outil de *Boyer Moore* est écrit en *Common Lisp*.

Citons aussi, dans [113], Darly John Stewart a proposé un langage formel, appelé *VV*, basé le langage V^3 Formal Equivalence Project) pour simuler statiquement et dynamiquement les langages *VHDL* et *Verilog*. Un programme *VV* est interprété comme une machine d'états finis. En utilisant SMV^4 basé sur la logique temporelle linéaire ; *LTL*, on peut vérifier formellement le comportement de langages *VHDL* et *Verilog*.

L'utilisation d'une sémantique générale *HDL* dans notre projet nous permettrait enrichir les premières de niveaux de conception dans la méthodologie qu'on a proposée en gardant plusieurs langages d'implémentation cible.

[3]résultat du *Verilog*.
[4]Symbolic Model Verifier.

Bibliographie

[1] Samar Abdi and Daniel Gajski. Formal verification of specification partitioning. Technical Report CECS-TR-03-06, Center for Embedded Computer Systems, UC Irvine, April 2003.

[2] J-R Abrial. Event b reference manual. In L. Lecomte, editor, *MATISSE : Methodologies and Technologies for Industrial Strength Systems Engineering*. ist :information socety technologoies, june 2001. IST-1999-11435.

[3] Jean-Raymond Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, 1996.

[4] Jean-Raymond Abrial. Le cahier des charges : Contenu, forme et analyse en vue de la formalisation. http ://www.atelierb.societe.com/ressources/articles/Cdc.pdf, Juin 1998.

[5] Jean-Raymond Abrial. Event driven electronic circuit construction, August 2001. http ://profs.sci.univr.it/ fontana/formalware/Abrial_4.ps.

[6] Jean-Raymond Abrial, Dominique Cansell, and Guy Laffite. "higher-order" mathematics in B. In *2nd International Conference on B and Z, (ZB2002)*, Saint-Martin d'Hères, France, January 2002.

[7] Jay K. Adams and Donald E. Thomas. The design of mixed hardware/software systems. In *Proceedings of the 33rd annual Conference on Design Automation*, pages 515–520. ACM Press, 1996.

[8] Roland Airiau, Jean-Michel Bergé, Vincent Olive, and Jacques Rouillard. *VHDL : Langage, modélisation, synthèse*. Presses polytechniques et universitaires romandes et CNET-ENST, 1998.

[9] Ammar Aljer, Jean-Louis Boulanger, and Georges Mariano. Formalization of digital circuits using the B method. In *Proceedings of CompRail VIII, Eighth International Conference on Computer Aided Design, Manufacture and Operation in the Railway and Other Advanced Mass*, Lemnos, Greece, June 2002.

[10] Ammar Aljer and Philippe Devienne. Co-design and refinement for safety-critical systems. In *Proceedings of the 19th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Cannes, France, October 2004.

[11] Ammar Aljer, Philippe Devienne, and Sophie Tison. Component based co-design and refinement. In *Proceedings of the IEEE International Conference on Information and Communication Technologies*, Damascus, Syria, April 2004.

[12] Ammar Aljer, Philippe Devienne, Sophie Tison, Jean-Louis Boulanger, and Georges Mariano. BHDL : Circuit design in B. In *Proceedings of IEEE (ACSD'03) Third International Conference on Application of Concurrency to System Design*, volume 1, Guimarães, Portugal, June 2003.

[13] Robert Allen and David Garlan. The Wright architectural specification language. Technical Report CMU-CS-96-TBD, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1996.

[14] Robert J. Allen, David Garlan, and James Ivers. Formal modeling and analysis of the HLA component integration standard. In *Proceedings of of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, Florida, November 1998. ACM.

[15] Guido Arnout. SystemC standard. In *Proceedings of Asia and South Pacific Design Automation Conference 2000 (ASP-DAC'00)*, pages 573–577, Yokohama, Japan, November 2000.

[16] B-Core Company, Harwell, UK. *B-Toolkit*, 1997. http ://www.b-core.com/.

[17] Peter Bellows and Brad Hutchings. JHDL - an HDL for reconfigurable systems. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, CA, April 1998. IEEE Computer Society Press.

[18] Victor Berman. Standard Verilog-VHDL interoperability. In *Verilog HDL Conference*, Santa Clara, CA, march 1994.

[19] Dominique Bied-Charreton. Introduction à la conception de circuits intégrés autotestables. Technical report, Institut de Recherche des Transports C.R.E.S.T.A., Septembre 1984.

[20] Dominique Bied-Charreton. Sécurité intrinséque et sécurité probabiliste dans les transports terrestre. Technical report, INRETS : Institut de National de recherche sur les transports et leur securité, Novembre 1998. ISBN 2-85782-510-2.

[21] Gerard M Blair. Verilog : accelerating digital design. *IEEE Electronics & Communication Engineering Journal*, 9 :68–72, April 1997.

[22] Dominique Borrione and Ashraf Salem. Denotational semantics of a synchronous VHDL subset. *Formal Methods System Design*, 7(1-2) :53–71, Augest 1995.

[23] Jean-Louis Boulanger, Georges Mariano, and Ammar Aljer. B-HDL an experiment to formalizing hardware by software formal specifications. In *Proceedings of EDCC-4 : 4th European Dependable Computing Conference*, Toulouse, France, October 2002.

[24] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch : a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pages 83–94, Vancouver, British Columbia, Canada, June 2000.

[25] Randal E. Bryant, Pankaj Chauhan, Edmund M. Clarke, and Amit Goel. A theory of consistency for modular synchronous systems. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer Aided Design (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 486–504, November 2000.

[26] Randal E. Bryant and James H. Kukula. Formal methods for functional verification. In A. Kuehlmann, editor, *The Best of ICCAD : 20 Years of Excellence in Computer-Aided Design*, pages 3–16, Carnegie Mellon University, 2003. Kluwer Academic Publishers. http ://www.cs.cmu.edu/ bryant/pubdir/iccad-best02.ps.

[27] J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE conference on Design automation*, pages 46–51. ACM Press, 1990.

[28] Michael Butler and Jerome Falampin. An approach to modelling and refining timing properties in B. In *Refinement of Critical Systems (RCS'02)*, Grenoble, France, January 2002.

[29] Michael Butler and Traian Muntean, editors. *RCS'02 : International Workshop on Refinement of Critical Systems*, Grenoble, France, January 2002. http ://www.matisse.qinetiq.com/workshop.htm.

[30] Embeddes electronic solutions : The heart of smart products for today and tomorrow. MOTOROLA, 2000. http ://www.motorola.com/mot/doc/0/281_MotDoc.pdf.

[31] Dominique Cansell. Modélisation incrémentale de systèmes par la preuve. In *11eme Rencontre INRIA-Industrie - "L'ingénierie du logiciel"*, INRIA - Rocquencourt, France, Janvier 2004.

[32] Dominique Cansell and Dominique Méry. Abstraction and refinement of features. In Stephen Gilmore and Mark Ryan, editors, *Language Constructs for Describing Features*, pages 65–84. Springer Verlag, 2000.

[33] Luca P. Carloni, Kenneth L. McMillan, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency-insensitive design. In *Proceedings of the International Conference on Computer-Aided Design,*. UC Berkeley, Cadence Design Laboratories, November 1999.

[34] F. Cave. *Event B to B translator User Manual*, june 2001. IST-1999-11435.

[35] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore SoCs. In *39th Design Automation Conference (DAC'02)*, volume 12. IEEE, June 2002.

[36] Daniel Chillet. De la description d'un système à la description des transistors en VHDL. Lannio, France.

[37] E. M. Clarke, J. R. Burch, O. Grumberg, D. E. Long, and K. L. McMillan. Automatic verification of sequential circuit designs. In *Mechanized reasoning and hardware design*, pages 105–120. Prentice-Hall, Inc., 1992.

[38] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of automated reasoning*, volume II, chapter 24, pages 1635–1790. Elsevier Science Publishers, 2001.

[39] Edmund M. Clarke and Jeannette M. Wing. Formal methods : State of the art and future directions. *ACM Computer Survey*, 28(4) :626–643, 1996. http ://www.cs.cmu.edu/afs/cs/usr/wing/www/mit/paper/paper.ps.

[40] ClearSY System Engineering, Aix en Provence 3, France. *AtelierB Animateur, Manuel Utilisateur*, 2003.

[41] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures : Views and Beyond*. Addison Wesley Professional, hardcover edition, septembre 2002.

[42] Doulos training courses. Church Hatch, 22 Market Place, Ringwood, Hampshire, BH24 1AW, UK. www.doulos.com/trainingframe.html.

[43] Rapport d'étude. Schémas électroniques en B. Technical Report C96-S-003, Steria, RATP, SNCF,INRETS, 1996.

[44] Jeremy Dick. Traçabilié exigences/spécifications : B et DOORS. In *Outils pour et autour de B*. INRETS-ESTAS : Institut de National de recherche sur les transports et leur securité, Octobre 2001.

[45] David Dill. Formal verification : Experiences and future prospects. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL'99)*, San Antonio, TX, January 1999. ACM Press.

[46] David L. Dill. What's between simulation and formal verification ? (extended abstract). In *Proceedings of the 35th annual Conference on Design Automation (DAC'98)*, pages 328–329, San Francisco, California, United States, June 1998.

[47] Rainer Dömer, Daniel D. Gajski, and Jianwen Zhu. Specification and design of embedded systems. *it+ti magazine*, June 1998. Oldenbourg Verlag, Germany.

[48] Mireille Ducassé. Introduction à la méthode B : Construction et vérification formelles de programmes. Rennes, France, 1999. Support de cours.

[49] Mireille Ducassé and Laurence Rozé. Revisiting the "Traffic lights" B case study. Technical Report PI-1424, AEE : Architecture Electronique Embarquée, IRISA, Rennes, France, November 2001.

[50] Emmanuel Durand, Anne-Marie Deplanche, and Yvon Trinquet. Languages de configuration. Technical Report IRCyN-LangArchi, AEE : Architecture Electronique Embarquée, Pittsburgh, PA, Mars 1999.

[51] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems : formal models, validation, and synthesis. In Giovanni De Micheli, Rolf Ernst, and Wayne Wolf, editors, *Readings in hardware/software co-design*, chapter Modeling, pages 86–107. Kluwer Academic Publishers, Norwell, MA, USA, June 2001.

[52] Duncan Elliott. Lab 6 : Verilog HDL, Febuary 2003. http ://feynman.ee.ualberta.ca/ẽlliott/ee552/labs/lab6/ Department of Electrical and Computer Engineering, University of Alberta.

[53] Jacky Estublier and Jean-Marie Favre. Component models and technology. In Ivica Crnkovic and Magnus Larsson, editors, *Building Reliable Component-Based Software Systems*, pages 57–86. Artech House Publishers, July 2002. ISBN 1-58053-327-2.

[54] Marc Frappier and Henri Habrias. Comparison of the software specification methods. In Marc Frappier and Henri Habrias, editors, *Software Specification Methods : An Overview Using a Case Study*. Springer, October 2000.

[55] David Garlan. Formal modeling and analysis of software architecture : Components, connectors, and events. In Marco Bernardo and Paola Inverardi, editors, *Formal Methods for Software Architectures*, pages 1–24. Springer-Verlag Berlin, September 2003.

[56] David Garlan, Bob Monroe, Drew Kompanek, and David Wile. Towards an ADL toolkit. http ://www-2.cs.cmu.edu/ acme/adltk/index.html.

[57] David Garlan, Robert Monroe, and David Wile. ACME : An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, Novembre 1997.

[58] David Garlan, Robert T. Monroe, and David Wile. ACME : Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[59] Indradeep Ghosh and Masahiro Fujita. Automatic test pattern generation for functional RTL circuits using assignment decision diagrams. In *Proceedings of the 37th Conference on Design Automation (DAC'00)*, pages 43–48, Los Angeles, California, United States, 2000. IEEE Computer Society Press.

[60] Mike Gordon. Specification and verification courses. The University of Cambridge Computer Laboratory, 2003. www.cl.cam.ac.uk/users/mjcg/Teaching/SpecVer1/SpecVer1.html.

[61] ALBE group. *The AcmeLib Programmer's Manual (version 1.0b)*, February 1999. http ://www-2.cs.cmu.edu/afs/cs/project/able/www/AcmeWeb/Javaual

[62] Arati Gupta. Formal hardware verification methods : A survey. *International journal Formal Methods in System Design International journal*, 1(2/3) :151–238, October 1992.

[63] Nicolas Halbwachs and Pascal Raymond. A tutorial of Lustre. www-verimag.imag.fr/ halbwach/PS/tutorial.ps, January 2002.

[64] Pieter Hartel, Michael Butler, Andrew Currie, Peter Henderson, Michael Leuschel, Andrew Martin, Adrian Smith, Ulrich Ultes-Nitsche, and Bob Walters. Questions and answers about ten formal methods. In *Proceeding of the 4th International Workshop on Formal Methods for Industrial Critical Systems*, pages 179–203, Trento, Italy, July 1999.

[65] Richard Harwell, Erik Aslaksen, Ivy Hooks, Roy Mengot, and Ken Ptack. What is a requirement ? In *Proceedings of the Third International Symposium of the NCOSE*, Toronto, Ontario, 1993.

[66] Yannick Hervé. *VHDL-AMS - Applications et enjeux industriels, Cours et exercices corrigés*. Dunod, 2002.

[67] C. Hymans. Checking safety properties of behavioral VHDL descriptions by abstract interpretation. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 444–460. Springer, 2002.

[68] C. Hymans. Modular analysis of circuit description language by abstract interpretation : Application to the automatic extraction of circuit shapes. In *Designing Correct Circuits (DCC'02)*. ETAPS 2002, April 2002.

[69] C. Hymans. Design and implementation of an abstract interpreter for VHDL. In *Proceeding of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*, Lecture Notes in Computer Science. Springer, 2003.

[70] Wilson Ifill, Ib Sorensen, and Steve Schneider. The use of B to specify, design and verify hardware. In *High integrity software*, pages 43–62. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[71] Jawahar Jain, Amit Narayan, Masahiro Fujita, and Alberto Sangiovanni-Vincentelli. Formal verification of combinational circuits. In *Proceedings of the 10th International Conference on VLSI Design*, Hyderabad, India, January 1997.

[72] Andrew Kompanek. *AcmeStudio User's Manual*, January 2004. http ://www.csl.sri.com/users/rar/AcmeStudioManual.ps.

[73] K.P. Lam. Hardware and software co-design : A case study on multidimensional moments production for high performance signal processing applications. In Marco Bernardo and Paola Inverardi, editors, *Formal Methods for Software Architectures*. Springer-Verlag Berlin, Septembre 2003.

[74] Axel Van Lamsweede. From system goals to software architecture. In Marco Bernardo and Paola Inverardi, editors, *HiPC : International Conference on High Performence Computing*, Hayderabad, India, December 2001.

[75] Emmanuel Letier and Axel van Lamsweerde. Deriving operational software specifications from system goals. *SIGSOFT Software Engineering Notes*, 27(6) :119–128, 2002.

[76] Markus Lindgren. Summary of "a classification and comparison framework for software architecture description languages". www.idt.mdh.se/ msd/save/ ccourse/summaries/ADL_comparison.pdf, April 2003.

[77] Georges Mariano, Ammar Aljer, and Jean-Louis Boulanger. Conception sûre de circuit basée sur la notion de propriété. In *14èmes Journées Internationales Génie Logiciel & Ingénierie De Systèmes et leurs Applications*, CNAM, Paris, France,, Décembre 2001.

[78] Georges Mariano and Jean-Louis Boulanger. Modélisation formelle de circuits numériques par la méthode B. Technical Report ESTAS-RT1999-25, INRETS-ESTAS, 20 rue Elisee Reclus, 59650 Villeneuve d'Ascq, France, décembre 1999.

[79] Michael C. McFarland. Formal verification of sequential hardware : A tutorial. *IEEE transactions on Computer-Aided Design of Integrated Circuits Systems*, 12(5) :633–54, May 1993.

[80] Nenad Medvidovic and Richard N. Taylor. A multi-level approach to low-power IC design. *IEEE Spectrum*, 35(2), February 1998.

[81] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions Software Engineering*, 26(1) :70–93, 2000.

[82] Austin Melton, editor. *Software Measurement*. International Thomson Computer Press, 1996.

[83] Stephan Merz. Model checking : A tutorial overview. In F. Cassez, C. Jard, B. Rozoy, and M.D. Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, Berlin, 2001.

[84] Robert T. Monroe. *Rapid Development of Custom Software Architecture Design Environments*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1999. CMU-CS-99-161.

[85] Robert T. Monroe. Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, Pittsburgh, PA 15213, January 2001.

[86] Jean R. Moonen, Judi T. Romijn, Olaf Sies, Jan Springintveld, Loe G. M. Feijs, and Ron L.C. Koymans. A two-level approach to automated conformance testing of VHDL designs. In *IWTCS : International Workshop on Testing Communicating Systems*, Cheju, Korea, Septembre 1997.

[87] Traian Muntean and Kaisa Sere, editors. *RCS'03 : International Workshop on Refinement of Critical Systems*, Turku , Finland, June 2003.

[88] Matthias Mutz. Register transfer level VHDL models without clocks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 153–158, Le Palais des Congrés de Paris, France, February 1998. IEEE Computer Society.

[89] M. Nicoladis, N. Zaidan, T. Calin, and D. Bied-Charreton. SIS : a fail-safe interface realised in smart power technology. In *Proceedings of IEEE International On-Line Testing Workshop*, Palma de Mallorca, Spain, July 2000.

[90] B. Nicolescu and R. Velazco. Detecting soft errors by a purely software approach : Method, tools and experimental results. In *DATE'03 : Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Munich, Germany, March 2003.

[91] Félix Nicoli. *Verification Formelle de descriptions VHDL comportementales*. PhD thesis, Universite d'Aix-Marseille I, 1999.

[92] Sashi Obilisetty, Vice President, General Manager, and Saurin Shroff. The role of HDL checking and reuse in verification. New England Development Center, 2000. http ://www.transeda.com/whitepapers/RuleCheckWP.pdf.

[93] Marcio T. Oliveira and Alan J. Hu. High-level specification and automatic generation of IP interface monitors. In *Proceedings of the 39th conference on Design automation*, pages 129–134. ACM Press, 2002.

[94] Gordon J. Pace. *Hardware Design Based on Verilog HDL*. PhD thesis, Oxford University Computing Laboratory, 1998.

[95] K.S. Papadomanolakis, T. Kakarountas, A. Tsoukalis, S. Nikolaidis, and C.E. Goutis. Low power hardware-software co-design for safety-critical applications. Technical Report EP28593/UP/D2IC&D, University of Patras, Pittsburgh, PA, September 1999.

[96] Terence Parr, John Lilly, Peter Wells, Ric Klaren, and Mika lllouz. *ANTLR Reference Manual*. jGuru, October 2000. http ://ftp.camk.edu.pl/private/chris/antlrman/antlrman.pdf.

[97] David Pellerin. An introduction to HDLs for simulation and synthesis. Technical report, PeakFPGA Customer Service Center, 5252 N Edgewood Dr., Suite 175, Provo, Utah 84604, 2003. http ://www.acceda.com/support/vhdpaper.pdf.

[98] David Pellerin and Douglas Taylor. *VHDL Made Easy !* Hardcover, September 1996.

[99] Lars Philipson and Lunds Tekniska Hogskola. Survey compares formal verification tools. EETims Network http ://www.eetimes.com/story/OEG20011128S0037, November 2001.

[100] Laurence Pierre. An automatic generalization method for the inductive proof of replicated and parallel architectures. In *TPCD : Theorem Provers in Circuit Design*, pages 72–91. Springer-Verlag, April 1995.

[101] Juha Plosila, Kaisa Sere, and Marina Waldén. Component-based asynchronous circuit design in B. Technical report, TUCS, December 2000.

[102] Ralf Reetz, Klaus Schneider, and Thomas Kropf. Formal specification in VHDL for hardware verification. In *Proceedings of the conference on Design, automation and test in Europe*, pages 257–265. IEEE Computer Society, 1998.

[103] Jérome Rocheteau. Utilisation de Phox. INRETS, Mars 2004.

[104] Roshanak Roshandel. Mae : An architectural evolution environment. In *Proceedings of ESEC-FSE03 : Workshop on Specification and Verification of Component-Based System*, Helsinki, Finland, September 2003.

[105] Hisashi Sasaki, Kazunori Mizushima, and Takeshi Sasaki. Semantic validation of VHDL-AMS by an abstract state machine. In *BMAS'97 (IEEE/VIUF International Workshop on Behavioral Modeling and Simulation)*, pages 61–68, Arlington, VA, October 1997.

[106] Steve Schneider. *The B-Method : An Introduction*. cornerstones of computing. Palgrave, 2001.

[107] Emil Sekerinski and Kaisa Sere, editors. *Program Development by Refinement : Case Studies Using the B Method*. Formal Approaches to Computing and Information Technology (FACIT). Springer, 1999.

[108] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions Software Engineering*, 21(4) :314–335, April 1995.

[109] Kanna Shimizu and Stanford University. A methodology for writing and exploiting formal specifications. Formal Verification Group, 2002. http ://chicory.stanford.edu/ kannas/publications/sigda.pdf.

[110] Gregor Siwinski. The increasing importance of HDL verification. *The Quarterly Journal for Xilinx Programmable Logic Users*, pages 27–29, 1999.

[111] Kevin Steppe, Greg Bylenok, David Garlan, Bradley Schmerl, Kanat Abirov, and Nataliya Shevchenko. Two-tiered architectural design for automotive control systems : An experience report. In *Proceedings of Automotive Software Workshop on Future Generation Software Architectures in the Automotive Domain*, San Diego, CA, January 2004.

[112] Steria, Aix en Provence 3, France. *Le Langage B : Manuel de référence*, 1996. modifié en 1998, Ver. 8.

[113] Darly John Stewart. *A Uniform Semantics for Verilog and VHDL Suitable for Both Simulation and Verification*. PhD thesis, St. Catharine's College, University of Cambridge, Cambridge CB2 1RL, UK, August 2001.

[114] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108, Vienna, Austria, 2001. ACM Press.

[115] Stuart Swan, Drik Vermeersch, Dundar Dumlugol, Peter Hardee, Takashi Hasegawa, Adam Rose, and Marello Coppola. Functional specification for SystemC 2.0, April 2002. www.cse.iitd.ernet.in/ panda/SYSTEMC/LangDocs/FuncSpec20.pdf.

[116] John Peter Van Tassel. *Femto-VHDL : The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant*. PhD thesis, University of Cambridge, Cambridge, UK, July 1993.

[117] Aldec Team. Mixed SystemC VHDL and Verilog design entry and verification. http ://www.aldec.com/Riviera/riviera_elite.pdf.

[118] ISIS Project Team. Interfaces fail-safe implémentées en circuits intégrées. Technical Report 1, TIMA : Techniques of Informatics and Microelectronics for computer Architecture, France, Février 2000.

[119] Robert T.Monroe. Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Octobre 1998. Revised January, 2001.

[120] Yves Le Traon, Daniel Deveaux, and Jean-Marc Jézéquel. Self-testable components : from pragmatic tests to design-for-testability methodology. In *Proceedings of Technology of Object-Oriented Languages and Systems Conference TOOLS29*, pages 96–107, Nancy, France, June 1999.

[121] Poul Frederick Williams. *Formal Verification Based on Boolean Expression Diagrams*. PhD thesis, Department of Information Technology, Technical University of Denmark, Lyngby, Denmark, August 2001. ISBN 87-89112-59-8.

[122] Wayne Wolf. What and why about architecture for embedded systems. Vancouver, British Columbia, July 2000.

[123] Dan Zhao, Shambhu Upadhyaya, and Martin Margala. Control constrained resource partitioning for complex SoCs. In *Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03)*, Boston, Massachusetts, November 2003.

THÈSE

pour l'obtention du titre de

DOCTEUR en INFORMATIQUE

à l'Université des Sciences et Technologies de Lille

par

Ammar ALJER

Co-design et Raffinement en B : BHDL Tool, plateforme pour la conception de composants numériques

---

Soutenue le : **21 décembre 2004** devant la Commission d'examen

ANNEXES

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UMR 8022
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 – Télécopie : +33 (0)3 28 77 85 37 – email : `direction@lifl.fr`

Annexe A

STD LOGIC 1164

A.1 La Machine B_STD_LOGIC_1164_0

```
/*?

    FILE        :   B_STD_LOGIC_1164_0.mch
    DESCRIPTION :   On implante le paquetage VHDL STD_LOGIC_1164
?*/


/*
    v0.3    20/08/00    1. Refonte
                        2. Mise a jour pour preparation du paquetage xx_VECTOR
*/

MACHINE
    B_STD_LOGIC_1164_0


/*?
    -- logic state system  (unresolved)
    TYPE std_ulogic IS ( 'U',  -- Uninitialized
                         'X',  -- Forcing  Unknown
                         '0',  -- Forcing  0
                         '1',  -- Forcing  1
                         'Z',  -- High Impedance
                         'W',  -- Weak     Unknown
                         'L',  -- Weak     0
```

```
                        'H',  -- Weak     1
                        '-'   -- Don't care
                      );
?*/

SETS
    STD_ULOGIC  = {     UU      /* uninitialized   */
                ,     XX      /* etat inconnu    */
                ,     OO      /* 0 fort      */
                ,     II      /* 1 fort      */
                ,     ZZ      /* Haute impedance */
                ,     WW      /* Inconnu faible  */
                ,     LL      /* 0 faible    */
                ,     HH      /* 1 faible        */
                ,     DD      /* Don't CARE      */
                }

/*
    UU  sert a initialiser le systeme et permet de verifier
        qu'a tout moment une sortie est bien le resultat d'un calcul
    XX  initialise mais il y a plusieurs sources contradictoire
        0, 1 ou Z
    OO  connecte a la masse
    II  connecte a l'alimentation
    LL, HH  pas de valeurs particuliere, mais interprete
                  respectivement comme 0 et 1
*/

CONSTANTS
/* Les objets suivants sont tous des constantes car
   ils n'evoluent pas dans la specification VHDL
*/
    XMAP
,   BIT
,   XOI
,   XOIZ
,   UXOI
,   UXOIZ
,   STD_LOGIC

/*?
```

```
    -- industry standard logic type

    SUBTYPE std_logic IS resolved std_ulogic;

    -- common subtypes

    SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1';
                       -- ('X','0','1')
    SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z';
                       -- ('X','0','1','Z')
    SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1';
                       -- ('U','X','0','1')
    SUBTYPE UX01Z   IS resolved std_ulogic RANGE 'U' TO 'Z';
                       -- ('U','X','0','1','Z')
?*/

,   AND_ULOGIC  /* TABLE OF THE FUNCTION    */
,   RESOLVED    /* TABLE OF THE FUNCTION RESOLVED   */
,   AND_STD     /* TABLE OF THE FUNCTION AND        */
,   NAND_STD
,   OR_STD      /* TABLE OF THE FUNCTION OR         */
,   NOR_STD
,   XOR_STD     /* TABLE OF THE FUNCTION XOR        */
,   XNOR_STD
,   NOT_STD     /* TABLE OF THE FUNCTION NOT        */
,   CVT_TO_BIT  /* TABLE OF THE FUNCTION CVT_TO_XOI     */
,   CVT_TO_XOI  /* TABLE OF THE FUNCTION CVT_TO_BIT     */
,   CVT_TO_XOIZ     /* TABLE OF THE FUNCTION CVT_TO_XOIZ    */
,   CVT_TO_UXOI /* TABLE OF THE FUNCTION CVT_TO_UXOI    */
,   IS_X_VALUE  /* CONVERSION TABLE OF STD_LOGIC TO BOOL*/


PROPERTIES
          BIT               <:  STD_ULOGIC
&       BIT             =  {   OO   ,   II  }

&       XMAP            : STD_ULOGIC
&         XMAP          : BIT
&       XMAP            = OO

&       XOI             <:  STD_ULOGIC
```

```
&       XOI             =  {   XX   ,   OO   ,   II  }

&       XOIZ            <:  STD_ULOGIC
&       XOIZ            =  {   XX   ,   OO   ,   II   ,   ZZ  }

&       UXOI            <:  STD_ULOGIC
&       UXOI            =  {   UU   ,   XX   ,   OO   ,   II  }

&       UXOIZ           <:  STD_ULOGIC
&       UXOIZ           =  {   UU   ,   XX   ,   OO   ,   II   ,   ZZ  }

&       STD_LOGIC       <:  STD_ULOGIC
/*
        On fait ici un essai, il n'es pas facile de dire qu'elle est
        la meilleur forme pour la preuve a voir sur la duree
    &       STD_LOGIC       =  {   UU   ,   XX   ,   OO   ,   II   ,
                                   ZZ   ,   WW   ,   LL   ,   HH
                                }
*/
&       STD_LOGIC       =   STD_ULOGIC - { DD}


/* OPERATEUR LOGIC DE BASE */
/* ======================= */

&   AND_ULOGIC  : STD_ULOGIC * STD_ULOGIC   --> STD_LOGIC
&   RESOLVED    : STD_ULOGIC * STD_ULOGIC   --> STD_LOGIC

&   AND_STD     : STD_ULOGIC * STD_ULOGIC   --> UXOI
&   NAND_STD    : STD_ULOGIC * STD_ULOGIC   --> UXOI
&   OR_STD      : STD_ULOGIC * STD_ULOGIC   --> UXOI
&   NOR_STD     : STD_ULOGIC * STD_ULOGIC   --> UXOI
&   XOR_STD     : STD_ULOGIC * STD_ULOGIC   --> UXOI
&   XNOR_STD    : STD_ULOGIC * STD_ULOGIC   --> UXOI
&   NOT_STD     : STD_ULOGIC            --> UXOI

/* OPERATEUR DE CONVERSION */
/* ======================= */

&   CVT_TO_XOI  : STD_ULOGIC --> XOI
&   CVT_TO_BIT  : STD_ULOGIC --> BIT
```

```
&   CVT_TO_XOIZ     : STD_ULOGIC --> XOIZ
&   CVT_TO_UXOI     : STD_ULOGIC --> UXOI

/* RETOUR AU BOOLEEN */
/* ================= */

&   IS_X_VALUE  : STD_ULOGIC --> BOOL


/* Et on ajoute les tables de valuation */
/* ==================================== */

&  RESOLVED = { (UU,UU) |-> UU
              , (UU,XX) |-> UU
              , (UU,OO) |-> UU
              , (UU,II) |-> UU
              , (UU,ZZ) |-> UU
              , (UU,WW) |-> UU
              , (UU,LL) |-> UU
              , (UU,HH) |-> UU
              , (UU,DD) |-> UU

              , (XX,UU) |-> UU
              , (XX,XX) |-> XX
              , (XX,OO) |-> XX
              , (XX,II) |-> XX
              , (XX,ZZ) |-> XX
              , (XX,WW) |-> XX
              , (XX,LL) |-> XX
              , (XX,HH) |-> XX
              , (XX,DD) |-> XX

              , (OO,UU) |-> UU
              , (OO,XX) |-> XX
              , (OO,OO) |-> OO
              , (OO,II) |-> XX
              , (OO,ZZ) |-> OO
              , (OO,WW) |-> OO
              , (OO,LL) |-> OO
              , (OO,HH) |-> OO
              , (OO,DD) |-> XX
```

```
    , (II,UU) |-> UU
    , (II,XX) |-> XX
    , (II,OO) |-> XX
    , (II,II) |-> II
    , (II,ZZ) |-> II
    , (II,WW) |-> II
    , (II,LL) |-> II
    , (II,HH) |-> II
    , (II,DD) |-> XX

    , (ZZ,UU) |-> UU
    , (ZZ,XX) |-> XX
    , (ZZ,OO) |-> OO
    , (ZZ,II) |-> XX
    , (ZZ,ZZ) |-> ZZ
    , (ZZ,WW) |-> WW
    , (ZZ,LL) |-> LL
    , (ZZ,HH) |-> HH
    , (ZZ,DD) |-> XX

    , (WW,UU) |-> UU
    , (WW,XX) |-> XX
    , (WW,OO) |-> OO
    , (WW,II) |-> II
    , (WW,ZZ) |-> WW
    , (WW,WW) |-> WW
    , (WW,LL) |-> WW
    , (WW,HH) |-> WW
    , (WW,DD) |-> XX

    , (LL,UU) |-> UU
    , (LL,XX) |-> XX
    , (LL,OO) |-> OO
    , (LL,II) |-> II
    , (LL,ZZ) |-> LL
    , (LL,WW) |-> WW
    , (LL,LL) |-> LL
    , (LL,HH) |-> WW
    , (LL,DD) |-> XX
```

```
                , (HH,UU) |-> UU
                , (HH,XX) |-> XX
                , (HH,OO) |-> OO
                , (HH,II) |-> II
                , (HH,ZZ) |-> HH
                , (HH,WW) |-> WW
                , (HH,LL) |-> WW
                , (HH,HH) |-> HH
                , (HH,DD) |-> XX

                , (DD,UU) |-> UU
                , (DD,XX) |-> XX
                , (DD,OO) |-> XX
                , (DD,II) |-> XX
                , (DD,ZZ) |-> XX
                , (DD,WW) |-> XX
                , (DD,LL) |-> XX
                , (DD,HH) |-> XX
                , (DD,DD) |-> XX
                }

&  AND_STD  = { (UU,UU) |-> UU
                , (UU,XX) |-> UU
                , (UU,OO) |-> OO
                , (UU,II) |-> UU
                , (UU,ZZ) |-> UU
                , (UU,WW) |-> UU
                , (UU,LL) |-> OO
                , (UU,HH) |-> UU
                , (UU,DD) |-> UU

                , (XX,UU) |-> UU
                , (XX,XX) |-> XX
                , (XX,OO) |-> OO
                , (XX,II) |-> XX
                , (XX,ZZ) |-> XX
                , (XX,WW) |-> XX
                , (XX,LL) |-> OO
                , (XX,HH) |-> XX
                , (XX,DD) |-> XX
```

```
    , (OO,UU) |-> OO
    , (OO,XX) |-> OO
    , (OO,OO) |-> OO
    , (OO,II) |-> OO
    , (OO,ZZ) |-> OO
    , (OO,WW) |-> OO
    , (OO,LL) |-> OO
    , (OO,HH) |-> OO
    , (OO,DD) |-> OO

    , (II,UU) |-> UU
    , (II,XX) |-> XX
    , (II,OO) |-> OO
    , (II,II) |-> II
    , (II,ZZ) |-> XX
    , (II,WW) |-> XX
    , (II,LL) |-> OO
    , (II,HH) |-> II
    , (II,DD) |-> XX

    , (ZZ,UU) |-> UU
    , (ZZ,XX) |-> XX
    , (ZZ,OO) |-> OO
    , (ZZ,II) |-> XX
    , (ZZ,ZZ) |-> XX
    , (ZZ,WW) |-> XX
    , (ZZ,LL) |-> OO
    , (ZZ,HH) |-> XX
    , (ZZ,DD) |-> XX

    , (WW,UU) |-> UU
    , (WW,XX) |-> XX
    , (WW,OO) |-> OO
    , (WW,II) |-> XX
    , (WW,ZZ) |-> XX
    , (WW,WW) |-> XX
    , (WW,LL) |-> OO
    , (WW,HH) |-> XX
    , (WW,DD) |-> XX

    , (LL,UU) |-> OO
```

```
                , (LL,XX) |-> OO
                , (LL,OO) |-> OO
                , (LL,II) |-> OO
                , (LL,ZZ) |-> OO
                , (LL,WW) |-> OO
                , (LL,LL) |-> OO
                , (LL,HH) |-> OO
                , (LL,DD) |-> OO

                , (HH,UU) |-> UU
                , (HH,XX) |-> XX
                , (HH,OO) |-> OO
                , (HH,II) |-> II
                , (HH,ZZ) |-> XX
                , (HH,WW) |-> XX
                , (HH,LL) |-> OO
                , (HH,HH) |-> II
                , (HH,DD) |-> XX

                , (DD,UU) |-> UU
                , (DD,XX) |-> XX
                , (DD,OO) |-> OO
                , (DD,II) |-> XX
                , (DD,ZZ) |-> XX
                , (DD,WW) |-> XX
                , (DD,LL) |-> OO
                , (DD,HH) |-> XX
                , (DD,DD) |-> XX
                }

&     OR_STD = { (UU,UU) |-> UU
                , (UU,XX) |-> UU
                , (UU,OO) |-> UU
                , (UU,II) |-> II
                , (UU,ZZ) |-> UU
                , (UU,WW) |-> UU
                , (UU,LL) |-> UU
                , (UU,HH) |-> II
                , (UU,DD) |-> UU

                , (XX,UU) |-> UU
```

```
    , (XX,XX) |-> XX
    , (XX,OO) |-> XX
    , (XX,II) |-> II
    , (XX,ZZ) |-> XX
    , (XX,WW) |-> XX
    , (XX,LL) |-> XX
    , (XX,HH) |-> II
    , (XX,DD) |-> XX

    , (OO,UU) |-> UU
    , (OO,XX) |-> XX
    , (OO,OO) |-> OO
    , (OO,II) |-> II
    , (OO,ZZ) |-> XX
    , (OO,WW) |-> XX
    , (OO,LL) |-> OO
    , (OO,HH) |-> II
    , (OO,DD) |-> XX

    , (II,UU) |-> II
    , (II,XX) |-> II
    , (II,OO) |-> II
    , (II,II) |-> II
    , (II,ZZ) |-> II
    , (II,WW) |-> II
    , (II,LL) |-> II
    , (II,HH) |-> II
    , (II,DD) |-> II

    , (ZZ,UU) |-> UU
    , (ZZ,XX) |-> XX
    , (ZZ,OO) |-> XX
    , (ZZ,II) |-> II
    , (ZZ,ZZ) |-> XX
    , (ZZ,WW) |-> XX
    , (ZZ,LL) |-> XX
    , (ZZ,HH) |-> II
    , (ZZ,DD) |-> XX

    , (WW,UU) |-> UU
    , (WW,XX) |-> XX
```

```
                , (WW,OO) |-> XX
                , (WW,II) |-> II
                , (WW,ZZ) |-> XX
                , (WW,WW) |-> XX
                , (WW,LL) |-> XX
                , (WW,HH) |-> II
                , (WW,DD) |-> XX

                , (LL,UU) |-> UU
                , (LL,XX) |-> XX
                , (LL,OO) |-> OO
                , (LL,II) |-> II
                , (LL,ZZ) |-> XX
                , (LL,WW) |-> XX
                , (LL,LL) |-> OO
                , (LL,HH) |-> II
                , (LL,DD) |-> XX

                , (HH,UU) |-> II
                , (HH,XX) |-> II
                , (HH,OO) |-> II
                , (HH,II) |-> II
                , (HH,ZZ) |-> II
                , (HH,WW) |-> II
                , (HH,LL) |-> II
                , (HH,HH) |-> II
                , (HH,DD) |-> II

                , (DD,UU) |-> UU
                , (DD,XX) |-> XX
                , (DD,OO) |-> XX
                , (DD,II) |-> II
                , (DD,ZZ) |-> XX
                , (DD,WW) |-> XX
                , (DD,LL) |-> XX
                , (DD,HH) |-> II
                , (DD,DD) |-> XX
               }

& XOR_STD    = { (UU,UU) |-> UU
                , (UU,XX) |-> UU
```

```
    , (UU,OO) |-> UU
    , (UU,II) |-> UU
    , (UU,ZZ) |-> UU
    , (UU,WW) |-> UU
    , (UU,LL) |-> UU
    , (UU,HH) |-> UU
    , (UU,DD) |-> UU

    , (XX,UU) |-> UU
    , (XX,XX) |-> XX
    , (XX,OO) |-> XX
    , (XX,II) |-> XX
    , (XX,ZZ) |-> XX
    , (XX,WW) |-> XX
    , (XX,LL) |-> XX
    , (XX,HH) |-> XX
    , (XX,DD) |-> XX

    , (OO,UU) |-> UU
    , (OO,XX) |-> XX
    , (OO,OO) |-> OO
    , (OO,II) |-> II
    , (OO,ZZ) |-> XX
    , (OO,WW) |-> XX
    , (OO,LL) |-> OO
    , (OO,HH) |-> II
    , (OO,DD) |-> XX

    , (II,UU) |-> UU
    , (II,XX) |-> XX
    , (II,OO) |-> II
    , (II,II) |-> OO
    , (II,ZZ) |-> XX
    , (II,WW) |-> XX
    , (II,LL) |-> II
    , (II,HH) |-> OO
    , (II,DD) |-> XX

    , (ZZ,UU) |-> UU
    , (ZZ,XX) |-> XX
    , (ZZ,OO) |-> XX
```

```
, (ZZ,II) |-> II
, (ZZ,ZZ) |-> XX
, (ZZ,WW) |-> XX
, (ZZ,LL) |-> XX
, (ZZ,HH) |-> II
, (ZZ,DD) |-> XX

, (WW,UU) |-> UU
, (WW,XX) |-> XX
, (WW,OO) |-> XX
, (WW,II) |-> II
, (WW,ZZ) |-> XX
, (WW,WW) |-> XX
, (WW,LL) |-> XX
, (WW,HH) |-> II
, (WW,DD) |-> XX

, (LL,UU) |-> UU
, (LL,XX) |-> XX
, (LL,OO) |-> OO
, (LL,II) |-> II
, (LL,ZZ) |-> XX
, (LL,WW) |-> XX
, (LL,LL) |-> OO
, (LL,HH) |-> II
, (LL,DD) |-> XX

, (HH,UU) |-> II
, (HH,XX) |-> II
, (HH,OO) |-> II
, (HH,II) |-> II
, (HH,ZZ) |-> II
, (HH,WW) |-> II
, (HH,LL) |-> II
, (HH,HH) |-> II
, (HH,DD) |-> II

, (DD,UU) |-> UU
, (DD,XX) |-> XX
, (DD,OO) |-> XX
, (DD,II) |-> II
```

```
                , (DD,ZZ) |-> XX
                , (DD,WW) |-> XX
                , (DD,LL) |-> XX
                , (DD,HH) |-> II
                , (DD,DD) |-> XX
                }
&  NOT_STD  = {  UU      |-> XX
                , XX      |-> XX
                , OO      |-> II
                , II      |-> OO
                , ZZ      |-> XX
                , WW      |-> XX
                , LL      |-> II
                , HH      |-> OO
                , DD      |-> XX
                }
& CVT_TO_BIT ={  UU      |-> XMAP
                , XX      |-> XMAP
                , OO      |-> OO
                , II      |-> II
                , ZZ      |-> XMAP
                , WW      |-> XMAP
                , LL      |-> OO
                , HH      |-> II
                , DD      |-> XMAP
                }

& CVT_TO_XOI ={  UU      |-> XX
                , XX      |-> XX
                , OO      |-> OO
                , II      |-> II
                , ZZ      |-> XX
                , WW      |-> XX
                , LL      |-> OO
                , HH      |-> II
                , DD      |-> XX
                }

& CVT_TO_XOIZ={  UU      |-> XX
                , XX      |-> XX
                , OO      |-> OO
```

```
                , II      |-> II
                , ZZ      |-> ZZ
                , WW      |-> XX
                , LL      |-> OO
                , HH      |-> II
                , DD      |-> XX
                }

& CVT_TO_UX0I={  UU      |-> UU
                , XX      |-> XX
                , OO      |-> OO
                , II      |-> II
                , ZZ      |-> XX
                , WW      |-> XX
                , LL      |-> OO
                , HH      |-> II
                , DD      |-> XX
                }

& IS_X_VALUE= {  UU      |-> TRUE
                , XX      |-> TRUE
                , OO      |-> FALSE
                , II      |-> FALSE
                , ZZ      |-> TRUE
                , WW      |-> TRUE
                , LL      |-> FALSE
                , HH      |-> FALSE
                , DD      |-> TRUE
                }

/* Il y a aussi des fonctions construites a partir des autres */
&   NAND_STD    =  (AND_STD;NOT_STD)
&   NOR_STD     =  (OR_STD;NOT_STD)
&   XNOR_STD    =  (XOR_STD;NOT_STD)


OPERATIONS

/*
    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
```

```
    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
--  function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
    function xnor   ( l : std_ulogic; r : std_ulogic ) return ux01;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;
*/

out <-- Resolved(in1,in2)=
PRE
        in1,in2      :  STD_ULOGIC * STD_ULOGIC
THEN
    out:=RESOLVED(in1,in2)
END;

out  <-- And(in1,in2)=
PRE
        in1,in2      :  STD_ULOGIC * STD_ULOGIC
THEN
    out := AND_STD(in1,in2)
END;

out  <-- Nand(in1,in2)=
PRE
        in1,in2      :  STD_ULOGIC * STD_ULOGIC
THEN
    out :=  NAND_STD(in1,in2)
END;

out <-- Or(in1,in2)=
PRE
        in1,in2      :  STD_ULOGIC * STD_ULOGIC
THEN
    out := OR_STD(in1,in2)
END;

out <-- Nor(in1,in2)=
PRE
        in1,in2      :  STD_ULOGIC * STD_ULOGIC
THEN
    out := NOR_STD(in1,in2)
```

```
END;

out <-- Xor(in1,in2)=
PRE
        in1,in2      :  STD_ULOGIC * STD_ULOGIC
THEN
    out := XOR_STD(in1,in2)
END;

out <-- Xnor(in1,in2)=
PRE
        in1,in2      :  STD_ULOGIC * STD_ULOGIC
THEN
    out :=  XNOR_STD(in1,in2)
END;

out <-- Not(in1)=
PRE
        in1      :  STD_ULOGIC
THEN
    out := NOT_STD(in1)
END;


/* OPERATION de CONVERSION */
/* ======================= */

out <-- From_STD_ULOGIC_To_BIT(in1)=
PRE
        in1:STD_ULOGIC
THEN
    out := CVT_TO_BIT(in1)
END;

/*
    FUNCTION To_StdULogic       ( b : BIT       ) RETURN std_ulogic;
*/

out <-- From_BIT_To_StdULOGIC(in1)=
PRE
        in1:BIT
```

```
THEN
    /* Pour forcer le typage,   il faut passer par un ANY afin de
       definir explicitement le type de la variable de sortie out
       sinon son type sera BIT et non pas STD_ULOGIC
    */
    ANY
        temp
    WHERE
        temp : STD_ULOGIC
    &   temp = in1
    THEN
        out := temp
    END
END;

/*
    FUNCTION To_X01  ( b : BIT            ) RETURN  X01;
*/

out <-- From_BIT_To_XOI(in1)=
PRE
        in1:BIT
THEN
    out := CVT_TO_XOI(in1)
END;

/*
    FUNCTION To_X01Z ( b : BIT            ) RETURN  X01Z;
*/

out <-- From_BIT_To_XOIZ(in1)=
PRE
        in1:BIT
THEN
    out := CVT_TO_XOIZ(in1)
END;

/*
    FUNCTION To_UX01  ( b : BIT            ) RETURN  UX01;
*/
```

```
out <-- From_BIT_To_UXOI(in1)=
PRE
        in1:BIT
THEN
    out := CVT_TO_UXOI(in1)
END;

/*
    FUNCTION To_X01  ( s : std_ulogic     ) RETURN  X01;
*/

out <-- From_STD_ULOGIC_To_XOI(in1)=
PRE
        in1:STD_ULOGIC
THEN
    out := CVT_TO_XOI(in1)
END;

/*
    FUNCTION To_X01Z ( s : std_ulogic     ) RETURN  X01Z;
*/

out <-- From_STD_ULOGIC_To_XOIZ(in1)=
PRE
        in1:STD_ULOGIC
THEN
    out := CVT_TO_XOIZ(in1)
END;

/*
  FUNCTION To_UX01  ( s : std_ulogic     ) RETURN  UX01;
*/

out <-- From_STD_ULOGIC_To_UXOI(in1)=
PRE
        in1:STD_ULOGIC
THEN
    out := CVT_TO_UXOI(in1)
END;
```

```
/*? Passage d'un STD_ULOGIC au BOOLEEN ?*/
/*  ================================  */

/*
    FUNCTION Is_X ( s : std_ulogic     ) RETURN  BOOLEAN;
*/

out <-- IS_X(in1)=
PRE
       in1:STD_ULOGIC
THEN
   out := IS_X_VALUE(in1)
END;



/*?     Operation d'impression:
    Il est important de prevoire une operation
      d'impression pour les tests du module
?*/

Print (bb) =
PRE
    bb : STD_ULOGIC
THEN
    skip
END



END /*? of file B_STD_LOGIC_1164_0.mch ?*/
```

A.2 B_STD_LOGIC_1164_VECTOR_0

```
/*?
FILE :  B_STD_LOGIC_1164_VECTOR_0.mch
DESCRIPTION :  On implante la partie "*_vector"
                        du paquetage VHDL STD_LOGIC_1164
?*/

/* RELEASES

   v0.2 20/08/00
   1. Refonte
   2. Mise a jour pour preparation du paquetage xx_VECTOR
*/

MACHINE
      B_STD_LOGIC_1164_VECTOR_0

DEFINITIONS
      APPLY(op,in1,in2,out) ==
        ANY
           vv
        WHERE
           vv : STD_LOGIC_VECTOR
     & !xx.( (xx : 1 .. max({in1'vector_size,in2'vector_size}) )
                => ( (vv'vector)(xx) = op( (in1'vector)(xx)
                                         , (in2'vector)(xx)
                                         )
                   )
                )
       THEN
           out := vv
       END
; MAX_VECTOR == 1000

SEES
      IEEE.B_STD_LOGIC_1164_0

CONSTANTS
      STD_LOGIC_VECTOR
```

```
,STD_ULOGIC_VECTOR
,Max_Element


PROPERTIES
      Max_Element : NAT1

/*?
     On construit l'ensemble de tous les vecteurs de STD_LOGIC
?*/
& STD_LOGIC_VECTOR = struct (
                 vector : 1.. Max_Element --> STD_LOGIC,
                 vector_size : NAT1
)
& card(STD_LOGIC_VECTOR) <MAX_VECTOR
& !xx.((xx:STD_LOGIC_VECTOR) => xx'vector_size < Max_Element)

/* Idem pour STD_ULOGIC */
& STD_ULOGIC_VECTOR = struct (
                      vector : 1.. Max_Element --> STD_LOGIC,
                      vector_size : NAT1
                     )
& card(STD_ULOGIC_VECTOR) <MAX_VECTOR
& !xx.((xx:STD_ULOGIC_VECTOR) => xx'vector_size < Max_Element)

OPERATIONS

out <-- Resolved(in1,in2)=
PRE
          in1,in2      :  STD_ULOGIC_VECTOR * STD_ULOGIC_VECTOR
THEN
          APPLY(RESOLVED,in1,in2,out)
END;

out  <-- And(in1,in2)=
PRE
      in1 : STD_LOGIC_VECTOR
& in2 : STD_LOGIC_VECTOR
THEN
      APPLY(AND_STD,in1,in2,out)
END;
```

```
out  <-- Or(in1,in2)=
PRE
      in1 : STD_LOGIC_VECTOR
& in2 : STD_LOGIC_VECTOR
THEN
      APPLY(OR_STD,in1,in2,out)
END;


/*?  Operation d'impression:
      Il est important de prevoire une operation
           d'impression pour les tests du module
?*/

Print (bb) =
PRE
      bb : STD_LOGIC_VECTOR
THEN
      skip
END

END /*? of file B_STD_LOGIC_1164_VECTOR_0.mch ?*/
```

Annexe B

Parseur BHDL

```
// Antlr grammar for VHDL imported from "The University of Cincinnati"
//  grammar (written in PCCTS)
// Some modifications have been done :
// - The rule "options" has been renamed => reserved word

// - May 13, 2003 : A part of the lexer (written in flex) has been put
//   in the parser part.
//   For instance, some of the lexical rules ("decimal_integer_literal",
//   "exp", "decimal_floating_point_literal", "based_floating_point_literal"
//   and "real_literal", ...) have been translated in the parser (see
//   BASED_LITERAL, BASED_INTEGER,EXP, DECIMAL_INTEGER_LITERAL and
//   BASED_INTEGER_LITERAL, abstract_literal and integer_literal.)

// - May 19, 2003 : The grammar is associated with a treewalker (treewalker.g).
//   It is the reason why some rules have been modified, especially in the
//   case of recursive structures based on COMMA, that is,by adding an
//   intermediate symbol.

//  May 19, 2003, Ammar Aljer, Philippe Devienne, Pierre Antoine Laloux

//  July 26, 2003
//     The parser has been modified to accept VHDL code generated by VGUI
//     Empty units are removed, useful VHDL comments which are useful in B
//     They are captured in order to form INVARIANT clause and
//                                        DEFINITION clause in B

//  07,07,2003 Ammar Aljer, all the ?s are compared with the original version
```

```
// Copyright (c) 1993-2001 The University of Cincinnati.
// All rights reserved.

// UC MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
// THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO
// THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
// PURPOSE, OR NON-INFRINGEMENT.  UC SHALL NOT BE LIABLE FOR ANY DAMAGES
// SUFFERED BY LICENSEE AS A RESULT OF USING, RESULT OF USING, MODIFYING
// OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

// By using or copying this Software, Licensee agrees to abide by the
// intellectual property laws, and all other applicable laws of the
// U.S., and the terms of this license.

// You may modify, distribute, and use the software contained in this
// package under the terms of the "GNU LIBRARY GENERAL PUBLIC LICENSE"
// version 2, June 1991. A copy of this license agreement can be found
// in the file "LGPL", distributed with this archive.

// Authors: Philip A. Wilsey    philip.wilsey@ieee.org
//          Dale E. Martin      dmartin@cliftonlabs.com
//          Timothy J. McBrayer
//          Malolan Chetlur     mal@ece.uc.edu

//----------------------------------------------------------------------------
//
// $Id: vhdl.gg,v 1.37 2001/08/28 14:09:28 dmartin Exp $
//
//----------------------------------------------------------------------------

{
import java.io.*;
import antlr.collections.AST;

}

class bhdlParser extends Parser ;

options {
    exportVocab=vhdl;               // Call its vocabulary "vhdl"
    defaultErrorHandler = true ;
```

```
    buildAST=true;

    k=2;

         }


tokens
{
TOPLEVEL      ="top_level";
ABSTRACT      = "abstract";
CONCRETE      = "concrete";
ACTIVE        = "active" ;
ASCENDING     = "ascending" ;
BASE          = "base" ;
DELAYED       = "delayed" ;
DRIVING       = "driving" ;
DRIVING_VALUE = "driving_value" ;
EVENT         = "event" ;
HIGH          = "high" ;
//These tokens (IDENT...) are used to abduct the non determinism
// due to recursive sentences
IDENT         = "logical_library_name_list_root";
IDENT1        = "aggregate_root";
IDENT2        = "selected_waveforms_root";
IDENT3        = "sensitivity_list_root";
IDENT4        = "sequential_waveform_root";
IDENT5        = "entity_name_list_root";
IDENT6        = "instantiation_list_root";
IDENT7        = "signal_list_root";
IDENT8        = "enumeration_type_definition_root";
IDENT9        = "array_type_definition_head_root";
IDENT10       = "index_constraint_root";
IDENT11       = "identifier_list_root";
IDENT12       = "association_list_in_root";
IDENT13       = "entity_class_entry_list_root";
IDENT14       = "group_constituent_list_root";
IDENT15       = "association_list_out_root";
IDENT16       = "step_limit_specification_head_root";
IDENT17       = "array_nature_definition_head_root";
IDENT18       = "signal_assignment_st";
```

```
IDENT19       = "wave_element";
IDENT20       = "se_st";
IDENT21       = "label1";
IDENT22       = "colon1";
IDENT23       = "costat2b";
IDENT24       = "instantiate_stmt";
IDENT25       = "newcomp";

IMAGE         = "image" ;
INSTANCE_NAME = "instance_name" ;
LAST_ACTIVE   = "last_active" ;
LAST_EVENT    = "last_event" ;
LAST_VALUE    = "last_value" ;
LEFT          = "left" ;
LEFTOF        = "leftof" ;
LENGTH        = "length" ;
LOW           = "low" ;
PATH_NAME     = "path_name" ;
POS           = "pos" ;
PRED          = "pred" ;
QUIET         = "quiet " ;
REVERSE_RANGE = "reverse_range" ;
RIGHT         = "right" ;
RIGHTOF       = "rightof" ;
SIMPLE_NAME   = "simple_name" ;
STABLE        = "stable" ;
SUCC          = "succ" ;
TRANSACTION   = "transaction " ;
VAL           = "val" ;
VALUE         = "value" ;


ABS           = "abs" ;
ACCESS        = "access" ;
ACROSS        = "across" ;
AFTER         = "after" ;
ALIAS         = "alias" ;
ALL           = "all" ;
AND           = "and" ;
ARCHITECTURE  = "architecture" ;
ARRAY         = "array" ;
```

```
ASSERT        = "assert" ;
ATTRIBUTE     = "attribute" ;
BEGIN         = "begin" ;
BLOCK         = "block" ;
BODY          = "body" ;
BREAK         = "break" ;
BUFFER        = "buffer" ;
BUS           = "bus" ;
CASE          = "case" ;
COMPONENT     = "component" ;
CONFIGURATION = "configuration" ;
CONSTANT      = "constant" ;
DISCONNECT    = "disconnect" ;
DOWNTO        = "downto" ;
ELSE          = "else" ;
ELSIF         = "elsif" ;
END           = "end" ;
ENTITY        = "entity" ;
EXIT          = "exit" ;
FILE          = "file" ;
FOR           = "for" ;
FUNCTION      = "function" ;
GENERATE      = "generate" ;
GENERIC       = "generic" ;
GROUP         = "group" ;
GUARDED       = "guarded" ;
IF            = "if" ;
IMPURE        = "impure" ;
IN            = "in" ;
INERTIAL      = "inertial" ;
INOUT         = "inout" ;
IS            = "is" ;
LABEL         = "label" ;
LIBRARY       = "library" ;
LIMIT         = "limit " ;
LINKAGE       = "linkage" ;
LITERAL       = "literal" ;
LOOP          = "loop" ;
MAP           = "map" ;
MOD           = "mod" ;
NAND          = "nand" ;
```

```
NATURE        = "nature " ;
NEW           = "new" ;
NEXT          = "next" ;
NOISE         = "noise " ;
NOR           = "nor" ;
NOT           = "not" ;
NULL          = "null" ;
OF            = "of" ;
ON            = "on" ;
OPEN          = "open" ;
OR            = "or" ;
OTHERS        = "others" ;
OUT           = "out" ;
PACKAGE       = "package" ;
PORT          = "port" ;
POSTPONED     = "postponed" ;
PROCEDURAL    = "procedural" ;
PROCEDURE     = "procedure" ;
PROCESS       = "process" ;
PROTECTED     = "protected" ;
PURE          = "pure" ;
QUANTITY      = "quantity" ;
RANGE         = "range" ;
RECORD        = "record" ;
REFERENCE     = "reference" ;
REGISTER      = "register" ;
REJECT        = "reject" ;
REM           = "rem" ;
REPORT        = "report" ;
RETURN        = "return" ;
ROL           = "rol" ;
ROR           = "ror" ;
SELECT        = "select" ;
SEVERITY      = "severity" ;
SHARED        = "shared" ;
SIGNAL        = "signal" ;
SLA           = "sla" ;
SLL           = "sll" ;
SPECTRUM      = "spectrum" ;
SRA           = "sra" ;
SRL           = "srl" ;
```

```
SUBNATURE     = "subnature" ;
SUBTYPE       = "subtype" ;
TERMINAL      = "terminal" ;
THEN          = "then" ;
THROUGH       = "through" ;
TO            = "to" ;
TOLERANCE     = "tolerance" ;
TRANSPORT     = "transport" ;
TYPE          = "type" ;
UNAFFECTED    = "unaffected" ;
UNITS         = "units" ;
UNTIL         = "until" ;
USE           = "use" ;
VARIABLE      = "variable" ;
WAIT          = "wait" ;
WHEN          = "when" ;
WHILE         = "while" ;
WITH          = "with" ;
XNOR          = "xnor" ;
XOR           = "xor" ;

// CHOICE        = "choice" ;

}
{
// This variable is used to memorize the entity of an architecture in order
// to transfer the invariant comment and definition comment between both of
// them using a tree comment
String Entity_of_Arc="";

//A tree that containes all of B comments
AST comments=null;

// This variable is used to concat many lines of comments in only one string
String commentcollect="";
}

design_file! :
                {  // This branch create the root of VHDL comments
                   // which are useful in B.
                    #comments=#([IDENTIFIER,"comments"]);
```

```
                }
      df:design_file1
                {
                    #comments.setNextSibling(#df);
                    #design_file=#comments;
                }
    ;

design_file1 :
      ( design_unit )+ EOF!
    ;

design_unit :
      context_clauses library_unit
    ;

library_unit :
      primary_unit
    | secondary_unit
    ;

primary_unit :
      entity_declaration
    | configuration_declaration
    | package_declaration
    ;

secondary_unit :
      architecture_body
    | package_body
    ;

library_clause :
      LIBRARY^ logical_library_name_list SEMI_COLON
    ;

logical_library_name_list !:
      ln1:logical_name ln2:logical_library_name_list_tail
                {#logical_library_name_list =
                            #([IDENT
                             , "logical_library_name_list_root"]
```

```
                              , ln1
                              , ln2
                             );
                }
    ;

logical_library_name_list_tail :
    ( COMMA logical_name )*
    ;

logical_name :
      identifier
    ;

context_clauses :
      ( context_item )*
    ;

context_item :
      library_clause
    | use_clause
    ;

//entity_declaration :
//ENTITY^ identifier IS entity_header entity_declarative_part
//( entity_statement_part )? END  ( ENTITY )? ( simple_name )? SEMI_COLON
//;
// entity_declaration is modified in order to accepte empty entity that
// is generated
// by vgui
// empty entity will be deleted

entity_declaration :
// to ignore empty entities which are generated by VGUI
     !(ENTITY (identifier) IS END)
        => ENTITY (identifier|TOPLEVEL) IS END
          (identifier|TOPLEVEL) SEMI_COLON
    | (ENTITY identifier IS entity_header)
        => ENTITY^  identifier IS e:entity_header
                    entity_declarative_part
                    ( entity_statement_part )?
```

```
                    END  ( ENTITY )? ( simple_name )?
                    SEMI_COLON
    ;

entity_header :
      ( generic_clause )? ( port_clause )?
    ;

entity_declarative_part :
      ( entity_declarative_item )*
    ;

entity_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declarative_item
    | disconnection_specification
    | use_clause
    | ( LIMIT | NATURE | SUBNATURE | QUANTITY | TERMINAL )
          => ( step_limit_specification
               | nature_declaration
               | subnature_declaration
               | quantity_declaration
               | terminal_declaration
             )
    | ( GROUP IDENTIFIER COLON )
         => group_declaration
    | group_template_declaration
    ;

entity_statement_part :
      ( BEGIN )? ( entity_statement )*
    ;

entity_statement :
```

```
      ( stmt_label )? ( POSTPONED )? entity_stmt
    ;

entity_stmt :
      concurrent_assertion_statement
    | concurrent_procedure_call
    | process_statement
    ;

concurrent_procedure_call :
      complex_name SEMI_COLON
    ;

//architecture_body :
// ARCHITECTURE^
//    identifier OF son:selected_or_simple_name IS
//    architecture_declarative_part
//    architecture_statement_part END ( ARCHITECTURE )?
//    ( simple_name )? SEMI_COLON
//    ;
// In order to delete top_level archirecure we may modify
// architecture_body
// as fowllows

architecture_body :
// to ignore empty architectures which are generated by VGUI.
// and to ignore top_level architectures generated by VGUI.
      ! (ARCHITECTURE identifier OF TOPLEVEL)
            =>  ( ARCHITECTURE identifier OF TOPLEVEL IS
                  architecture_declarative_part
                  architecture_statement_part END ( ARCHITECTURE )?
                  ( simple_name )? SEMI_COLON
                )
    | ! ( ARCHITECTURE^ identifier OF selected_or_simple_name IS
           BEGIN END  simple_name SEMI_COLON
          )
            =>  ( ARCHITECTURE identifier OF selected_or_simple_name
                  IS BEGIN END   simple_name  SEMI_COLON
                )
    | (ARCHITECTURE identifier OF selected_or_simple_name)
         =>  ( ARCHITECTURE^ identifier OF son:selected_or_simple_name
```

```
                IS architecture_declarative_part
                  architecture_statement_part END ( ARCHITECTURE )?
                  ( simple_name )? SEMI_COLON
              )
    ;

architecture_declarative_part :
      ( block_declarative_item )*
    ;

architecture_statement_part :
      ( BEGIN )? ( architecture_statement_element )*
    ;

architecture_statement_element :
      ( CASE | NULL | PROCEDURAL | IF condition USE
       | simple_expression EQUAL_EQUAL
      )
          => simultaneous_statement
    | concurrent_statement
    ;

configuration_declaration :
       CONFIGURATION^ identifier OF selected_or_simple_name IS
       configuration_declarative_part  block_configuration END
       ( CONFIGURATION )? ( simple_name )? SEMI_COLON
    ;

configuration_declarative_part :
      ( configuration_declarative_item )*
    ;

configuration_declarative_item :
      use_clause
    | attribute_specification
    | group_declaration
    ;

block_configuration :
      FOR^ block_specification ( use_clause )*
      ( configuration_item )* END FOR SEMI_COLON
```

```
    ;

block_specification :
      selected_or_simple_name ( L_PAREN index_specifier R_PAREN )?
    ;

configuration_item :
      ( component_configuration )
          => ( component_configuration )
    | block_configuration
    ;

component_configuration :
      FOR^ component_specification ( binding_indication SEMI_COLON )?
      ( block_configuration )? END FOR SEMI_COLON
    ;

concurrent_statement! :
                     {#concurrent_statement=#([IDENT21,"label1"]);}
      ( st:stmt_label {#concurrent_statement.addChild(#st); })?

      (co: COLON {#concurrent_statement.addChild(#co);})?

       cs:concurrent_stmt {#concurrent_statement.setNextSibling(#cs);}
       // {#concurrent_statement.addChild(#cs);}
    ;

concurrent_stmt :
       ( BLOCK | FOR | IF | BREAK | CASE | NULL | PROCEDURAL |
         simple_expression EQUAL_EQUAL
        )
            => not_postponeable
    |!  ( po:POSTPONED! )? pos:postponeable
         {#concurrent_stmt=#([IDENT23,"costat2b"],pos);}
    ;

not_postponeable :
      block_statement
    | ( generate_scheme GENERATE )
          => generate_statement
    | ( IF | CASE | NULL | PROCEDURAL | simple_expression EQUAL_EQUAL )
```

```
          => simultaneous_stmt
    | concurrent_break_statement
    ;

postponeable :
      process_statement
    | concurrent_assertion_statement
    | selected_signal_assignment_statement
    | ( target LESS_EQUAL optionss conditional_waveforms SEMI_COLON)
        => conditional_signal_assignment
    //commentsr is a clause of B comments
    | (   COMPONENT
        | ENTITY
        | CONFIGURATION
        | cn:complex_name c:commentsr! (   generic_map_aspect
                                         | port_map_aspect
                                       )
       )   =>  instantiate_statement
    | concurrent_procedure_call
    ;

 commentsr :
      ( bcomment_defs bcomment_e bcomment_invs )?
    ;

//bcomment_defs! :
//            {
//                    commentcollect="";
//            }
//      (bd:BCOMMENT_DEF
        { commentcollect=commentcollect.concat(bd.getText()+"\r\n"); } )+
//            {
//                    #bcomment_defs=#([BCOMMENT_DEF,commentcollect]);
//            }
//    ;

//bcomment_invs! :
//            {
//                    commentcollect="";
//            }
//    ( bd:BCOMMENT_INV {commentcollect=
```

```
                              commentcollect.concat(bd.getText());} )+
//            {
//                      #bcomment_invs=#([BCOMMENT_INV,commentcollect]);
//            }
//    ;
bcomment_defs :
    BCOMMENT_DEF! L_PAREN! identifier_list1 R_PAREN!
;

bcomment_invs :
     BCOMMENT_INV! L_PAREN! identifier_list1 R_PAREN!
;

bcomment_e! :
            {
                    commentcollect="";
            }
    ( bd:BCOMMENT_E
         {commentcollect=commentcollect.concat(bd.getText()+"\r\n");}
     )+
            {
                    #bcomment_e= #([BCOMMENT_E,commentcollect]);
            }
    ;


block_statement :
      BLOCK^ ( L_PAREN expression R_PAREN )?
      ( IS )? block_header block_declarative_part
      architecture_statement_part END BLOCK ( simple_name )? SEMI_COLON
    ;

block_header :
      ( generic_clause ( generic_map_aspect SEMI_COLON )? )?
      ( port_clause ( port_map_aspect SEMI_COLON )? )?

    ;

block_declarative_part :
      ( block_declarative_item )*
    ;
```

```
block_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | disconnection_specification
    | attribute_declarative_item
    | configuration_specification
    | use_clause
    | ( LIMIT | NATURE | SUBNATURE | QUANTITY | TERMINAL )
          => ( step_limit_specification
                 | nature_declaration
                 | subnature_declaration
                 | quantity_declaration
                 | terminal_declaration
             )
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

process_statement :
      PROCESS^ ( L_PAREN sensitivity_list R_PAREN )?
        ( IS )? process_declarative_part
      sequence_of_statements END ( POSTPONED )?
        PROCESS ( simple_name )? SEMI_COLON
    ;

process_declarative_part :
      ( process_declarative_item )*
    ;

process_declarative_item :
      subprogram
    | type_declaration
```

```
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declarative_item
    | use_clause
    | ( GROUP IDENTIFIER COLON ) => group_declaration
    | group_template_declaration
    ;

concurrent_assertion_statement :
      assertion SEMI_COLON
    ;

concurrent_call_statement :
      complex_name SEMI_COLON
    ;

instantiate_statement! :
              {#instantiate_statement=
                     #([IDENT24,"instantiate_stmt"]);
              }
      (    en:ENTITY
             {#instantiate_statement.addChild(#en);}
          |co:CONFIGURATION
             {#instantiate_statement.addChild(#co);}
          |( COMPONENT )?
       )

       cn:complex_name
         {#instantiate_statement.addChild(#cn);}
       c:commentsr
      {
          AST current_Comment_Tree=null;
          String current_archi=#cn.getText();
          if (#c!=null)
          {
                #current_Comment_Tree=#([IDENTIFIER,current_archi],c);
                #comments.addChild(#current_Comment_Tree);
          }
```

```
      }
      ( gma:generic_map_aspect
         {#instantiate_statement.addChild(#gma);}
       )?
       (
       pma:port_map_aspect
           {#instantiate_statement.addChild(#pma);}
       )?
       sc:SEMI_COLON
        {#instantiate_statement.addChild(#sc);}

//    | ENTITY^ complex_name ( generic_map_aspect )?
          ( port_map_aspect )? SEMI_COLON
//    | CONFIGURATION^ complex_name ( generic_map_aspect )?
           ( port_map_aspect )? SEMI_COLON
// The same treatement for  all branchs
    | ENTITY complex_name ( generic_map_aspect )?
        ( port_map_aspect )? SEMI_COLON
    | CONFIGURATION complex_name ( generic_map_aspect )?
        ( port_map_aspect )? SEMI_COLON
    ;

conditional_signal_assignment :
      target LESS_EQUAL optionss conditional_waveforms SEMI_COLON
    ;

target :
      ( complex_name ) => ( complex_name )
    | aggregate
    ;

aggregate !:
      lp:L_PAREN ae:aggregate_entry at:aggregate_tail rp:R_PAREN
      {#aggregate = #([IDENT1, "aggregate_root"], lp, ae, at, rp);}
    ;

aggregate_tail :
      ( COMMA aggregate_entry )*
    ;

aggregate_entry :
```

```
      choices ( EQUAL_GREATER expression )?
    ;

conditional_waveforms :
      concurrent_waveform
       ( WHEN condition ( ELSE conditional_waveforms )? )?
    ;

selected_signal_assignment_statement :
      WITH expression SELECT target LESS_EQUAL
          optionss selected_waveforms SEMI_COLON
    ;

//modif3
selected_waveforms !:
      swh:selected_waveforms_head swt:selected_waveforms_tail
      {#selected_waveforms =
           #([IDENT2, "selected_waveforms_root"], swh, swt);
      }
    ;

//modif3
selected_waveforms_head :
      concurrent_waveform WHEN choices
    ;

selected_waveforms_tail :
      ( COMMA selected_waveforms_head )*
    ;

optionss :
      ( GUARDED )? ( delay_mechanism )?
    ;

delay_mechanism :
      TRANSPORT
    | ( REJECT time_expression )? INERTIAL
    ;

generate_statement :
      generate_scheme GENERATE^ ( ( block_declarative_part BEGIN )
```

```
          => block_declarative_part BEGIN )? architecture_statement_part
             END GENERATE ( simple_name )? SEMI_COLON
    ;

generate_scheme :
      FOR^ identifier IN discrete_range
    | IF condition
    ;

sequence_of_statements :
      ( BEGIN )? ( sequential_statement )*
    ;

sequential_statement !:
        {AST sl=#([IDENTIFIER,"nolabel"]);}
        ( sl:stmt_label )? ss:sequential_stmt sc:SEMI_COLON
       { #sequential_statement=#([IDENT20,"se_st"],sl,ss,sc);}
    ;

sequential_stmt :
      wait_statement
    | assertion_statement
    | report_statement
    | ( NULL ) => null_statement
    | ( target LESS_EQUAL ) => sa:signal_assignment_statement
    | ( target COLON_EQUAL ) => variable_assignment_statement
    | ( complex_name SEMI_COLON ) => procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | break_statement
    ;

stmt_label :
      vhdl_label
    ;

wait_statement :
```

```
      WAIT ( sensitivity_clause )?
      ( condition_clause )? ( timeout_clause )?
    ;

sensitivity_clause :
      ON sensitivity_list
    ;

sensitivity_list !:
      cn:complex_name slt:sensitivity_list_tail
      {#sensitivity_list =
          #([IDENT3, "sensitivity_list_root"], cn, slt);
      }
    ;

sensitivity_list_tail :
      ( COMMA complex_name )*
    ;

condition_clause :
      UNTIL condition
    ;

condition :
      boolean_expression
    ;

timeout_clause :
      FOR time_expression
    ;

assertion_statement :
      assertion
    ;

assertion :
      ASSERT condition ( REPORT expression )?
     ( SEVERITY expression )?
    ;

report_statement :
```

```
      REPORT expression ( SEVERITY expression )?
    ;

null_statement :
      NULL
    ;

signal_assignment_statement ! :
      ta:target le:LESS_EQUAL ss:sequential_signal_assign_stmt
     {#signal_assignment_statement=
           #([IDENT18, "sequential_signal_assign_st"],ta,le,ss);
     }
    ;


variable_assignment_statement :
      target COLON_EQUAL^ variable_assign_stmt
    ;

procedure_call_statement :
      complex_name
    ;

sequential_signal_assign_stmt :
      ( delay_mechanism )? sequential_waveform
    ;

variable_assign_stmt :
      expression
    ;

sequential_waveform !:
      we:waveform_element swt:sequential_waveform_tail
         {
           #sequential_waveform =
            #(  [IDENT4, "sequential_waveform_root"]
               , we, swt
             );
         }
    ;
```

```
sequential_waveform_tail :
      ( COMMA waveform_element )*
    ;

concurrent_waveform :
      sequential_waveform
    | UNAFFECTED
    ;

waveform_element !:
      exp1:expression ( af:AFTER exp2:expression )?
     {#waveform_element=#([IDENT19,"wave_element"],exp1,af,exp2);}
    ;

if_statement :
      IF condition THEN sequence_of_statements
      ( ( ELSIF ) => elsif_stmt )?
      ( ELSE sequence_of_statements )?
        END IF ( simple_name )?
    ;

elsif_stmt :
      ELSIF condition THEN sequence_of_statements
        ( ( ELSIF ) => elsif_stmt )?
    ;

case_statement :
      CASE expression IS ( case_statement_alternative )+
       END CASE ( simple_name )?
    ;

case_statement_alternative :
      WHEN choices EQUAL_GREATER sequence_of_statements
    ;

choices :
      choice (CHOICE1 choice )*
    | OTHERS
    ;

choice :
```

```
      expression ( direction simple_expression )?
    ;

loop_statement :
      ( FOR ) => for_loop_statement
    | while_loop_statement
    ;

for_loop_statement :
      FOR identifier IN discrete_range LOOP sequence_of_statements
         END LOOP ( simple_name )?
    ;

while_loop_statement :
      ( WHILE condition )? LOOP sequence_of_statements
         END LOOP ( simple_name )?
    ;

next_statement :
      NEXT ( vhdl_label )? ( WHEN condition )?
    ;

exit_statement :
      EXIT ( vhdl_label )? ( WHEN condition )?
    ;

return_statement :
      RETURN ( expression )?
    ;

package_declaration :
      PACKAGE^ identifier IS package_declarative_part END
     ( PACKAGE )? ( simple_name )? SEMI_COLON
    ;

package_declarative_part :
      ( package_declarative_item )*
    ;

package_declarative_item :
      subprogram_declaration
```

```
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declarative_item
    | disconnection_specification
    | use_clause
    | ( NATURE | SUBNATURE | TERMINAL )
        => ( nature_declaration | subnature_declaration
             | terminal_declaration
           )
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

package_body :
      PACKAGE BODY^ identifier IS package_body_declarative_part
      END ( PACKAGE BODY )? ( simple_name )? SEMI_COLON
    ;

package_body_declarative_part :
      ( package_body_declarative_item )*
    ;

package_body_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | use_clause
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
```

```
    ;

subprogram :
      subprogram_header ( subprogram_body )? SEMI_COLON
    ;

subprogram_declaration :
      subprogram_header SEMI_COLON
    ;

subprogram_header :
      PROCEDURE^ designator ( L_PAREN formal_parameter_list R_PAREN )?
    | ( side_effects )? FUNCTION^ designator
      ( L_PAREN formal_parameter_list R_PAREN )? RETURN type_mark
    ;

side_effects :
      PURE
    | IMPURE
    ;

designator :
      operator_symbol
    | identifier
    ;

operator_symbol :
      STRING_LITERAL
    ;

formal_parameter_list :
      interface_list
    ;

subprogram_body :
      IS subprogram_declarative_part sequence_of_statements END
       ( FUNCTION | PROCEDURE )? ( designator )?
    ;

subprogram_declarative_part :
      ( subprogram_declarative_item )*
```

```
    ;

subprogram_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declarative_item
    | use_clause
    ;

signature :
//      L_BRACKET ( type_mark ( COMMA type_mark )* )?
       ( RETURN type_mark )? R_BRACKET
      L_BRACKET ( type_mark ( COMMA type_mark )* )?
      ( RETURN type_mark )? R_BRACKET
    ;

attribute_declarative_item :
      ATTRIBUTE^ ( attribute_declaration_tail
                 | attribute_specification_tail
                )
    ;

attribute_declaration_tail :
      identifier COLON^ type_mark SEMI_COLON
    ;

attribute_specification_tail :
      identifier OF entity_specification IS
       expression SEMI_COLON
    ;

attribute_specification :
      ATTRIBUTE^ attribute_specification_tail
    ;

entity_specification :
```

```
      entity_name_list COLON entity_class
    ;

entity_name_list :!
      enla:entity_name_list_head enlt:entity_name_list_tail
      {#entity_name_list =
           #([IDENT5, "entity_name_list_root"], enla, enlt);}
    | OTHERS
    | ALL
    ;

entity_name_list_head:
      entity_designator ( signature )?
    ;

entity_name_list_tail :
      ( COMMA entity_name_list_head)*
    ;

entity_designator :
      simple_name
    | operator_symbol
    ;

entity_class :
      ENTITY
    | ARCHITECTURE
    | CONFIGURATION
    | PROCEDURE
    | FUNCTION
    | PACKAGE
    | TYPE
    | SUBTYPE
    | CONSTANT
    | SIGNAL
    | VARIABLE
    | COMPONENT
    | LABEL
    | LITERAL
    | UNITS
    | GROUP
```

```
    | FILE
    | ( NATURE | SUBNATURE | QUANTITY | TERMINAL )
    ;

configuration_specification :
      FOR^ component_specification binding_indication SEMI_COLON
    ;

component_specification :
      instantiation_list COLON^ complex_name
    ;

instantiation_list :!
      sn:simple_name ilt:instantiation_list_tail
            {
                #instantiation_list =
                   #([IDENT6, "instantiation_list_root"], sn, ilt
                );
            }
    | OTHERS
    | ALL
    ;

instantiation_list_tail :
      ( COMMA simple_name )*
    ;

binding_indication :
      ( USE entity_aspect )?
      ( generic_map_aspect )?
      ( port_map_aspect )?
    ;

entity_aspect :
      ENTITY^ selected_or_simple_name
      ( L_PAREN simple_name R_PAREN )?
    | CONFIGURATION^ complex_name
    | OPEN
    ;

generic_map_aspect :
```

```
      GENERIC^ MAP L_PAREN association_list_in R_PAREN
    ;

port_map_aspect :
      PORT^ MAP L_PAREN association_list_in R_PAREN
    ;

disconnection_specification :
      DISCONNECT guarded_signal_specification
      AFTER expression SEMI_COLON
    ;

guarded_signal_specification :
      signal_list COLON^ type_mark
    ;

signal_list :!
      cn:complex_name slt:signal_list_tail
            {
                #signal_list =
                   #([IDENT7, "signal_list_root"], cn, slt
                     );
            }
    | OTHERS
    | ALL
    ;

signal_list_tail :
      ( COMMA complex_name )*
    ;

use_clause :
      USE^ complex_name ( COMMA selected_name )*
       SEMI_COLON
    ;

scalar_type_definition :
      enumeration_type_definition
    | range_constraint ( physical_type_definition )?
    ;
```

```
enumeration_type_definition !:
      lp:L_PAREN el:enumeration_literal
          etdt:enumeration_type_definition_tail rp:R_PAREN
            {
                #enumeration_type_definition =
                    #([IDENT8, "enumeration_type_definition_root"]
                       , lp, el, etdt, rp
                      );
            }
    ;

enumeration_type_definition_tail :
      ( COMMA enumeration_literal )*
    ;

enumeration_literal :
      identifier
    | CHARACTER_LITERAL
    ;

range_constraint :
      RANGE^ range
    ;

range :
      ( ( selected_name QUOTE ( attribute_range
            | attribute_reverse_range ) ) )
            => ( ( selected_name QUOTE
              ( attribute_range | attribute_reverse_range ) ) )
    | simple_expression direction simple_expression
    ;

physical_type_definition :
      UNITS base_unit_declaration ( secondary_unit_declaration )*
      END UNITS ( simple_name )?
    ;

base_unit_declaration :
      identifier SEMI_COLON
    ;
```

```
secondary_unit_declaration :
      identifier EQUAL physical_literal SEMI_COLON
    ;

physical_literal :
      ( integer_literal )? simple_name
    ;

abstract_literal :
      integer_literal
//    | real_literal
    ;

composite_type_definition :
      array_type_definition
    | record_type_definition
    ;

//modif2
array_type_definition :
      ARRAY^ array_type_definition_head OF subtype_indication
    ;

//modif2
array_type_definition_head !:
      lp:L_PAREN ad:array_dimension
        atdat:array_type_definition_head_tail rp:R_PAREN
            {
                #array_type_definition_head =
                    #([IDENT9, "array_type_definition_head_root"]
                    , lp, ad, atdat, rp);
            }
    ;

array_type_definition_head_tail :
      ( COMMA array_dimension )*
    ;

array_dimension :
      ( index_subtype_definition )
        => ( index_subtype_definition )
```

```
    | ( discrete_range )
    ;

index_subtype_definition :
      type_mark RANGE LESS_GREATER
    ;

record_type_definition :
      RECORD^ ( element_declaration )+ END RECORD ( simple_name )?
    ;

element_declaration :
      identifier_list COLON^ subtype_indication SEMI_COLON
    ;

access_type_definition :
      ACCESS subtype_indication
    ;

file_type_definition :
      FILE OF type_mark
    ;

type_declaration :
      TYPE^ identifier ( IS type_definition )? SEMI_COLON
    ;

type_definition :
      scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition
    | ( PROTECTED BODY )
        => protected_type_body
    | protected_type_declaration
    ;

subtype_declaration :
      SUBTYPE^ identifier IS subtype_indication SEMI_COLON
    ;
```

```
subtype_indication :
         ( ( complex_name type_mark )
               => complex_name type_mark ( constraint )?
         | type_mark ( constraint )? ) ( ( TOLERANCE )
               => TOLERANCE expression )?
   ;

selected_or_simple_name :
      simple_name ( DOT simple_name )*
    ;

type_mark :
      selected_or_simple_name
    ;

constraint :
      range_constraint
    | index_constraint
    ;

index_constraint !:
      lp:L_PAREN dr:discrete_range
       ict:index_constraint_tail rp:R_PAREN
      {#index_constraint =
           #([IDENT10, "index_constraint_root"], lp, dr, ict, rp);
      }
    ;

index_constraint_tail :
      ( COMMA discrete_range )*
    ;

constant_declaration :
      CONSTANT^ identifier_list COLON subtype_indication
      ( initialization )? SEMI_COLON
    ;

identifier_list !:
      i:identifier ilt:identifier_list_tail
      {#identifier_list =
         #([IDENT11, "identifier_list_root"], i, ilt);
```

```
      }
    ;

identifier_list1 !:
       i:identifier ilt:identifier_list_tail
       {#identifier_list1 = #([IDENT11, "identifier_list_root"]
                , i, ilt);
        }
     ;



identifier_list_tail :
      ( COMMA identifier )*
    ;

initialization :
      COLON_EQUAL expression
    ;

signal_declaration :
      SIGNAL^ identifier_list COLON
          subtype_indication ( signal_kind )?
        ( initialization )? SEMI_COLON
    ;

signal_kind :
      REGISTER
    | BUS
    ;

shared_variable_declaration :
      SHARED VARIABLE variable_declaration_body
    ;

variable_declaration :
      ( SHARED )? VARIABLE variable_declaration_body
    ;

variable_declaration_body :
      identifier_list COLON^ subtype_indication
```

```
                ( initialization )? SEMI_COLON
    ;

file_declaration :
      FILE^ identifier_list COLON subtype_indication
           ( file_open_information )? SEMI_COLON
    ;

file_open_information :
      ( OPEN expression )? IS file_logical_name
    ;

mode :
      IN
    | OUT
    | INOUT
    | BUFFER
    | LINKAGE
    ;

file_logical_name :
      expression
    ;

interface_declaration :
//      (   ( ( CONSTANT | SIGNAL | VARIABLE ) )?
//           identifier_list COLON^ ( mode )? subtype_indication
//                ( BUS )? ( initialization )?
      (   ( ( CONSTANT | SIGNAL | VARIABLE ) )? identifier_list
            COLON ( mode )? subtype_indication ( BUS )?
          (   initialization )?
        | FILE identifier_list COLON subtype_indication
        | ( TERMINAL identifier_list COLON subnature_indication
          | QUANTITY identifier_list COLON ( IN | OUT )?
            subtype_indication ( initialization )?
        )
      )
    ;

interface_list :
      interface_declaration ( SEMI_COLON interface_declaration )*
```

```
    ;

association_list_in !:
      a:association alit:association_list_in_tail
            {
                #association_list_in =
                       #([IDENT12, "association_list_in_root"], a, alit);
            }
    ;

association_list_in_tail :
      ( COMMA association )*
    ;

association :
      association_element ( EQUAL_GREATER association_element )?
    ;

association_element :
      expression
    | OPEN
    ;

alias_declaration :
      ALIAS^ alias_designator ( COLON alias_indication )?
       IS complex_name ( signature )? SEMI_COLON
    ;

alias_designator :
      identifier
    | CHARACTER_LITERAL
    | operator_symbol
    ;

alias_indication :
      ( type_mark TOLERANCE )
        => subnature_indication
    | subtype_indication
    ;

component_declaration :
```

```
      COMPONENT^ identifier ( IS )?
          ( generic_clause )? ( port_clause )? END COMPONENT
      ( simple_name )? SEMI_COLON
    ;

generic_clause :
      GENERIC^ L_PAREN generic_list R_PAREN SEMI_COLON
    ;

port_clause :
      PORT^ L_PAREN port_list R_PAREN SEMI_COLON
    ;

generic_list :
      interface_list
    ;

port_list :
      interface_list
    ;

group_declaration :
      GROUP^ identifier COLON selected_or_simple_name
      L_PAREN group_constituent_list R_PAREN SEMI_COLON
    ;

group_template_declaration :
      GROUP identifier IS L_PAREN entity_class_entry_list R_PAREN
    ;

entity_class_entry_list !:
      ece:entity_class_entry ecelt:entity_class_entry_list_tail
      {#entity_class_entry_list =
               #( [IDENT13, "entity_class_entry_list_root"]
                  , ece, ecelt
                );
      }
    ;

entity_class_entry_list_tail :
      ( COMMA entity_class_entry )*
```

```
    ;

entity_class_entry :
      entity_class ( LESS_GREATER )?
    ;

group_constituent_list !:
      gc:group_constituent
      gclt:group_constituent_list_tail
      {#group_constituent_list =
             #([IDENT14, "group_constituent_list_root"], gc, gclt);
      }
    ;

group_constituent_list_tail :
      ( COMMA group_constituent )*
    ;

group_constituent :
      ( CHARACTER_LITERAL )
        => CHARACTER_LITERAL
    | complex_name
    ;

boolean_expression :
      expression
    ;

time_expression :
      expression
    ;

expression :
      relation expression_tail
    ;

expression_tail :
      (
      //( and_or_xor_xnor ) =>
                 // warning syntactic predicate superflous
      and_or_xor_xnor relation )+
```

```
    | ( ( nand_nor )
        => nand_nor relation )?
    ;

relation :
      shift_expression ( ( relational_operator )
        => relational_operator shift_expression )?
    ;

shift_expression :
      simple_expression ( ( shift_operator )
        => shift_operator simple_expression )?
    ;

simple_expression :
     ( sign_operator )? term (
     // ( adding_operator ) =>
                // warning syntactic predicate superflous
     adding_operator term )*
    ;

term :
      factor (
      //( multiplying_operator ) =>
                // warning syntactic predicate superflous
      multiplying_operator factor )*
    ;

factor :
      primary ( (EXPONENT primary)
        => ( EXPONENT primary ) )?
      | abs_not primary
    ;

numeric_literal :
      abstract_literal ( ( IDENTIFIER ) => complex_name )?
    ;

literal :
      numeric_literal
    | CHARACTER_LITERAL
```

```
    | STRING_LITERAL
    | BIT_STRING_LITERAL
    | NULL
    ;

primary :
      ( STRING_LITERAL L_PAREN )
        => complex_name
    | ( literal )
        => ( literal )
    | complex_name
    | allocator
    | aggregate
    ;

association_list_out !:
      ea:element_association
            alot:association_list_out_tail
      {#association_list_out =
           #([IDENT15, "association_list_out_root"], ea, alot);
      }
    ;

association_list_out_tail :
      ( COMMA element_association )*
    ;

allocator :
      ( NEW subtype_indication )
        => ( NEW subtype_indication )
    | ( NEW complex_name )
    ;

element_association :
      ( choices EQUAL_GREATER expression )
        => ( choices EQUAL_GREATER expression )
    | association
    ;

complex_name :
      attribute_name
```

```
    ;

attribute_name :
      selected_name ( QUOTE attribute )*
    ;

attribute :
      ( selected_name )
        => attribute_user_defined
    | attribute_active
    | attribute_ascending
    | attribute_base
    | attribute_delayed
    | attribute_driving_value
    | attribute_driving
    | attribute_event
    | attribute_high
    | attribute_image
    | attribute_instance_name
    | attribute_last_active
    | attribute_last_event
    | attribute_last_value
    | attribute_leftof
    | attribute_left
    | attribute_length
    | attribute_low
    | attribute_path_name
    | attribute_pos
    | attribute_pred
    | attribute_range
    | attribute_rightof
    | attribute_right
    | attribute_reverse_range
    | attribute_simple_name
    | attribute_stable
    | attribute_succ
    | attribute_transaction
    | attribute_quiet
    | attribute_value
    | attribute_val
    | ( aggregate )
```

```
    ;

attribute_user_defined :
      selected_name
    ;

attribute_active :
      ACTIVE
    ;

attribute_ascending :
      ASCENDING ( ( L_PAREN )
        => L_PAREN expression R_PAREN )?
    ;

attribute_base :
      BASE
    ;

attribute_delayed :
      DELAYED ( ( L_PAREN )
        => L_PAREN expression R_PAREN )?
    ;

attribute_driving_value :
      DRIVING_VALUE
    ;

attribute_driving :
      DRIVING
    ;

attribute_event :
      EVENT
    ;

attribute_high :
      HIGH ( ( L_PAREN )
        => L_PAREN expression R_PAREN )?
    ;
```

```
attribute_image :
      IMAGE L_PAREN expression R_PAREN
    ;

attribute_instance_name :
      INSTANCE_NAME
    ;

attribute_last_active :
      LAST_ACTIVE
    ;

attribute_last_event :
      LAST_EVENT
    ;

attribute_last_value :
      LAST_VALUE
    ;

attribute_leftof :
      LEFTOF^ L_PAREN expression R_PAREN
    ;

attribute_left :
      LEFT^ ( ( L_PAREN )
        => L_PAREN expression R_PAREN )?
    ;

attribute_length :
      LENGTH^ ( ( L_PAREN )
        => L_PAREN expression R_PAREN )?
    ;

attribute_low :
      LOW^ ( ( L_PAREN )
        => L_PAREN expression R_PAREN )?
    ;

attribute_path_name :
      PATH_NAME
```

```
    ;

attribute_pos :
      POS^ L_PAREN expression R_PAREN
    ;

attribute_pred :
      PRED^ L_PAREN expression R_PAREN
    ;

attribute_range :
      RANGE ( ( L_PAREN )
          => L_PAREN expression R_PAREN )?
    ;

attribute_rightof :
      RIGHTOF^ L_PAREN expression R_PAREN
    ;

attribute_right :
      RIGHT^ ( ( L_PAREN )
            => L_PAREN expression R_PAREN )?
    ;

attribute_reverse_range :
      REVERSE_RANGE ( ( L_PAREN )
          => L_PAREN expression R_PAREN )?
    ;

attribute_simple_name :
      SIMPLE_NAME
    ;

attribute_stable :
      STABLE^ ( ( L_PAREN )
        => L_PAREN expression R_PAREN )?
    ;

attribute_succ :
      SUCC^ L_PAREN expression R_PAREN
    ;
```

```
attribute_transaction :
      TRANSACTION
    ;

attribute_quiet :
      QUIET^ ( ( L_PAREN )
        => L_PAREN expression R_PAREN )?
    ;

attribute_value :
      VALUE^ ( ( L_PAREN )
        => L_PAREN expression R_PAREN )?
    ;

attribute_val :
      VAL^ L_PAREN expression R_PAREN
    ;

selected_name :
//      indexed_name ( DOT^ ( indexed_name
                        | ALL ( L_PAREN index_specifier R_PAREN )* )
                      )*
      indexed_name ( DOT ( indexed_name
                         | ALL ( L_PAREN index_specifier R_PAREN )* )
                      )*
    ;

indexed_name :
      name ( L_PAREN index_specifier R_PAREN )*
    ;

index_specifier :
      ( expression direction )
        => discrete_range ( EQUAL_GREATER expression )?
    | association_list_out
    ;

slice_specifier :
      discrete_range
    ;
```

```
name :
      simple_name
    | STRING_LITERAL
    | CHARACTER_LITERAL
    ;

simple_name :
      identifier
    ;

alpha_literals :
      BIT_STRING_LITERAL
    | CHARACTER_LITERAL
    | STRING_LITERAL
    | NULL
    ;

physical_type_name :
      identifier
    ;

discrete_range :
      ( range )
        => ( range )
    | subtype_indication
    ;

direction :
      TO
    | DOWNTO
    ;

vhdl_label :
      identifier
    ;
integer_literal :
      DECIMAL_INTEGER_LITERAL
    | BASED_INTEGER_LITERAL
    ;
```

```
// real_literal :
//      decimal_floating_point_literal
//    | based_floating_point_literal
//     ;

identifier :
      IDENTIFIER
    ;

nature_declaration :
      NATURE^ identifier IS nature_definition SEMI_COLON
    ;

nature_definition :
      scalar_nature_definition
    | array_nature_definition
    | record_nature_definition
    ;

terminal_declaration :
      TERMINAL^ identifier_list COLON subnature_indication SEMI_COLON
    ;

quantity_declaration :
      QUANTITY^ quantity_declaration_tail SEMI_COLON
    ;

quantity_declaration_tail :
//      ( identifier_list COLON )
//           => identifier_list COLON^ subtype_indication
                 ( source_aspect | initialization )?
      ( identifier_list COLON )
          => identifier_list COLON subtype_indication
                       ( source_aspect | initialization )?
    | ( identifier_list (   TOLERANCE
                          | COLON_EQUAL
                          | ACROSS
                          | THROUGH
                        )
       )
          => across_through_aspect_body
```

```
            (  ACROSS ( across_through_aspect_body THROUGH )?
              | THROUGH
            )
                 terminal_aspect
    | terminal_aspect
    ;

across_through_aspect_body :
      identifier_list ( TOLERANCE expression )?
      ( initialization )?
    ;

terminal_aspect :
      complex_name ( TO complex_name )?
    ;

source_aspect :
      SPECTRUM^ simple_expression COMMA^ simple_expression
    | NOISE simple_expression
    ;

//modif1
step_limit_specification :
      LIMIT^ step_limit_specification_head COLON
       type_mark WITH expression SEMI_COLON
    ;

//modif1
step_limit_specification_head :!
   sosn:selected_or_simple_name slsat:step_limit_specification_head_tail
    {
       #step_limit_specification_head =
          #([IDENT16, "step_limit_specification_head_root"], sosn, slsat);
    }
    | OTHERS
    | ALL
    ;

step_limit_specification_head_tail :
      ( COMMA selected_or_simple_name )*
    ;
```

```
scalar_nature_definition :
      type_mark ACROSS type_mark THROUGH identifier REFERENCE
    ;

//modif4
array_nature_definition :
      ARRAY^ array_nature_definition_head OF subnature_indication
    ;

//modif4
array_nature_definition_head !:
      lp:L_PAREN ad:array_dimension
       andat:array_nature_definition_head_tail rp:R_PAREN
      {
        #array_nature_definition_head =
            #([IDENT17, "array_nature_definition_head_root"]
             , lp, ad, andat, rp
            );
      }
    ;

array_nature_definition_head_tail :
      ( COMMA array_dimension )*
    ;

record_nature_definition :
      RECORD^ ( nature_element_declaration )+
      END RECORD ( simple_name )?
    ;

nature_element_declaration :
      identifier_list COLON^ subnature_indication SEMI_COLON
    ;

subnature_declaration :
      SUBNATURE^ identifier IS subnature_indication SEMI_COLON
    ;

subnature_indication :
      type_mark ( index_constraint )?
```

```
        ( TOLERANCE expression ACROSS expression THROUGH )?
    ;

concurrent_break_statement :
//      BREAK^ ( break_element ( COMMA^ break_element )* )?
//     ( ON^ complex_name ( COMMA^ complex_name )* )?
//       ( WHEN expression )? SEMI_COLON
    BREAK ( break_element ( COMMA break_element )* )?
   ( ON complex_name ( COMMA complex_name )* )?
    ( WHEN expression )? SEMI_COLON
    ;

set_of_simultaneous_statements :
      ( concurrent_statement )*
    ;

simultaneous_statement :
      ( stmt_label! )? simultaneous_stmt
    ;

simultaneous_stmt :
      simple_simultaneous_statement
    | simultaneous_if_statement
    | simultaneous_case_statement
    | simultaneous_procedural_statement
    | simultaneous_null_statement
    ;

simple_simultaneous_statement :
      simple_expression EQUAL_EQUAL simple_expression
        ( TOLERANCE expression )? SEMI_COLON
    ;

simultaneous_if_statement :
      IF condition USE set_of_simultaneous_statements
      ( ELSIF condition USE set_of_simultaneous_statements )*
      ( ELSE set_of_simultaneous_statements )?
         END USE ( simple_name )? SEMI_COLON
    ;

simultaneous_case_statement :
```

```
      CASE expression USE
      ( WHEN choices EQUAL_GREATER set_of_simultaneous_statements )+
      END CASE ( simple_name )? SEMI_COLON
    ;

simultaneous_procedural_statement :
      PROCEDURAL^ ( IS )? ( procedural_declarative_item )*
     sequence_of_statements
      END PROCEDURAL ( simple_name )? SEMI_COLON
    ;

procedural_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | alias_declaration
    | attribute_declarative_item
    | use_clause
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

simultaneous_null_statement :
             NULL SEMI_COLON
    ;

break_statement :
//      BREAK ( break_element ( COMMA^ break_element )* )?
          ( WHEN expression )? SEMI_COLON
    BREAK ( break_element ( COMMA break_element )* )?
          ( WHEN expression )? SEMI_COLON
    ;

break_element :
      ( FOR complex_name USE )? complex_name
              EQUAL_GREATER expression
    ;
```

```
protected_type_declaration :
      PROTECTED^ ( protected_type_declarative_item )*
              END PROTECTED ( simple_name )?
    ;

protected_type_declarative_item :
      subprogram_declaration
    | attribute_specification
    | use_clause
    ;

protected_type_body :
      PROTECTED BODY^ ( protected_type_body_declarative_item )*
        END PROTECTED BODY ( simple_name )?
    ;

protected_type_body_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declarative_item
    | use_clause
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

and_or_xor_xnor :
       AND
     | OR
     | XOR
     | XNOR
     ;

nand_nor :
      NAND
    | NOR
```

```
    ;

relational_operator :  EQUAL | NOT_EQUAL | LESS | LESS_EQUAL
                            | GREATER | GREATER_EQUAL ;

shift_operator  : SLL | SRL | SLA | SRA | ROL | ROR ;

sign_operator  : PLUS | MINUS ;

adding_operator : PLUS | MINUS | AMPERSAND ;

multiplying_operator : MULTIPLY | DIVIDE | MOD | REM ;

abs_not : ABS | NOT ;


class bhdlLexer extends Lexer ;
options {
    exportVocab     =   vhdl    ; // Call its vocabulary "vhdl"
    charVocabulary = '\0'..'\377';
    k=9;
    testLiterals=true ;
    caseSensitive=false ;
    caseSensitiveLiterals=false ;
        }
// A partir du lexer de Savant


L_PAREN       : '(' ;
EQUAL_EQUAL   : "==" ;
AMPERSAND     : '&' ;
R_PAREN       : ')' ;
L_BRACKET     : '[' ;
R_BRACKET     : ']' ;
COMMA         : ',' ;
COLON_EQUAL   : ":=" ;
COLON         : ':' ;
SEMI_COLON    : ';' ;
LESS_GREATER  : "<>" ;
LESS_EQUAL    : "<=" ;
LESS          : "<" ;
```

```
EQUAL_GREATER : "=>" ;
EQUAL         : '=' ;
GREATER_EQUAL : ">=" ;
GREATER       : '>' ;
CHOICE1       : '|' ;
// CHOICE2       : '!' ;
NOT_EQUAL     : "/=" ;
DOT           : '.' ;
EXPONENT      : "**" ;
MULTIPLY      : '*'  ;
DIVIDE        : '/' ;
PLUS          : '+' ;
MINUS         : '-' ;
UNDERSCORE    : '_' ;

BIT_STRING_LITERAL
  : 'b' '"' ('0' | '1') ( ('_')? ('0' | '1') ) '"'
  | 'b' '%' ('0' | '1') ( ('_')? ('0' | '1') ) '%'
  | 'o' '"' ('0'..'7') ( ('_')? ('0'..'7') ) '"'
  | 'o' '%' ('0'..'7') ( ('_')? ('0'..'7') ) '%'
  | 'x' '"' ('0'..'9' | 'a'..'f') ( ('_')?
                ('0'..'9' | 'a'..'f') ) '"'
  | 'x' '%' ('0'..'9' | 'a'..'f') ( ('_')?
                ('0'..'9' | 'a'..'f') ) '%' ;

STRING_LITERAL :
           '"'
             ((~('\n'|'\r'|'"')))+
           '"'
           { $setType(STRING_LITERAL); };

protected
QUOTE : '\''  ;

protected
CHARACTER_LITERAL :
           '\'' (~('\n'|'\r')) '\''
           ;

CHARACTER_LITERAL_OR_QUOTE :
          ( '\'' (~('\n'|'\r')) '\'') => CHARACTER_LITERAL
```

```
                    { $setType(CHARACTER_LITERAL) ;}
          | ('\'' ) => QUOTE { $setType(QUOTE) ;}
  ;

WS
  : ( ' ' |
      '\t'
    | '\f'
    | ( "\r\n"
      | '\n'
      )
      { newline(); }
    )
    { $setType(Token.SKIP); }

  ;



protected
DIGIT : '0'..'9' ;

protected
LETTER :   'a'..'z' ;

IDENTIFIER
options {testLiterals=true ; }
  : LETTER ( LETTER | DIGIT | '_' )*
  ;

//COMMENT
//  : "--" (~('\n'|'\r'))*
//    { $setType(Token.SKIP); }
//  ;

BASED_INTEGER_LITERAL_OR_DECIMAL_INTEGER_LITERAL :
      (INTEGER '#') => BASED_INTEGER_LITERAL
                       { $setType(BASED_INTEGER_LITERAL) ;}
    | (INTEGER) =>  DECIMAL_INTEGER_LITERAL
                       { $setType(DECIMAL_INTEGER_LITERAL) ;}
;
```

```
protected
DECIMAL_INTEGER_LITERAL : INTEGER ( DOT INTEGER )? ( EXP )? ;

protected
BASED_INTEGER_LITERAL :
   INTEGER '#' BASED_INTEGER ( DOT BASED_INTEGER )? '#' ( EXP )?
 ;

protected
INTEGER : DIGIT (( UNDERSCORE )? DIGIT)*  ;

protected
BASED_INTEGER
:  (DIGIT | LETTER) ( ( UNDERSCORE )? (DIGIT | LETTER) )*
;

protected
EXP
: ('e') (PLUS|MINUS)? INTEGER
;

//protected
//BCOMMENT_DEF
//: "-- #d"! (~('\n'|'\r'))*
//| "--" (~('\n'|'\r'))*   { $setType(Token.SKIP); }
//;


// | "-- #d"! (~('\n'|'\r'))*  { $setType(BCOMMENT_DEF); }
COMMENT
: "-- #i"    { $setType(BCOMMENT_INV); }
| "-- #d"    { $setType(BCOMMENT_DEF); }
| "-- #="!   (~('\n'|'\r'))*  { $setType(BCOMMENT_E); }
| "--" (~('\n'|'\r'))*   { $setType(Token.SKIP); }
;
```

Annexe C

Parseur d'Arbre VHDL

```
// VHDL2B Tree parser
//
//        This ANTLR parser can walk the VHDL AST and transforms
//        a part of it to the associated B one
//
//        Authors :
//        Ammar Aljer, Philippe Devienne, Pierre Antoine Laloux
//        Current Version : Dec 2003
//

{
import java.io.*;
import antlr.collections.AST;
}

class treewalker extends TreeParser ;
options {
    importVocab=vhdl;               // Call its vocabulary "vhdl"
    defaultErrorHandler = true ;
    buildAST=true;
  //  codeGenMakeSwitchThreshold = 3;
  //  codeGenBitsetTestThreshold = 4;
    k=1;
         }
  {
  AST entity_comment;
  PrintWriter amfile = null;
  String comment1;
```

```
  String comment2;
  }
design_file :
{
try{
 amfile = new PrintWriter(
 new BufferedWriter(new FileWriter("./B/source" + ".vhdl")));
  }catch(java.io.IOException e){}
}
      (c:comments! {#entity_comment=#c.getFirstChild();})? design_file1
    {
    amfile.close();
    }
    ;

comments :
      #(IDENTIFIER (#(IDENTIFIER
                     identifier_list1
                     BCOMMENT_E identifier_list1))
                     *)
    ;

design_file1 :
      (design_unit)*
    ;

//context_clauses are not used to generate B code
//context_clauses contain library clause and use clause
design_unit :
      context_clauses!  library_unit
    ;

library_unit :
      primary_unit
    | secondary_unit
    ;

//configuration_declaration and package_declaration
// are not used to generate B code
primary_unit :
      entity_declaration
```

```
    |! configuration_declaration
    |! package_declaration
    ;

secondary_unit :
       architecture_body
    |! package_body
    ;

library_clause! :
      #(LIBRARY logical_library_name_list SEMI_COLON)
    ;

logical_library_name_list :
      #(IDENT logical_name (COMMA logical_name)*)
    ;

logical_name :
      identifier
    ;

context_clauses :
      ( context_item )*
    ;

context_item :
      library_clause
    | use_clause
    ;

entity_declaration! :
      #(en:ENTITY {amfile.print(en.getText()+ " ");}
      ident:identifier {amfile.print(ident.getText()+ " ");}
      is:IS {amfile.println(is.getText());}
      entih:entity_header
      entity_declarative_part
        ( entity_statement_part )?
        e:END {amfile.print(e.getText()+ " ");}
      ( ENTITY )?
      ( simple_name )?
        sc:SEMI_COLON {amfile.println(sc.getText()+ " ");} )
```

```
{   AST tempA;
    AST MachineA;
    AST SeesA;
    AST VariablesA;
    AST DefinitionsA;
    AST InvariantA;
    AST InitialisationA;
    AST OperationsA;
    AST NextA;
    AST ModeA;
    AST TypeA;
    AST SeesSearch;

    AST DefVariablesA;
    AST DefTextA;
    AST InvVariablesA;
    AST TempDVariableA;
    AST TempIVariableA;
    AST OneVariableA;
    AST OneDVariableA;
    AST OneIVariableA;

    String OneVariableS;
    String OneDVariableS;
    String OneIVariableS;

    String DefTempVariable;
    String InvTempVariable;

    String Oper;
    String OperMod;
    String OperType;
    String SearchS;

    #tempA=#([ABSTRACT,"ABSTRACT"]);
    #MachineA=#([IDENTIFIER,"MACHINE"]);
    #SeesA=#([IDENTIFIER,"SEES"]);
    #VariablesA=#([IDENTIFIER,"VARIABLES"]);
    #DefinitionsA=#([IDENTIFIER,"DEFINITIONS"]);
    #InvariantA=#([IDENTIFIER,"INVARIANT"]);
    #InitialisationA=#([IDENTIFIER,"INITIALISATION"]);
```

```
#OperationsA=#([IDENTIFIER,"OPERATIONS"]);



#ident.setNextSibling(null);
#MachineA.setFirstChild(#ident);
if (#entih != null)
 {
   #NextA=#entih.getFirstChild();
   while (#NextA != null)
    {
      #TypeA=#NextA.getFirstChild();
      #ModeA=#TypeA.getNextSibling();

      #SeesSearch=#SeesA.getFirstChild();
      SearchS=TypeA.getText();
      while ( (#SeesSearch != null)  &&
             !(SearchS.equals(#SeesSearch.getText() )  )
             )
       {
         #SeesSearch = #SeesSearch.getNextSibling();
       }
     if (#SeesSearch == null)
      {
       #SeesA.addChild( #( [#TypeA.getType(),
                            #TypeA.getText()] )
                      );
      }

     #VariablesA.addChild( #([#NextA.getType(),
                              #NextA.getText()])
                         );

     #InitialisationA.addChild( #([#NextA.getType(),
                                   #NextA.getText()
                                  ],
                                  [#TypeA.getType(),
                                   #TypeA.getText()
                                  ]
                                 )
                               );
```

```
      #OperationsA.addChild( #( [ #NextA.getType(),
                                  #NextA.getText()
                                ],
                                [ #ModeA.getType(),
                                  #ModeA.getText()
                                ],
                                [ #TypeA.getType(),
                                  #TypeA.getText()
                                ]
                              )
                            );


      #NextA=#NextA.getNextSibling();
    }
  }
AST comment_search;
String search2=#ident.getText();
#comment_search=#entity_comment;
while (       (#comment_search!=null)
           && !(search2.equals(#comment_search.getText()))
       )
  {
   #comment_search=#comment_search.getNextSibling();
  }
if (#comment_search!=null)
  {
   #DefVariablesA=#comment_search.getFirstChild();
   #DefTextA=#DefVariablesA.getNextSibling();
   #InvVariablesA=
        #DefVariablesA.getNextSibling().getNextSibling();
   #DefVariablesA.setNextSibling(null);
   #DefTextA.setNextSibling(null);

   #OneVariableA=#VariablesA.getFirstChild();
   #OneDVariableA=#DefVariablesA.getFirstChild();
   #OneIVariableA=#InvVariablesA.getFirstChild();

   while (#OneVariableA!=null)
    {
      OneVariableS =#OneVariableA.getText();
```

```
    OneDVariableS=#OneDVariableA.getText();
    OneIVariableS=#OneIVariableA.getText();



    if ( !(OneVariableS.equals(OneIVariableS)) )
     {
       #TempDVariableA=#OneDVariableA.getNextSibling();
       #TempIVariableA=#OneIVariableA.getNextSibling();
     while  (   (#TempIVariableA!=null)
       && (!(OneVariableS.equals(#TempIVariableA.getText()))))
       {

          #TempDVariableA=#TempDVariableA.getNextSibling();
          #TempIVariableA=#TempIVariableA.getNextSibling();
        }

       if   (#TempIVariableA!=null)

         {
           DefTempVariable=OneDVariableS;
           InvTempVariable=OneIVariableS;
           #OneDVariableA.setText(#TempDVariableA.getText());
           #OneIVariableA.setText(#TempIVariableA.getText());
           #TempDVariableA.setText(OneDVariableS);
           #TempIVariableA.setText(OneIVariableS);
         }

     }
    #OneVariableA= #OneVariableA.getNextSibling();
    #OneDVariableA=#OneDVariableA.getNextSibling();
    #OneIVariableA=#OneIVariableA.getNextSibling();
  }

 #DefinitionsA.addChild(#DefVariablesA);
 #DefinitionsA.addChild(#DefTextA);
 #InvariantA.addChild( #InvVariablesA);
}

//if there are no comments
//  else
```

```
            //  {
            //  #DefinitionsA=;
            //  #InvariantA=;
            //  }

           #tempA.addChild(#MachineA);
           #tempA.addChild(#SeesA);
           #tempA.addChild(#VariablesA);
           #tempA.addChild(#DefinitionsA);
           #tempA.addChild(#InvariantA);
           #tempA.addChild(#InitialisationA);
           #tempA.addChild(#OperationsA);
           #entity_declaration=#tempA;
          }
    ;

entity_header :
      ( generic_clause! )? ( port_clause )?
    ;

entity_declarative_part :
      ( entity_declarative_item )*
    ;

entity_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declarative_item
    | disconnection_specification
    | use_clause
    | ( LIMIT | NATURE | SUBNATURE | QUANTITY | TERMINAL )
        => ( step_limit_specification
             | nature_declaration
             | subnature_declaration
             | quantity_declaration
```

```
            | terminal_declaration
           )
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

entity_statement_part :
      ( BEGIN )? ( entity_statement )*
    ;

entity_statement :
      ( stmt_label )? ( POSTPONED )? entity_stmt
    ;

entity_stmt :
      concurrent_assertion_statement
    | concurrent_procedure_call
    | process_statement
    ;

concurrent_procedure_call :
      complex_name SEMI_COLON
    ;

architecture_body !:
      #( ar:ARCHITECTURE  {amfile.print(ar.getText()+ " ");}
       i1:identifier {amfile.print(i1.getText()+ " ");}
       of:OF {amfile.print(of.getText()+ " ");}

        sosn:selected_or_simple_name {amfile.print(sosn.getText()+ " ");}
        is:IS {amfile.println(is.getText()+ " ");}

         adp:architecture_declarative_part

         asp:architecture_statement_part

         end:END {amfile.println(); amfile.print( end.getText()+ " ");}
         ( arch:ARCHITECTURE {amfile.print( arch.getText()+ " ");} )?
         ( sn:simple_name )?
         sc:SEMI_COLON {amfile.print( sc.getText()+ " ");}  )
```

```
            { if ( #asp.getFirstChild() != null)
               {
                #adp.setText("IMPORTS");
                #architecture_body=#([CONCRETE,"CONCRETE"],
                                     #([IDENTIFIER,"REFINEMENT"],i1,sosn),
                                     #([IDENTIFIER,"REFINES"],sosn),
                                     adp ,
                                     #(  [IDENTIFIER,"ARC_BODY"]
                                       , asp
                                      )
                                    );
              }
             else
              {
                #adp.setText("IMPORTS");
                #architecture_body=#([CONCRETE,"CONCRETE"],
                                     #([IDENTIFIER,"REFINEMENT"],
                                       i1,sosn
                                       ),
                                     #([IDENTIFIER,"REFINES"]
                                      ,sosn
                                       ),
                                     adp
                                   );
              }
            }

    ;

architecture_declarative_part !:
      bdi:block_declarative_item adp:architecture_declarative_part
      {   AST  tempA;
                           #tempA=#bdi;
                           #adp.addChild(tempA);
                           #architecture_declarative_part=#adp;
      }
    | {
        #architecture_declarative_part=#([IDENTIFIER,"ArcDecPart"]);
      }
    ;
```

```
architecture_statement_part !:
      ( be:BEGIN    {amfile.print(be.getText()+ " ");}
              )?
              {
                #architecture_statement_part
                 = #([IDENTIFIER,"architecture_statement_part"]);
              }
      ( ase:architecture_statement_element
        {if (#ase != null)
                {AST tt;
                 #tt=null;
                 #tt=#([IDENT25,"newcomp"]);
                 #tt.addChild(#ase);
                 #architecture_statement_part.addChild(#tt);}
          }

        )*
    ;

architecture_statement_element! :
      ( CASE | NULL | PROCEDURAL | IF condition USE
        | simple_expression EQUAL_EQUAL
       )
        => simultaneous_statement!
    |  cs:concurrent_statement
          {if (#cs != null)
               {#architecture_statement_element=
                  #cs.getFirstChild();
               }
           }
    ;

configuration_declaration :
      #( CONFIGURATION identifier OF selected_or_simple_name IS
         configuration_declarative_part block_configuration
         END ( CONFIGURATION )? ( simple_name )? SEMI_COLON)
    ;

configuration_declarative_part :
      ( configuration_declarative_item )*
    ;
```

```
configuration_declarative_item :
      use_clause
    | attribute_specification
    | group_declaration
    ;

block_configuration! :
      #(FOR block_specification ( use_clause )*
         ( configuration_item )* END FOR SEMI_COLON
       )
    ;

block_specification :
      selected_or_simple_name ( L_PAREN index_specifier R_PAREN )?
    ;

configuration_item :
      ( component_configuration )
        => ( component_configuration )
    | block_configuration
    ;

component_configuration! :
      #( FOR component_specification
         ( binding_indication SEMI_COLON )?
         ( block_configuration )?
          END FOR SEMI_COLON
       )
    ;

concurrent_statement! :

     #(IDENT21    {amfile.println();}
     ( sl:stmt_label {amfile.print(#sl.getText());})?
     (co:COLON! {amfile.print(#co.getText());})?)

     cs:concurrent_stmt
     {if (sl != null) {#concurrent_statement=#(IDENT21,sl,cs);}}
    ;
```

```
concurrent_stmt :
      ( BLOCK | FOR | IF | BREAK | CASE | NULL | PROCEDURAL
         | simple_expression EQUAL_EQUAL
      )
        => not_postponeable
    |    #(IDENT23 postponeable)
    ;

not_postponeable :
      block_statement
    | ( generate_scheme GENERATE )
        => generate_statement
    | ( IF | CASE | NULL | PROCEDURAL
          | simple_expression EQUAL_EQUAL
       )
        => simultaneous_stmt
    | concurrent_break_statement
    ;

postponeable :
      process_statement
    | concurrent_assertion_statement
    | selected_signal_assignment_statement
    | ( target LESS_EQUAL )
        => conditional_signal_assignment
    |! ( COMPONENT | ENTITY | CONFIGURATION | complex_name
         ( generic_map_aspect | port_map_aspect )
       )
        => is:instantiate_statement
              {#postponeable =#is.getFirstChild();}
    | concurrent_procedure_call
    ;


 commentsr :
      (BCOMMENT_DEF BCOMMENT_INV)?
    ;

block_statement :
      #( BLOCK ( L_PAREN expression R_PAREN )?
         ( IS )? block_header block_declarative_part
```

```
        architecture_statement_part END BLOCK ( simple_name )?
        SEMI_COLON
      )
    ;

block_header! :
      ( generic_clause ( generic_map_aspect SEMI_COLON )? )?
        ( port_clause ( port_map_aspect SEMI_COLON )? )?
    ;

block_declarative_part :
      ( block_declarative_item )*
    ;

block_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | disconnection_specification
    | attribute_declarative_item
    | configuration_specification
    | use_clause
    | ( LIMIT | NATURE | SUBNATURE | QUANTITY | TERMINAL )
        => ( step_limit_specification
             | nature_declaration
             | subnature_declaration
             | quantity_declaration
             | terminal_declaration
           )
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

process_statement :
```

```
      #(PROCESS ( L_PAREN sensitivity_list R_PAREN )? ( IS )?
      process_declarative_part sequence_of_statements
      END ( POSTPONED )? PROCESS ( simple_name )? SEMI_COLON)
    ;

process_declarative_part :
      ( process_declarative_item )*
    ;

process_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declarative_item
    | use_clause
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

concurrent_assertion_statement :
      assertion SEMI_COLON
    ;

concurrent_call_statement :
      complex_name SEMI_COLON
    ;

instantiate_statement :

       #(IDENT24
            (ENTITY!|CONFIGURATION!|( COMPONENT )?)
            cn:complex_name {amfile.print(cn.getText());}
            ( generic_map_aspect! )?
            ( port_map_aspect )?
            SEMI_COLON!
       )
```

```
//    | #(ENTITY complex_name ( generic_map_aspect )?
//         ( port_map_aspect )? SEMI_COLON
//       )
//    | #( complex_name ( generic_map_aspect )?
//          ( port_map_aspect )? SEMI_COLON
//       )
    ;

conditional_signal_assignment :
      target LESS_EQUAL optionss conditional_waveforms SEMI_COLON
    ;

target :
      ( complex_name )
        => ( complex_name )
    | aggregate
    ;


aggregate :
      #( IDENT1 L_PAREN! aggregate_entry
         (COMMA aggregate_entry)* R_PAREN!
       )

    ;

aggregate_entry :
      choices ( EQUAL_GREATER expression )?
    ;

conditional_waveforms :
      concurrent_waveform ( WHEN condition
                             ( ELSE conditional_waveforms )?
                           )?
    ;

selected_signal_assignment_statement :

      WITH expression SELECT target LESS_EQUAL
      optionss selected_waveforms SEMI_COLON
    ;
```

```
selected_waveforms :
      #( IDENT2 selected_waveforms_head
         (COMMA selected_waveforms_head)*
       )
    ;

selected_waveforms_head :
      concurrent_waveform WHEN choices
    ;

optionss :
      ( GUARDED )? ( delay_mechanism )?
    ;

delay_mechanism :
      TRANSPORT
    | ( REJECT time_expression )? INERTIAL
    ;

generate_statement :
      generate_scheme GENERATE
       ( ( block_declarative_part BEGIN )
        => block_declarative_part BEGIN )?
             architecture_statement_part
          END GENERATE ( simple_name )? SEMI_COLON
    ;

generate_scheme :
      #(FOR identifier IN discrete_range)
    | IF condition
    ;

sequence_of_statements :
      ( BEGIN )? ( sequential_statement )*
    ;

sequential_statement :
     #(IDENT20 ( stmt_label )? sequential_stmt SEMI_COLON)
    ;
```

```
sequential_stmt :
      wait_statement
    | assertion_statement
    | report_statement
    | ( NULL )
        => null_statement
    | ( target LESS_EQUAL )
        => signal_assignment_statement
    | ( target COLON_EQUAL )
        => variable_assignment_statement
    | ( complex_name SEMI_COLON )
        => procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | break_statement
    ;

stmt_label :
      vhdl_label
    ;

wait_statement :
      WAIT ( sensitivity_clause )?
      ( condition_clause )? ( timeout_clause )?
    ;

sensitivity_clause :
      ON sensitivity_list
    ;

sensitivity_list :
      #(IDENT3 complex_name (COMMA complex_name)*)
    ;

condition_clause :
      UNTIL condition
    ;
```

```
condition :
      boolean_expression
    ;

timeout_clause! :
      FOR time_expression
    ;

assertion_statement :
      assertion
    ;

assertion :
      ASSERT condition
      ( REPORT expression )?
      ( SEVERITY expression )?
    ;

report_statement :
      REPORT expression ( SEVERITY expression )?
    ;

null_statement :
      NULL
    ;

signal_assignment_statement :
      #(IDENT18 target LESS_EQUAL sequential_signal_assign_stmt)
    ;


variable_assignment_statement :
      #(COLON_EQUAL target variable_assign_stmt)
    ;

procedure_call_statement :
      complex_name
    ;

sequential_signal_assign_stmt :
```

```
      ( delay_mechanism )? sequential_waveform
    ;

variable_assign_stmt :
      expression
    ;

sequential_waveform :
      #(IDENT4 waveform_element (COMMA waveform_element)*)
    ;

concurrent_waveform :
      sequential_waveform
    | UNAFFECTED
    ;

waveform_element :
      #(IDENT19 expression
      ( AFTER expression )?)
    ;

if_statement :
      IF condition THEN sequence_of_statements
       ( ( ELSIF ) => elsif_stmt )?
       ( ELSE sequence_of_statements )?
       END IF ( simple_name )?
    ;

elsif_stmt :
      ELSIF condition THEN sequence_of_statements
      ( ( ELSIF ) => elsif_stmt )?
    ;

case_statement :
      CASE expression IS ( case_statement_alternative )+
        END CASE ( simple_name )?
    ;

case_statement_alternative :
      WHEN choices EQUAL_GREATER sequence_of_statements
    ;
```

```
choices :
      choice (CHOICE1  choice )*
    | OTHERS
    ;

choice :
      expression ( direction simple_expression )?
    ;

loop_statement :
      ( FOR )
        => for_loop_statement
    | while_loop_statement
    ;

for_loop_statement :
      FOR identifier IN discrete_range
      LOOP sequence_of_statements END
      LOOP ( simple_name )?
    ;

while_loop_statement :
      ( WHILE condition )? LOOP sequence_of_statements
      END LOOP ( simple_name )?
    ;

next_statement :
      NEXT ( vhdl_label )? ( WHEN condition )?
    ;

exit_statement :
      EXIT ( vhdl_label )? ( WHEN condition )?
    ;

return_statement :
      RETURN ( expression )?
    ;

package_declaration :
      #(PACKAGE identifier IS package_declarative_part END
```

```
          ( PACKAGE )? ( simple_name )? SEMI_COLON
        )
    ;

package_declarative_part :
      ( package_declarative_item )*
    ;

package_declarative_item :
      subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declarative_item
    | disconnection_specification
    | use_clause
    | ( NATURE | SUBNATURE | TERMINAL )
        => ( nature_declaration | subnature_declaration
             | terminal_declaration
           )
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

package_body :
      #( BODY PACKAGE  identifier IS package_body_declarative_part
         END ( PACKAGE BODY )? ( simple_name )? SEMI_COLON)
    ;

package_body_declarative_part :
      ( package_body_declarative_item )*
    ;

package_body_declarative_item :
      subprogram
```

```
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | use_clause
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

subprogram :
      subprogram_header ( subprogram_body )? SEMI_COLON
    ;

subprogram_declaration :
      subprogram_header SEMI_COLON
    ;

subprogram_header :
      #(PROCEDURE designator
          ( L_PAREN formal_parameter_list R_PAREN )?)
    | #(FUNCTION ( side_effects )? designator
          ( L_PAREN formal_parameter_list R_PAREN )?
          RETURN type_mark
       )
    ;

side_effects :
      PURE
    | IMPURE
    ;

designator :
      operator_symbol
    | identifier
    ;

operator_symbol :
      STRING_LITERAL
```

```
    ;

formal_parameter_list :
      interface_list
    ;

subprogram_body :
      IS subprogram_declarative_part sequence_of_statements END
      ( FUNCTION | PROCEDURE )? ( designator )?
    ;

subprogram_declarative_part :
      ( subprogram_declarative_item )*
    ;

subprogram_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declarative_item
    | use_clause
    ;

signature :
      L_BRACKET ( type_mark ( COMMA type_mark )* )?
     ( RETURN type_mark )? R_BRACKET
    ;

attribute_declarative_item :
      #(ATTRIBUTE ( attribute_declaration_tail
                  | attribute_specification_tail
                )
         )
    ;

attribute_declaration_tail :
      #(COLON identifier type_mark SEMI_COLON)
```

```
    ;

attribute_specification_tail :
      identifier OF entity_specification IS expression SEMI_COLON
    ;

attribute_specification :
      #(ATTRIBUTE attribute_specification_tail)
    ;

entity_specification :
      entity_name_list COLON entity_class
    ;

entity_name_list :
      #(IDENT5 entity_designator ( signature )?
         (COMMA entity_designator ( signature )?)*
        )
    | OTHERS
    | ALL
    ;

entity_designator :
      simple_name
    | operator_symbol
    ;

entity_class :
      ENTITY
    | ARCHITECTURE
    | CONFIGURATION
    | PROCEDURE
    | FUNCTION
    | PACKAGE
    | TYPE
    | SUBTYPE
    | CONSTANT
    | SIGNAL
    | VARIABLE
    | COMPONENT
    | LABEL
```

```
    | LITERAL
    | UNITS
    | GROUP
    | FILE
    | ( NATURE | SUBNATURE | QUANTITY | TERMINAL )
    ;

configuration_specification! :
      #(FOR component_specification binding_indication SEMI_COLON)
    ;

component_specification :
      #(COLON instantiation_list complex_name)
    ;

instantiation_list :
      #(IDENT6 simple_name (COMMA simple_name)*)
    | OTHERS
    | ALL
    ;

binding_indication !:
      ( USE entity_aspect )?
      ( generic_map_aspect )?
      ( port_map_aspect )?
    ;

entity_aspect :
      #( ENTITY selected_or_simple_name
         ( L_PAREN simple_name R_PAREN )?
       )
    | #(CONFIGURATION complex_name)
    | OPEN
    ;

generic_map_aspect :
      #(GENERIC MAP L_PAREN! aslistin R_PAREN!)
    ;

port_map_aspect :
      #(po:PORT {amfile.println(); amfile.print(po.getText()+ " ");}
```

```
        ma:MAP  { amfile.print(ma.getText()+ " ");}
        lp:L_PAREN! {amfile.print(lp.getText()+ " ");}
        as:aslistin
        rp:R_PAREN!  {amfile.print(rp.getText()+ " ");}
        )
      //{#port_map_aspect=#asl; }
    ;

disconnection_specification :
      DISCONNECT guarded_signal_specification
       AFTER expression SEMI_COLON
    ;

guarded_signal_specification :
      #(COLON signal_list type_mark)
    ;

signal_list :
      #(IDENT7 complex_name (COMMA complex_name)*)
    | OTHERS
    | ALL
    ;

use_clause :
      #(USE complex_name ( COMMA selected_name )*
         SEMI_COLON
      )
    ;

scalar_type_definition :
      enumeration_type_definition
    | range_constraint ( physical_type_definition )?
    ;

enumeration_type_definition :
      #(IDENT8 L_PAREN! enumeration_literal
          (COMMA enumeration_literal)*
           R_PAREN!
        )
    ;
```

```
enumeration_literal :
      identifier
    | CHARACTER_LITERAL
    ;

range_constraint :
      #(RANGE range)
    ;

range :
      ( ( selected_name QUOTE ( attribute_range
                              | attribute_reverse_range
                             )
        ) )
          => ( ( selected_name QUOTE
                       ( attribute_range | attribute_reverse_range )
             ) )
    | simple_expression direction simple_expression
    ;

physical_type_definition :
      UNITS base_unit_declaration ( secondary_unit_declaration )*
      END UNITS ( simple_name )?
    ;

base_unit_declaration :
      identifier SEMI_COLON
    ;

secondary_unit_declaration :
      identifier EQUAL physical_literal SEMI_COLON
    ;

physical_literal :
      ( integer_literal )? simple_name
    ;

abstract_literal :
      integer_literal
//    | real_literal
    ;
```

```
composite_type_definition :
      array_type_definition
    | record_type_definition
    ;

array_type_definition :
      #(ARRAY array_type_definition_head OF subtype_indication)
    ;

array_type_definition_head :
      #(IDENT9 L_PAREN! array_dimension
          (COMMA array_dimension)*
          R_PAREN!
        )
    ;

array_dimension :
      ( index_subtype_definition )
        => ( index_subtype_definition )
    | ( discrete_range )
    ;

index_subtype_definition :
      type_mark RANGE LESS_GREATER
    ;

record_type_definition :
      #(RECORD ( element_declaration )+ END RECORD ( simple_name )?)
    ;

element_declaration :
      #(COLON identifier_list subtype_indication SEMI_COLON)
    ;

access_type_definition :
      ACCESS subtype_indication
    ;

file_type_definition :
      FILE OF type_mark
```

```
    ;

type_declaration :
      #(TYPE identifier ( IS type_definition )? SEMI_COLON)
    ;

type_definition :
      scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition
    | ( PROTECTED BODY )
        => protected_type_body
    | protected_type_declaration
    ;

subtype_declaration :
      #(SUBTYPE identifier IS subtype_indication SEMI_COLON)
    ;

subtype_indication :
//      type_mark  (constraint)? (TOLERANCE expression)?
//    | complex_name type_mark (constraint)? (TOLERANCE expression)?
              ( ( complex_name type_mark )
                  => complex_name type_mark ( constraint )?
         | type_mark ( constraint )? ) ( ( TOLERANCE )
                                          => TOLERANCE expression )?
   ;

selected_or_simple_name :
      simple_name ( DOT simple_name )*
    ;

type_mark :
      selected_or_simple_name
    ;

constraint :
      range_constraint
    | index_constraint
    ;
```

```
index_constraint :
      #(  IDENT10 L_PAREN! discrete_range
          (COMMA discrete_range)* R_PAREN!
       )
    ;

constant_declaration :
      #(CONSTANT identifier_list COLON subtype_indication
        ( initialization )? SEMI_COLON
      )
    ;

identifier_list !:
     #(id:IDENT11
      i1:identifier  {amfile.print(i1.getText());}
        {#identifier_list=#(id, #i1);}
       ( co:COMMA {amfile.print(co.getText()+" ");}
        i2:identifier  {amfile.print(i2.getText());}
        {#identifier_list.addChild(#i2);}
       )*
      )
    ;

identifier_list1 !:
     #(id:IDENT11
      i1:identifier  {amfile.print(i1.getText());}
        {#identifier_list1=#(id, #i1);}
       ( co:COMMA {amfile.print(co.getText()+" ");}
        i2:identifier  {amfile.print(i2.getText());}
        {#identifier_list1.addChild(#i2);}
       )*
      )
    ;




initialization :
      COLON_EQUAL expression
```

```
    ;

signal_declaration! :
      #(SIGNAL identifier_list COLON! subtype_indication
          ( signal_kind )? ( initialization )? SEMI_COLON
      )
    ;

signal_kind :
      REGISTER
    | BUS
    ;

shared_variable_declaration :
      SHARED VARIABLE variable_declaration_body
    ;

variable_declaration :
      ( SHARED )? VARIABLE variable_declaration_body
    ;

variable_declaration_body :
      #(COLON identifier_list subtype_indication
       ( initialization )? SEMI_COLON
       )
    ;

file_declaration :
      #(FILE identifier_list COLON subtype_indication
          ( file_open_information )? SEMI_COLON
      )
    ;

file_open_information :
      ( OPEN expression )? IS file_logical_name
    ;

mode :
      IN
    | OUT
    | INOUT
```

```
    | BUFFER
    | LINKAGE
    ;

file_logical_name :
      expression
    ;

interface_declaration !:
      ( ( ( CONSTANT!
            | si:SIGNAL {amfile.println(si.getText());}
            | VARIABLE! ) )?
      li:identifier_list
      co:COLON {amfile.print(co.getText()+ "  ");}
      ( mo:mode {amfile.print(mo.getText()+ "  ");} )?
        id:subtype_indication {amfile.print(id.getText()+ "  ");}
         ( BUS! )?
      ( initialization! )?
          {
            AST t;
            AST tempA;
            AST tNext;
            AST currentIST;
            #t=#li.getFirstChild();
            #tNext=#t.getNextSibling();
            #t.setNextSibling(null);
            if (mo!= null)
             {

               #tempA=#(t,[IDENTIFIER,id.getText()],
                          [mo.getType(),mo.getText()]
                        );

             }
            else
             {
              #tempA=#(t,[IDENTIFIER,id.getText()]);

             }
            #interface_declaration=#([IDENTIFIER ,"PortList"],tempA);
            while (#tNext !=  null)
```

```
            {
              #t=#tNext;

              #tNext=#t.getNextSibling();
              #t.setNextSibling(null);
              if (mo!= null)
               {
                 #tempA=#(t,[IDENTIFIER,id.getText()],
                          [mo.getType(),mo.getText()]
                         );

               }
              else
               {
                 #tempA=#(t,[IDENTIFIER,id.getText()]);

               }
                #interface_declaration.addChild(#tempA);
             }

          }
     | FILE identifier_list COLON subtype_indication
     | ( TERMINAL identifier_list COLON subnature_indication
         | QUANTITY identifier_list COLON ( IN | OUT )?
             subtype_indication ( initialization )?
       )
   )
    ;

interface_list !:
      i2:interface_declaration {  #interface_list=#i2;}
      ( sc:SEMI_COLON { amfile.print(sc.getText());}
        i1:interface_declaration
               {
                   #interface_list.addChild(#i1.getFirstChild());
               }
       )*

    ;

aslistin !:
```

```
      #( IDENT12 a1:association
         {amfile.print(a1.getText()+ " ");}
         {#aslistin=#([IDENTIFIER, "aslistin_root"],#a1);}
         ( co:COMMA! {amfile.print(co.getText()+ " ");}
           a2:association {amfile.print(a2.getText()+ " ");}
           {#aslistin.addChild(#a2);}
         )*
       )
    ;

association :
      association_element ( EQUAL_GREATER association_element )?
    ;

association_element :
      expression
    | OPEN
    ;

alias_declaration :
      #(ALIAS alias_designator ( COLON alias_indication )?
           IS complex_name ( signature )? SEMI_COLON)
    ;

alias_designator :
      identifier
    | CHARACTER_LITERAL
    | operator_symbol
    ;

alias_indication :
      ( type_mark TOLERANCE )
        => subnature_indication
    | subtype_indication
    ;
// generic_clause may be important but it is not sent to begenerater
component_declaration !:
      #( c:COMPONENT {amfile.print(c.getText()+ " ");}
       i:identifier {amfile.print(i.getText()+ " ");}
       (id:IS {amfile.print(id.getText()+ " ");} )?
       ( gc:generic_clause )?
```

```
      ( pl:port_clause )?
        end:END {amfile.print(end.getText()+ " ");}
        co:COMPONENT {amfile.print(co.getText()+ " ");}
        ( sn:simple_name {amfile.print(sn.getText()+ " ");})?
        sc:SEMI_COLON {amfile.print(sc.getText()+ " ");}
        )


       {#component_declaration=#(c,i,pl);}



/*{
  AST tempA;
  AST MachineA;
  AST SeesA;
  AST VariablesA;
  AST DefinitionsA;
  AST InvariantA;
  AST InitialisationA;
  AST OperationsA;
  AST NextA;
  AST ModeA;
  AST TypeA;
  AST SeesSearch;
  String Oper;
  String OperMod;
  String OperType;
  String SearchS;

  #tempA=#([IDENTIFIER,"COMPONENT"]);
  #MachineA=#([IDENTIFIER,"MACHINE"]);
  #SeesA=#([IDENTIFIER,"SEES"]);
  #VariablesA=#([IDENTIFIER,"VARIABLES"]);
  #DefinitionsA=#([IDENTIFIER,"DEFINITIONS"]);
  #InvariantA=#([IDENTIFIER,"INVARIANT"]);
  #InitialisationA=#([IDENTIFIER,"INITIALISATION"]);
  #OperationsA=#([IDENTIFIER,"OPERATIONS"]);

  #i.setNextSibling(null);
```

```
#MachineA.setFirstChild(#i);
#NextA=#pl.getFirstChild();
while (#NextA != null)
 {
   #TypeA=#NextA.getFirstChild();
   #ModeA=#TypeA.getNextSibling();
   #SeesSearch=#SeesA.getFirstChild();
   SearchS=TypeA.getText();
   while((#SeesSearch != null)
        && (!(SearchS.equals(#SeesSearch.getText())))
        )
     {
       #SeesSearch = #SeesSearch.getNextSibling();
     }
   if (#SeesSearch == null)
    {
      #SeesA.addChild(  #( [#TypeA.getType(),
                           #TypeA.getText()] )
                        );
    }
   #VariablesA.addChild( #([#NextA.getType(),
                            #NextA.getText()])
                         );
   #InvariantA.addChild( #( [#NextA.getType(),
                             #NextA.getText()],
                            [#TypeA.getType(),
                            #TypeA.getText()]
                           )
                         );
   #InitialisationA.addChild( #([#NextA.getType(),
                                 #NextA.getText()
                                 ],
                                [#TypeA.getType(),
                                 #TypeA.getText()
                                 ]
                                )
                              );
   #OperationsA.addChild( #([#NextA.getType(),
                             #NextA.getText()
                             ],
                             [#ModeA.getType(),
```

```
                                #ModeA.getText()],
                                [#TypeA.getType(),
                                #TypeA.getText()]
                                )
                            );

       #NextA=#NextA.getNextSibling();
     }

    #DefinitionsA.addChild(#([BCOMMENT_DEF,comment1] ));
    #InvariantA.addChild( #([BCOMMENT_INV,comment2] ));
    #tempA.addChild(#MachineA);
    #tempA.addChild(#SeesA);
    #tempA.addChild(#VariablesA);
    #tempA.addChild(#DefinitionsA);
    #tempA.addChild(#InvariantA);
    #tempA.addChild(#InitialisationA);
    #tempA.addChild(#OperationsA);

    #component_declaration=#tempA;
  }*/
    ;

generic_clause :
      #(GENERIC
       L_PAREN
        generic_list
        R_PAREN
        SEMI_COLON
        )
    ;

port_clause !:
      #(po:PORT {amfile.println();
                 amfile.println('\t'+po.getText());
                  }
        lp:L_PAREN {amfile.print("\t   "+lp.getText());}
        pl:port_list
        rp:R_PAREN {amfile.println();
                    amfile.print( '\t'+ rp.getText());
                   }
```

```
        sc:SEMI_COLON{amfile.println(sc.getText());
                    }
        )
      { #port_clause=#pl; }
    ;

generic_list :
      interface_list
    ;

port_list :
      interface_list
    ;

group_declaration :
      #(GROUP identifier COLON selected_or_simple_name
        L_PAREN group_constituent_list R_PAREN SEMI_COLON
       )
    ;

group_template_declaration :
      GROUP identifier IS L_PAREN entity_class_entry_list R_PAREN
    ;

entity_class_entry_list :
      #(IDENT13 entity_class_entry (COMMA entity_class_entry)*)
    ;

entity_class_entry :
      entity_class ( LESS_GREATER )?
    ;

group_constituent_list :
      #(IDENT14 group_constituent ( COMMA group_constituent)*)
    ;

group_constituent :
      ( CHARACTER_LITERAL )
        => CHARACTER_LITERAL
    | complex_name
    ;
```

```
boolean_expression :
      expression
    ;

time_expression :
      expression
    ;


expression :
      relation

expression_tail

    ;

expression_tail :
      (
      //( and_or_xor_xnor ) =>
      and_or_xor_xnor relation )+
    | ( ( nand_nor )
          => nand_nor relation )?
    ;

relation :
      shift_expression ( ( relational_operator )
                          => relational_operator shift_expression )?
    ;

shift_expression :
      simple_expression ( ( shift_operator )
                           => shift_operator simple_expression )?
    ;

simple_expression :
     ( sign_operator )? term (
     //( adding_operator ) =>
     adding_operator term )*
    ;
```

```
term :
      factor (
      //( multiplying_operator ) =>
      multiplying_operator factor )*
    ;

factor :

    primary

    (  EXPONENT primary  )?

      | abs_not primary
    ;

numeric_literal :
      abstract_literal ( complex_name )?
    ;

literal :
      numeric_literal
    | CHARACTER_LITERAL
    | STRING_LITERAL
    | BIT_STRING_LITERAL
    | NULL
    ;

primary :
      ( STRING_LITERAL L_PAREN )
        => complex_name
    | ( literal )
        => ( literal )
    | complex_name
    | allocator
    | aggregate
    ;

association_list_out :
      #(IDENT15 element_association (COMMA element_association)*)
    ;
```

```
allocator :
      ( NEW subtype_indication )
        => ( NEW subtype_indication )
    | ( NEW complex_name )
    ;

element_association :
      ( choices EQUAL_GREATER expression )
        => ( choices EQUAL_GREATER expression )
    | association
    ;

complex_name :
      attribute_name
    ;

attribute_name :
      selected_name ( QUOTE attribute )*
    ;

attribute :
      ( selected_name )
        => attribute_user_defined
    | attribute_active
    | attribute_ascending
    | attribute_base
    | attribute_delayed
    | attribute_driving_value
    | attribute_driving
    | attribute_event
    | attribute_high
    | attribute_image
    | attribute_instance_name
    | attribute_last_active
    | attribute_last_event
    | attribute_last_value
    | attribute_leftof
    | attribute_left
    | attribute_length
    | attribute_low
    | attribute_path_name
```

```
    | attribute_pos
    | attribute_pred
    | attribute_range
    | attribute_rightof
    | attribute_right
    | attribute_reverse_range
    | attribute_simple_name
    | attribute_stable
    | attribute_succ
    | attribute_transaction
    | attribute_quiet
    | attribute_value
    | attribute_val
    | ( aggregate )
    ;

attribute_user_defined :
      selected_name
    ;

attribute_active :
      ACTIVE
    ;

attribute_ascending :
      ASCENDING ( ( L_PAREN )
                    => L_PAREN expression R_PAREN )?
    ;

attribute_base :
      BASE
    ;

attribute_delayed :
      DELAYED ( ( L_PAREN )
                  => L_PAREN expression R_PAREN )?
    ;

attribute_driving_value :
      DRIVING_VALUE
    ;
```

```
attribute_driving :
      DRIVING
    ;

attribute_event :
      EVENT
    ;

attribute_high :
      HIGH ( ( L_PAREN )
               => L_PAREN expression R_PAREN )?
    ;

attribute_image :
      IMAGE L_PAREN expression R_PAREN
    ;

attribute_instance_name :
      INSTANCE_NAME
    ;

attribute_last_active :
      LAST_ACTIVE
    ;

attribute_last_event :
      LAST_EVENT
    ;

attribute_last_value :
      LAST_VALUE
    ;

attribute_leftof :
      #(LEFTOF L_PAREN expression R_PAREN)
    ;

attribute_left :
      #(LEFT ( ( L_PAREN )
                 => L_PAREN expression R_PAREN )?)
```

```
    ;

attribute_length :
      #(LENGTH ( ( L_PAREN )
                  => L_PAREN expression R_PAREN )?)
    ;

attribute_low :
      #(LOW ( ( L_PAREN )
                => L_PAREN expression R_PAREN )?)
    ;

attribute_path_name :
      PATH_NAME
    ;

attribute_pos :
      #(POS L_PAREN expression R_PAREN)
    ;

attribute_pred :
      #(PRED L_PAREN expression R_PAREN)
    ;

attribute_range :
      RANGE ( ( L_PAREN )
                => L_PAREN expression R_PAREN )?
    ;

attribute_rightof :
      #(RIGHTOF L_PAREN expression R_PAREN)
    ;

attribute_right :
      #(RIGHT ( ( L_PAREN )
                  => L_PAREN expression R_PAREN )?)
    ;

attribute_reverse_range :
      REVERSE_RANGE ( ( L_PAREN )
                        => L_PAREN expression R_PAREN )?
```

```
    ;

attribute_simple_name :
      SIMPLE_NAME
    ;

attribute_stable :
      #(STABLE ( ( L_PAREN )
                  => L_PAREN expression R_PAREN )?)
    ;

attribute_succ :
      #(SUCC L_PAREN expression R_PAREN)
    ;

attribute_transaction :
      TRANSACTION
    ;

attribute_quiet :
      #(QUIET ( ( L_PAREN )
                  => L_PAREN expression R_PAREN )?)
    ;

attribute_value :
      #(VALUE ( ( L_PAREN )
                  => L_PAREN expression R_PAREN )?)
    ;

attribute_val :
      #(VAL L_PAREN expression R_PAREN)
    ;

selected_name :
      indexed_name ( DOT
                       ( indexed_name
                         | ALL ( L_PAREN index_specifier R_PAREN )*
                        )
                   )*
    ;
```

```
indexed_name :
      name ( L_PAREN index_specifier R_PAREN )*
    ;

index_specifier :
      ( expression direction )
        => discrete_range ( EQUAL_GREATER expression )?
    | association_list_out
    ;

slice_specifier :
      discrete_range
    ;

name :
      simple_name
    | STRING_LITERAL
    | CHARACTER_LITERAL
    ;

simple_name :
      identifier
    ;

alpha_literals :
      BIT_STRING_LITERAL
    | CHARACTER_LITERAL
    | STRING_LITERAL
    | NULL
    ;

physical_type_name :
      identifier
    ;

discrete_range :
      ( range )
        => ( range )
    | subtype_indication
    ;
```

```
direction :
      TO
    | DOWNTO
    ;

vhdl_label :
      identifier
    ;

integer_literal :
      DECIMAL_INTEGER_LITERAL
    | BASED_INTEGER_LITERAL
    ;

//real_literal :
//      decimal_floating_point_literal
//    | based_floating_point_literal
//    ;

identifier :
      id:IDENTIFIER
    ;

nature_declaration :
      #(NATURE identifier IS nature_definition SEMI_COLON)
    ;

nature_definition :
      scalar_nature_definition
    | array_nature_definition
    | record_nature_definition
    ;

terminal_declaration! :
      #(TERMINAL identifier_list COLON subnature_indication SEMI_COLON)
    ;

quantity_declaration! :
      #(QUANTITY quantity_declaration_tail SEMI_COLON)
    ;
```

```
quantity_declaration_tail :
      ( identifier_list COLON )
        => identifier_list COLON subtype_indication
             ( source_aspect | initialization )?
    | ( identifier_list
             ( TOLERANCE | COLON_EQUAL | ACROSS | THROUGH )
       )
        => across_through_aspect_body
            ( ACROSS ( across_through_aspect_body THROUGH )?
               | THROUGH
            )
           terminal_aspect
    | terminal_aspect
    ;

across_through_aspect_body :
      identifier_list
      ( TOLERANCE expression )?
      ( initialization )?
    ;

terminal_aspect :
      complex_name ( TO complex_name )?
    ;

source_aspect :
      #(SPECTRUM COMMA simple_expression simple_expression)
    | NOISE simple_expression
    ;

step_limit_specification :
      #(LIMIT step_limit_specification_head COLON
         type_mark WITH expression SEMI_COLON
       )
    ;

step_limit_specification_head :
      #(IDENT16 selected_or_simple_name
       (COMMA selected_or_simple_name)*
       )
    | OTHERS
```

```
    | ALL
    ;

scalar_nature_definition :
      type_mark ACROSS type_mark THROUGH identifier REFERENCE
    ;

array_nature_definition :
      #(ARRAY array_nature_definition_head OF subnature_indication)
    ;

array_nature_definition_head :
      #(IDENT17 L_PAREN! array_dimension (COMMA array_dimension)* R_PAREN!)
    ;

record_nature_definition :
      #(RECORD ( nature_element_declaration )+ END RECORD ( simple_name )?)
    ;

nature_element_declaration :
      #(COLON identifier_list subnature_indication SEMI_COLON)
    ;

subnature_declaration :
      #(SUBNATURE IDENTIFIER IS subnature_indication SEMI_COLON)
    ;

subnature_indication :
      type_mark ( index_constraint )?
      ( TOLERANCE expression ACROSS expression THROUGH )?
    ;

concurrent_break_statement :
      BREAK ( break_element ( COMMA break_element )* )?
      ( ON complex_name ( COMMA complex_name )* )?
        ( WHEN expression )? SEMI_COLON
    ;

set_of_simultaneous_statements :
      ( concurrent_statement )*
    ;
```

```
simultaneous_statement :
       simultaneous_stmt
    ;

simultaneous_stmt :
      ( simple_expression EQUAL_EQUAL)
        => simple_simultaneous_statement
    | simultaneous_if_statement
    | simultaneous_case_statement
    | simultaneous_procedural_statement
    | simultaneous_null_statement
    ;

simple_simultaneous_statement :
      simple_expression EQUAL_EQUAL simple_expression
       ( TOLERANCE expression )? SEMI_COLON
    ;

simultaneous_if_statement :
      IF condition USE set_of_simultaneous_statements
      ( ELSIF condition USE set_of_simultaneous_statements )*
      ( ELSE set_of_simultaneous_statements )?
       END USE ( simple_name )? SEMI_COLON
    ;

simultaneous_case_statement :
      CASE expression USE
        ( WHEN choices EQUAL_
          GREATER set_of_simultaneous_statements
          )+
      END CASE ( simple_name )? SEMI_COLON
    ;

simultaneous_procedural_statement :
      #(PROCEDURAL ( IS )?
       ( procedural_declarative_item )* sequence_of_statements
      END PROCEDURAL ( simple_name )? SEMI_COLON)
    ;

procedural_declarative_item :
```

```
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | alias_declaration
    | attribute_declarative_item
    | use_clause
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

simultaneous_null_statement :
      NULL SEMI_COLON
    ;

break_statement :
      BREAK ( break_element ( COMMA break_element )* )?
     ( WHEN expression )? SEMI_COLON
    ;

break_element! :
      ( FOR complex_name USE )?
        complex_name EQUAL_GREATER expression
    ;

protected_type_declaration :
      #(PROTECTED ( protected_type_declarative_item )*
        END PROTECTED ( simple_name )?)
    ;

protected_type_declarative_item :
      subprogram_declaration
    | attribute_specification
    | use_clause
    ;

protected_type_body :
      #(BODY PROTECTED ( protected_type_body_declarative_item )* END
              PROTECTED BODY ( simple_name )?
```

```
        )
    ;

protected_type_body_declarative_item :
      subprogram
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declarative_item
    | use_clause
    | ( GROUP IDENTIFIER COLON )
        => group_declaration
    | group_template_declaration
    ;

and_or_xor_xnor! :
       AND
     | OR
     | XOR
     | XNOR
     ;

nand_nor! :
      NAND
    | NOR
    ;

relational_operator! :  EQUAL | NOT_EQUAL | LESS
                      | LESS_EQUAL | GREATER | GREATER_EQUAL ;

shift_operator!  : SLL | SRL | SLA | SRA | ROL | ROR ;

sign_operator!  : PLUS | MINUS ;

adding_operator! : PLUS | MINUS | AMPERSAND ;

multiplying_operator! : MULTIPLY | DIVIDE | MOD | REM ;
```

```
abs_not! : ABS | NOT ;
```

Annexe D

Generateur de B

```
// this file is checked on 29 jan 2004.
// signal_type is remplaced by bhdl_type on 29 jan 2004
// B TREE Walker.
//
//      It traverses the B tree generated by the treewalker which
//      parses the vgui tree
//
//      This version is correctly compiled on 24 feb 2003
//      Authors : Ammar Aljer, Philippe Devienne
//
//      Current version : Dec 2003

{
    import java.io.*;
    import antlr.collections.AST;
}

//----------------------------------------------------------
class  bgenerator extends TreeParser;
options
{
    importVocab     = vhdl;
            buildAST    = true;
    codeGenMakeSwitchThreshold = 3;
            codeGenBitsetTestThreshold = 4;

    k = 1;
}
```

```
//-----------------------------------------------

{       AST d_collect;
        AST p1;
        AST pp1;
        //Global variable to creat an abstract machine file
        PrintWriter amfile = null;
        //Global variable to creat a d DEFINITION file that
        //coressponds to an abstarct machin
     PrintWriter deffile = null;
        //Global variable to Distinguish between 2 rules
        //in their common sub rule
        String dist="";
        String dista="";
        String archi_name="";
        String entit_name="";String machine_abstraite="";
        String invText="";



        String inv_machine="";
        String commentaire="";
}


// Grammar
// -------------------

//sub_diagrams

diagram_design      :   (subdesign )*
;



subdesign  :
        abstract_machine
      | concrete_spec
;
```

```
abstract_machine    :   #(a:ABSTRACT
         machine_clause
         sees_clause
         var_clause
         def_clause
         invarclause
         initi_clause
         oper_clause
         {amfile.println("END");}
         )
    {
    // Abstarct machine file closing
    amfile.close();
    }
;



machine_clause      :   #(IDENTIFIER  id:IDENTIFIER )
        {//Action to creat and open an abstract machine file
            entit_name=#id.getText();
            amfile = null;
            try{
                    amfile = new PrintWriter(
                    new BufferedWriter(new
                         FileWriter("./B/"
                         + entit_name + ".mch"))
                                    );
               }catch(java.io.IOException e){}
            amfile.println("MACHINE");
            amfile.println("\t" + id.getText());
            amfile.println("");
        }//End of action
;

sees_clause         :
     #(IDENTIFIER
                    {amfile.println("SEES");}
       (id:IDENTIFIER
          /*{
```

```
            amfile.print( "\t   " + #id.getText() );
            if (id.getNextSibling() != null)
                { amfile.println("\t , ");}
        }*/
     )* )
        {
            amfile.println("\t   "+ "bhdl_type");
            amfile.println("");
            amfile.println("");
        }
    ;

var_clause      :   #(IDENTIFIER
                    {
                        amfile.println("VARIABLES");
                        amfile.print( "\t   ");
                    }
                   (id:IDENTIFIER
                    {
                        amfile.println(  #id.getText() );
                        if (id.getNextSibling() != null)
                            { amfile.print("\t , ");}
                    }
                   )* )
                    {
                        amfile.println("");
                        amfile.println("");
                    }
                   ;

  def_clause          : #(iden:IDENTIFIER
                {   amfile.println("DEFINITIONS");
     amfile.println( "\t"+ "\""+
                              "bhdl_type.def"+"\""+  ";"
                              );
                     amfile.println( "\t"+ "\""+
```

```
                            entit_name + ".def"+"\""
                           );
    amfile.println("");
                    amfile.println("");

                }
{//Action to creat and open an independent DEFINITION  file
       deffile = null;
     try{
             deffile = new PrintWriter(
             new BufferedWriter(
                       new FileWriter("./B/"
                       + entit_name + ".def")
                               )
                                   );
         }catch(java.io.IOException e){}
   deffile.println("
          /* This file contains all definitions related to "
         + entit_name +".mch file. */"
                          );
 //deffile.print("\t" + "inv_"
                       + archi_name + "_" + entit_name
                         );
deffile.println("DEFINITIONS");
 deffile.print("\t" + "inv_"  + entit_name);
                 }//End of action

       (liste3)?
       ( b:BCOMMENT_E)?
         {
  if (#b != null)
      {  deffile.print(  b.getText());
      }
  else
       {
          AST onevar;
          String compt01="";
          deffile.print(" == ( ");
          #onevar=iden.getNextSibling()
                      .getNextSibling()
                       .getFirstChild();
```

```
            while ( onevar!=null)
              {  deffile.println("");
                 deffile.print("\t\t");
                 if (compt01=="" )
                   {compt01="ok";
                   }
                 else
                    {  deffile.print(" & ");
                    }
                     deffile.print(" " +
                        onevar.getText() +
                        " : " +
                        onevar.getFirstChild().getText() +
                        " "
                      );
                  #onevar=#onevar.getNextSibling();
                 }
               if ( compt01 != "ok")
                 { deffile.print(" true ");
                 }
                deffile.print(" ) ");
               }
           deffile.println("");
           deffile.println("");

   }
                       )
    {
    // Definitions file closing
    deffile.close();
    }
;



//     {AST com=null;#com=#([BCOMMENT_DEF,"pas de def"]);}
                    (com1:BCOMMENT_DEF{#com=#com1;})? )
//    {
//
//        inv_machine=("inv_" + entit_name + com.getText());
//        amfile.print("\t" + inv_machine);
```

```
//        amfile.println("");
//        amfile.println("");
//    }
//
//    ;
//  invarclause        :
      #(IDENTIFIER {
              AST com=null;#com=#([BCOMMENT_INV,"pas de inv"]);}
              (com1:BCOMMENT_INV{#com=#com1;})?)
//      {
//
//          inv_machine=("inv_" + entit_name + com.getText() );
//          commentaire=com.getText();
//          + inv_machine);
//          amfile.println("");
//          amfile.println("");
//      }
//                ;

invarclause        :
           #(IDENTIFIER
                            {amfile.println("INVARIANT");
           //  if (#invarclause.getFirstChild()==null)
           //      { invText=("NO COMMENTS");}
           //  else
                  { invText=("inv_" + entit_name);}
              }
          (liste2)?)
          {amfile.println("\t"+invText );
           amfile.println("");
          }
;


var_type        :   #(var:IDENTIFIER  typ:IDENTIFIER)
       {
           if (dist != " := ")
               {
                   amfile.print( var.getText());
                   dist=" := ";
               }
```

```
            else
                {amfile.print( ", " + var.getText());}
        }
    ;

initi_clause        :   #(i:IDENTIFIER
                        {
                            amfile.println("INITIALISATION");
                            amfile.print("\t ");
                            dist="";
                        }
                    (vt:var_type
                        {

                        /*
                            if (vt.getNextSibling() != null)
                            { amfile.print("\t||  ");}
                        */
                        }
                    )*)
                        {
                            amfile.print(":( " + invText + ") ");
                            amfile.println("");
                            amfile.println("");
                        }
                    ;



oper_clause     :   #(i:IDENTIFIER
                        {
 amfile.println("OPERATIONS");
 amfile.println("\t" + "type_" + entit_name + " = ");

            // amfile.println("\t" + "PRE " + "true");
                        // amfile.print("\t"  );
              amfile.println("\t" + "BEGIN");
              amfile.print("\t" + "      " );
                     }

    (  vmt:var_mod_type {
```

```
    amfile.println(vmt.getText());
    if (vmt.getNextSibling() != null)
        { amfile.print("\t     " + ",");}
                                      }
         )*  )
         {
            amfile.println("\t     " + " :( " + invText +")");
            amfile.println("\t" + "END");
            amfile.println("");
            amfile.println("");
        }

;

var_mod_type        : #(i:IDENTIFIER   mode IDENTIFIER )
;
var_type_mod        : #(i:IDENTIFIER IDENTIFIER  mode )
;

concrete_spec       :
                        #(id:CONCRETE

                          imp_name_clause

                          refines_clause

                          imports_clause

                          (connections

                        )?)

                        {
                            amfile.println("END");
                            amfile.close();
                        }

;

imp_name_clause     :
          #(IDENTIFIER  id1:IDENTIFIER id2:IDENTIFIER)
```

```
        {
            archi_name = id1.getText();
            amfile = null;
            try{
                 amfile = new PrintWriter(
                 new BufferedWriter(new FileWriter("./B/"
                 + id1.getText() + "_" + id2.getText()
                 + ".ref")));
            }catch(java.io.IOException e){}
            amfile.println("REFINEMENT");
            amfile.println("\t" + id1.getText()
               + "_" + id2.getText());
            amfile.println("");
        }
;

refines_clause      :
        #(IDENTIFIER  id:IDENTIFIER)
                        {
                amfile.println("REFINES");
                amfile.println("\t" + id.getText());
                amfile.println("");
                amfile.println("SEES");
                amfile.println("\t" +"bhdl_type");
                amfile.println("");
             }
;



imports_clause      :
            #(IDENTIFIER  (#(c:COMPONENT i:IDENTIFIER
            #(IDENTIFIER ( pl:var_type_mod )*)))*)


;

//ARC_BODY bransh
connections     :
        #(p2:IDENTIFIER //ARC_BODY node
               {#p1=#p2;}
```

```
                // #(ii:IDENTIFIER IDENTIFIER
                {

                    String definition="";
                    amfile.println("INCLUDES");
                    dist="";
                    String compt0="";
                    String compt00="";

                    #pp1=#([IDENTIFIER,"vartree"]);
                    pp1.setFirstChild(null);

  #d_collect=#([IDENTIFIER,"DefintionsCollection"]);
  #d_collect.setFirstChild( null);
                }

(#(IDENTIFIER //architecture_statement_part
    (#(IDENT25 //IDENT25 copresponds to newcomp in the tree
        (i1:IDENTIFIER //The new compenent name
            #(IDENT23 //costat2b in the tree
                i2:IDENTIFIER //type of the new comp
                    {




 if (dist != "ok")
 {
     amfile.print("\t" + i1.getText() + "." + i2.getText());
     dist="ok";
 }
 else
 {
     amfile.print(", " + i1.getText() + "." + i2.getText());
 }

                        }
```

```
                        #(PORT MAP
                            l:liste
                            {
                                if (compt0 !="ok")
                                {
   definition=definition.concat( "\r\n" + "\t   " + "inv_"
                                    + i2.getText() + "( "
                                   );
//d_collect.addChild(i2);
     compt0="ok";
                        }
                    else
                        {
                            definition=definition.concat(
                            "\r\n" + "\t  " + " & " + "inv_"
                            + i2.getText() + "( "
                            );
                        }
 //local variables to copy noeds"
 //AST fstParOfList; //first variable of a parametrs list
 AST oneVarCopy;
 AST aMachineType;
 AST aMaTypeVar;
 //fstParOfList=null;
 oneVarCopy=null;
 aMachineType=null;
 aMaTypeVar=null;
 //fstParOfList=#l.getFirstChild();
 //if (fstParOfList!=null)
//{oneVarCopy=#( [#fstParOfList.getType(),
                  #fstParOfList.getText()]
                );
    //}
 if (i2!=null)
 {
  aMachineType=#( [#i2.getType(), #i2.getText()] );
 }



                // pp1.addChild(#oneVarCopy);
```

```
//verify that the new elements we add do not existe in
//d_collect tree before
//
aMaTypeVar=d_collect.getFirstChild();
while((#aMaTypeVar!=null)
    && (!((aMaTypeVar.getText())
        .equals(aMachineType.getText()))
    ))
{
#aMaTypeVar=#aMaTypeVar.getNextSibling();}
if (#aMaTypeVar==null)
             { d_collect.addChild(#aMachineType);}
//travers the port list parameters of one machine
//to add them to DEFINITION clause
// definition=definition.concat( #l.getText());
               AST o=null;
               #o=#l.getFirstChild();
               compt00="";
               while (o != null)
                   {
//copy the variable to add it to
//the variables of the machine
oneVarCopy=#( [#o.getType(), #o.getText()] );
pp1.addChild(#oneVarCopy);
                        if (compt00 !="ok")
             {
                 definition=definition.concat(
                                        #o.getText()
                                          );
                 compt00="ok";
             }
         else
             {
                 definition=definition.concat(
                                    ", " + #o.getText()
                                               );
             }
         #o=#o.getNextSibling();
      }
      definition=definition.concat( "  ) ");
```

```
        }
) // END
) // END
) // END IDENT23
))*// END IDENT25
)
)  // END archi Stat part
)   //END ARC BODY
//)
        {
            amfile.println("");
            amfile.println("");
        }
        {
            AST i=null;
            AST j=null;
            String variable="";
            String variable1="";
            String variable2="";
            amfile.println("VARIABLES");
            dista="";
             #j=pp1.getFirstChild();

                    while (j!=null)
                    {

//delete the repeted variables
    variable1=#j.getText();
   #i=#j;

   while (#i.getNextSibling()!=null)
       {
           variable2=#i.getNextSibling().getText();




                if (variable1.equals( variable2))
                    {
                            #i.setNextSibling(
```

```
                                    #i.getNextSibling()
                                          .getNextSibling());

                            }
                        else
                            {
                                    #i=#i.getNextSibling();
                            }
                  }
               #j=#j.getNextSibling();
           }


        #i=pp1.getFirstChild();
        while(#i!=null)
            {variable=#i.getText();
           if (dista != "ok")
            {
                        amfile.print("\t" + variable);
                        dista = "ok";
                    }
                else
                    {
                        amfile.print(", " + variable);
                    }

                #i=#i.getNextSibling();
            }

                        amfile.println("");
                        amfile.println("");
                    }


{
                    amfile.println("DEFINITIONS");
    amfile.println("\t \"bhdl_type.def\" ;");
                    amfile.print("\t" + "inv_" + archi_name
                    + "_" + entit_name);
                    AST k=null;
                    String compt="";
```

```
//delete: #k=#p1.getFirstChild()
// ca march sans #
  #k=pp1.getFirstChild();

 /* C pour afficher toutes les variables reelles

 while (#k != null)
      {
          variable = #k.getText();
          if (compt != "ok")
              {
                  amfile.print("( " + variable);
                  compt = "ok";
              }
          else
              {
                  amfile.print(", " + variable);
              }
          #k=#k.getNextSibling();
      }*/
amfile.println(" == " + "\n" + "\t   "
               + "( " + definition + "\n" + "\t   " + " )"
               );
amfile.println("");

            // in DEFINITION clause "component_type.def"
                     files are declared
                   #i=#d_collect.getFirstChild();
                   while(#i!=null)
                   {  variable=#i.getText();
                          amfile.println("; " + "\t" +"\""
                                        + variable+".def\""
                                        );
                           #i=#i.getNextSibling();
                   }
                                  }

                           {
                               amfile.println("INVARIANT");
    amfile.print("\t ");
    AST m=null;
```

```
String compt2="";

//#m=p1.getFirstChild();
//while (#m != null)
//    {
//        variable = #m.getText();
//        if (compt2 != "ok")
//            {
//                amfile.print(variable );
//                compt2 = "ok";
//            }
//        else
//            {
//                amfile.print(", " + variable);
//            }
//        #m=#m.getNextSibling();
//    }

             amfile.println("inv_" + archi_name
                      + "_" + entit_name
                      );
              amfile.println("");


   amfile.println("INITIALISATION");
   amfile.print("\t ");
   dist=" := " ;
   String compt3="";
   #m=pp1.getFirstChild();
   while (#m != null)
       {
           variable = #m.getText();
           if (compt3 != "ok")
               {
                   amfile.print(variable );
                   compt3 = "ok";
               }
           else
               {
                   amfile.print(", " + variable);
               }
```

```
                #m=#m.getNextSibling();
            }
        amfile.println(" : " + "( inv_" + archi_name
                                +   "_" + entit_name + "
                               )"
                    );
        amfile.println("");
    }
    {
        //generation de l'operation pour le raffinement
        amfile.println("OPERATIONS");
        amfile.println("\t" + "type_" + entit_name + " = ");
        amfile.println("\t" + "BEGIN");

        String compt33="";
        #m=pp1.getFirstChild();
        while (#m != null)
            {
                variable = #m.getText();
                if (compt33 != "ok")
                    {
                        amfile.print(variable );
                        compt33 = "ok";
                    }
                else
                    {
                        amfile.print(", " + variable);
                    }
                #m=#m.getNextSibling();
            }
        amfile.print(" : ");
        amfile.println("( " + "inv_" + archi_name + "_"
                        + entit_name + " )"
                    );
        amfile.println("\t" + "END");
        amfile.println("");
    }


;
```

```
liste       :   #(IDENTIFIER ( i:IDENTIFIER )*)
;


liste2      :
     #(IDENT11
    {invText=invText.concat( " ( " );}
      i1:IDENTIFIER  {invText=invText.concat(
                          i1.getText()
                      );}
    ( i:IDENTIFIER  {invText=invText.concat(
                        ", " + i.getText()
                        );}
    )*)
    {invText=invText.concat( " ) " + "\n");}
;


liste3      :
  #(IDENT11
  {deffile.print( " ( " );}
    i1:IDENTIFIER  {deffile.print( i1.getText() );}
  ( i:IDENTIFIER  {deffile.print( ", " + i.getText() );})*)
  {deffile.println( " ) " );}
;
//compenet_connections    :   #(id:IDENTIFIER
//  { printToStringln( #id.getText() );   }
// (IDENTIFIER connections_list)*)
//;



mode  :     "out"
        |   "in"
        |   "inout"
        |   "buffer"
        |   "linkage"
;
```

Résumé

La conception de systèmes complexes, appelée la conception système, représente le niveau le plus abstrait de conception. L'utilisation de spécifications formelles pendant cette étape est essentielle pour valider la conception, surtout pour satisfaire les besoins de sécurité.

Dans cette thèse on montre l'importance de développement par raffinement : de la spécification abstraite à l'implémentation pour 1) la traçabilité des besoins et exigences 2) une meilleure gestion du développement 3) une conception plus fiable de systèmes parce que cette implémentation est générée par étapes prouvés de construction d'abord de façon générique, puis par des composants logiciels, matériels ou autres.

Ce travail utilise les langages logiciels ADLs, les langages matériels HDLs et la méthode B ; méthode formelle par raffinement. Ceci nous a permet de développer une plateforme BHDL : plateforme pour la conception de composants numériques intégrant 1) une interface pour la conception structurelle de composants électroniques 2) un générateur de code VHDL et, enfin 3) un traducteur en langage formel pour prouver la consistance et le raffinement par l'AtelierB.

Abstract

During of the modelling of complex systems, the design entry, so called system entry, represents the highest level of abstraction of the total system. Within this very first stage of design, the use of a formal specification is considered for the design validation, especially in particular in the case of requirements of safety.

This thesis shows the need of development by refinement: from most abstract specification to the implementation, in order to ensure 1) the traceability of the needs and requirements, 2) a good management of the development and 3) a reliable design of the systems because it is generated by proven construction and, then, these systems are implemented by software, digital or other technologies.

This work makes use of the taxonomy of languages ADLs and HDLs and B method; a formal method for developing by refinement. This enabled us to develop the platform of BHDL Tool: platform of design of digital components; 1) an interface of structural description of electronic components, 2) a generator of code VHDL and finally 3) a translator into a formal language for proving the refinement and the consistency under AtelierB.