Une approche structurelle et comportementale de modélisation pour la vérification de composants VLSI

Catherine Bayol

▶ To cite this version:

Catherine Bayol. Une approche structurelle et comportementale de modélisation pour la vérification de composants VLSI. Autre [cs.OH]. Université Joseph-Fourier - Grenoble I, 1995. Français. tel-00005027

HAL Id: tel-00005027

https://tel.archives-ouvertes.fr/tel-00005027

Submitted on 24 Feb 2004

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

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

THESE

présentée par

Catherine BAYOL

pour obtenir le titre de

Docteur de l'Université Joseph FOURIER – Grenoble I

(arrêté ministériel du 5 Juillet 1984 et du 30 mars 1992)

Spécialité : Informatique

UNE APPROCHE STRUCTURELLE ET COMPORTEMENTALE DE MODELISATION POUR LA VERIFICATION DE COMPOSANTS VLSI

Date de soutenance : 12 Décembre 1995

Composition du jury :

MM.	Jacques VOIRON	Président
	Elie NAJM	Rapporteur
	Paolo PRINETTO	Rapporteur
Mme	Dominique BORRIONE	Directrice de thèse
MM.	Jean–Michel BERGÉ	Examinateur
	Bernard SOULAS	Examinateur

Thèse préparée au sein des laboratoires ARTEMIS/IMAG et ACSAR/DER/EDF

Je tiens à remercier

Monsieur le Professeur Jacques VOIRON, Directeur de l'UFR/IMA, qui a bien voulu me faire l'honneur de présider ce jury de thèse,

Monsieur Paolo PRINETTO, Professeur au Politecnico di Torino, avec qui j'ai eu le plaisir de travailler dans le cadre de projets Esprit et qui a accepté la charge d'être rapporteur,

Monsieur Elie NAJM, Professeur à l'ENST à Paris pour s'être interressé à mes travaux et être rapporteur,

Madame le Professeur Dominique BORRIONE, Directrice du Laboratoire ARTEMIS, qui a dirigé cette thèse ; je lui suis reconnaissante pour son attention, le temps qu'elle m'a accordé et sa constante gentillesse,

Monsieur Bernard SOULAS, Chef du groupe ACSAR de la Direction des Etudes et Recherches d'EDF qui m'a permis de réaliser cette thèse; je le remercie pour le temps qu'il m'a consacré et les connaissances qu'il m'a apportées,

Monsieur le professeur Jean–Michel BERGE pour sa participation au jury.

Je remercie également tous les membres (statutaires, stagiaires, CDD et thésards) des groupes ACSAR, MAI et GECC de la Direction des Etudes et Recherche d'EDF avec qui j'ai partagé de sympathiques moments au cours de ces trois années.

Une pensée affectueuse à Benoit et MMH pour leur aide à la rédaction de ce mémoire.

TABLE DES MATIERES

INTRODUCTION

Chapitre 1 : LE PROJET FICOMP

I – LE CONTEXTE DU PROJET FICOMP

I.1 – Les enjeux du projet **p.5**

I.2 – Une réalisation industrielle **p.6**

I.2.1 – La norme CEI *P.6*

La couche Physique ***p.6***

La couche Liaison de Données ***p.6***

La couche Application ***p.6***

I.2.2 – L'environnement du composant sur le réseau . p.7

I.2.3 – Le composant p.8

a) Ses principales caractéristiques ***p.8***

b) Sa fonctionnalité ***p.8***

I.2.4 – L'originalité de FICOMP p.9

I.2.5 – La partie logiciel : le micro–code p.9

II – DES OBJECTIFS DE VALIDATION

II.1 – Une démarche de conception **p.11**

II.1.1 – Une description au niveau système p.11

II.1.2 – D'une démarche descendante à une réalité ascendante p.11

II.1.3 – Un compromis Software/Hardware p.11

II.1.4 – Une vision d'ensemble p.11

II.2 – Modélisation et validation de la couche Liaison de Données de la norme... **p.12**

II.3 – La validation d'architecture **p.13**

II.3.1 – Une spécification au niveau RTL p.13

II.3.2 – Les principes d'abstraction p.13

II.3.3 – Les objectifs de validation p.14

II.4 – Un objectif de vérification par rapport à la norme p.14

II.4.1 – Le modèle Hardware/Software p.14

II.4.2 – La vérification p.14

III – DU CONTEXTE INDUSTRIEL AUX MOYENS DE VALIDATION

III.1 – VHDL : une norme incontournable p.15

III.1.1 – Une norme internationale p.15

III.1.2 – Des caractéristiques propres p.15

III.1.3 – Pour des besoins industriels p.16

III.2 – Les Algèbres de Processus et le monde de la vérification p16

III.3 – A mi–chemin : VOVHDL p.17

III.3.1 – Le besoin d'un langage intermédiaire p.17

III.3.2 – Cahier des charges p.18

III.3.3 – De VOVHDL vers VHDL p.18

III.3.4 – De VOVHDL vers les outils de preuve p.20

IV – LE POSITIONNEMENT PERSONNEL

IV.1 – Depuis une description VOVHDL vers un modèle vérifiable p.21

IV.2 – Modélisation de la communication VHDL p.21

Chapitre 2 – VOVHDL ET SON INTERPRETATION SEMANTIQUE SOUS FORME DE STE

I – LA SÉMANTIQUE DE COMMUNICATION DE VOVHDL

I.1 – Introduction à VOVHDL **p25**

I.1.1 – Un modèle abstrait p25

I.1.2 – Une structure hiérarchique p25

I.2 – Le support de communication **p26**

I.2.1 – Un processus p26

I.2.2 – Un port p26

I.2.3 – Un canal p27

I.3 – Une sémantique de communication dirigée par des protocoles **p28**

I.3.1 – Les protocoles de communication p28

I.3.2 – Les primitives de communication SEND et RECEIVE p30

I.3.3 – Les transferts d'informations p30

I.4 – Une sémantique de communication synchrone **p30**

II – LES PRINCIPES DE MODÉLISATION

II.1 – Les moyens de modélisation **p32**

II.1.1 – Qu'est ce qu'un STE? p32

II.1.2 – La conservation des traces p32

II.2 – Interprétation STE de chaque processus **p33**

II.2.1 – Interprétation STE d'un processus VOVHDL p33

II.2.2 – La partie générique et la partie Particularisée d'un processus p34

II.2.3 – Le modèle d'un processus VOVHDL sous forme de STE p36

II.2.4 – Une interprétation synchrone : une transition atomique p36

II.3 – Interprétation de la communication p36

II.3.1 – Une propriété fondamentale p36

II.3.2 – Un module de communication pour chaque canal p37

Le principe ***p37***

Un exemple ***p37***

Modélisation de la communication du système ***p37***

Deux points de discussion ***p37***

II.3.3 – Vers un module de communication UNIQUE p38

Une même vision pour tous les canaux ***p38***

Un unique module pour l'ensemble des canaux ***p39***

La modélisation STE du module de communication ***p40***

II.3.4 – Illustration sur un exemple p40

L'interêt d'une approche Prédicat dès que le modèle devient plus complexe ***p41***

II.4 – L'évolution de l'ensemble du modèle p42

III – DANS LA PRATIQUE

III.1 – Une description VOVHDL p44

Une base de donnée ***p44***

La description du système ***p44***

III.2 – Un processus VOVHDL sous forme STE p46

III.3 – Le module de communication p47

Chapitre 3 – LA MODELISATION

I – PRESENTATION DE L'OUTIL ASA+.

I.1 – Un outil industriel **p.51**

I.1.1 – Des applications diverses p.51

I.1.2 – Spécification et analyse : deux besoins p.51

I.2 – Le langage LSA+ **p.51**

I.2.1 – Une architecture statique p.51

I.2.2 – Une sémantique de comportement p.52

I.2.3 – Une sémantique de communication p.52

I.2.4 – Un système temps réel p.52

I.3 – Des moyens de spécification **p.52**

I.4 – Des moyens d'analyse **p.53**

I.4.1 – Validation et Vérification p.53

I.4.2 – Génération de scénarios de test p.53

II – L'ASPECT STRUCTUREL DU MODELE

II.1 – Une structure générique et hiérarchique **p.54**

II.1.1 – Une sémantique de communication distribuée p.54

II.1.2 – Une structure générique p.55

II.1.3 – Conservation du caractère synchrone p.55

III – L'ASPECT COMPORTEMENTAL DU MODELE

III.1 – La cellule de communication (comn.bdy) **p.57**

III.1.1 – Son fonctionnement p.57

III.1.2 – Son interprétation STE p.57

III.1.3 – L'interprétation avec l'outil p.58

III.2 – Le fonctionnement d'un module p.59

III.2.1 – La partie générique p.59

III.2.2 – La partie particularisée p.59

III.2.3 – La fusion des transitions à un niveau n donné p.61

III.2.4 – Un pas de simulation contrôlé p.62

IV – L'ASPECT ENVIRONNEMENT

IV.1 – Un problème de fermeture d'environnement p.62

IV.2 – Le modèle p.62

IV.2.1 – Le module *com_env* *p.63*

IV.2.2 – Un contrôleur d'environnement. p.63

IV.2.3 – Un module scénario p.64

V – L'APPLICATION PRATIQUE

Chapitre 4 – INTERPRETATION DE LA SEMANTIQUE VHDL

I – INTRODUCTION–POSITIONNEMENT–ETAT DE L'ART

I.1 –Un modèle formel de la sémantique VHDL : un besoin **p.69**

I.2 – Les différentes approches en fonction de leur motivation **p.69**

I.2.1 – Les motivations p.69

I.2.2 – Les formalismes en vue de l'utilisation d'outils spécifique p.70

I.2.3 – Les formalismes en vue de l'utilisation de démonstrateurs de théoremes généraux p.74

II – LA SÉMANTIQUE DE COMMUNICATION VHDL

II.1 – Une brève présentation de VHDL **p.77**

II.1.1 – Les caractéristiques de VHDL p.77

II.1.2 – Les objets manipulés p.78

II.2 – Les éléments de communication et de synchronisation **p.79**

II.2.1 –Signaux et pilotes : éléments de communication p.79

1– Les différentes affectations de signaux **p.79**

2– Signaux explicites et implicites **p.81**

3– Les pilotes **p.82**

4– Valeurs courantes, effective et pilotées d'un signal, signal actif **p.82**

II.2.2 – la synchronisation p.83

1– la notion de processus **p.83**

2 –L'instruction Wait : élément de synchronisation **p.84**

II.2.3 – La notion de blocs et de composants p.85

II.2.4 – le temps VHDL p.86

II.3 – La sémantique de simulation **p.86**

II.3.1 – La phase d'élaboration p.87

II.3.2 –La phase de simulation p.87

III – LES PRINCIPES DE MODÉLISATION

III.1 – Définitions et principes p.90

III.2 – Préservation du caractère synchrone p.91

III.3 – Principe de formalisation STe de la simulation VHDL p.92

III.4 – Un processus VHDL p.93

III.4.1 – Son rôle p.93

III.4.2 – Interprétation STE de son comportement p.93

III.4.3 – Modélisation sous forme canonique p.94

III.5 – Le noyau de communication p.96

III.5.1 – Son rôle p.96

III.5.2 – Son comportement p.96

III.5.3 – Son interprétation STE p.97

III.5.4 – Les opérations du noyau p.98

III.6 – La synchronisation de l'ensemble au moyen du Rendez–Vous p.98

III.6.1 – l'aspect synchrone d'un processus p.98

III.6.2 – L'évolution concurrente de l'ensemble p.98

IV – LA MODELISATION

IV.1 – La modélisation des processus p.100

IV.2 – le noyau p.101

CONCLUSION

BIBLIOGRAPHIE

ANNEXE1 : l'exemple du DMA

ANNEXE2 : Depuis les donnees du concepteur jusqu'au modèle ASA+
Application à l'interface utilisateur

INTRODUCTION

Présentation du sujet de thèse

Le sujet de la thèse concerne l'élaboration d'une méthode de mise en oeuvre de description formelle pour la modélisation de composants électroniques en vue de leur vérification au niveau système. Cette étude s'intéresse plus particulièrement au domaine des protocoles de communication, et leur implantation sous la forme de composants micro–programmés.

Dans ce contexte, la thèse se situe aux frontières de deux domaines: les systèmes communicants et les circuits VLSI (Very Large Scale Integrated Circuits). A ce titre, elle s'intéresse à l'interconnexion des techniques de description formelle, communément mises en oeuvre dans chacun des deux domaines suivants, à savoir : l'approche CCS (Calculus of Communicating Systems) et l'approche VHDL (VHSIC Hardware description Langage). La question primordiale est la suivante : comment effectuer la traduction sémantique entre une implantation décrite en VHDL en un modèle élaboré selon la sémantique CCS ?

La réalisation de cet objectif passe par une étude approfondie des modes de synchronisation des processus VHDL afin d'en proposer une représentation formelle sous forme d'une algèbre de communication.

La sémantique de communication du langage VHDL est ainsi clairement identifiée au moyen d'une approche structurelle, consistant à cerner les lois de composition (en terme de synchronisation) des processus VHDL. La sémantique d'exécution qui en découle est alors à rapprocher de l'implantation qui en est faite dans les simulateurs actuels.

En outre, une approche comportementale permet d'interpréter de manière canonique chaque processus sous forme de deux éléments; le premier reflète l'aspect communication du processus et est modélisé de manière générique, le second s'attache à caractériser le comportement propre du processus donné.

Cette étude permet de supporter une méthode de vérification des aspects séquentiels d'un système, décrits en VHDL, sur la base des techniques de bisimulation, telles que celles définies dans le domaine des algèbres de processus.

Le contexte de la thèse

Définition du contexte de FICOMP

Les travaux préliminaires à cette thèse se sont déroulés dans le contexte d'un projet ESPRIT (FICOMP) à caractère industriel.

Ce projet ESPRIT a pour but de favoriser l'émergence de la norme internationale relative au bus de terrain par la démonstration de sa faisabilité. Son objectif est la conception et la réalisation d'un composant ASIC qui supporte les spécifications de la norme applicable à la couche liaison de données (Data Link Layer DLL) FIELDBUS.

Ce projet prévoit également la réalisation d'une carte de communication accueillant le composant FICOMP, ainsi que le logiciel qui implante les spécifications de la couche Application FIELDBUS. Il a permis notamment de montrer la faisabilité opérationnelle de la mise en place de procédures de vérification formelle conjointement aux phases de conception.

contexte de validation / vérification

Les travaux effectués font suite aux travaux de modélisation et de validation de la couche liaison de données du projet de norme FIELDBUS, effectués dans le cadre du projet EUREKA EU68. Elle consiste à appliquer des techniques de vérification formelle à la conception et à la réalisation du composant FICOMP afin de minimiser les risques d'erreurs de conception vis à vis de la conformité de la norme.

A un niveau système, l'objectif est de valider les spécifications de la partie matériel et de la partie logiciel (implantée sous la forme de micro–code), et de vérifier la conformité du comportement résultant de leur coopération par rapport aux spécifications de la norme.

Ce document se décompose suivant les quatre chapitres :

chap1 – Le projet FICOMP

Le premier chapitre a pour objectif de présenter le contexte du projet industriel dans le cadre duquel se sont déroulés les travaux. Après une rapide présentation du composant FICOMP, les objectifs de validation sont énoncés. Nous exprimons les besoins de définir un nouveau langage (VOVHDL) ainsi que les chaînes d'outils qui permettent de relier une description d'architecture, donnée selon le formalisme VHDL, à une description utilisable pour la vérification : c'est à dire suivant un formalisme Algèbre de Processus.

Enfin, la dernière partie de ce chapitre situe mes travaux dans ce contexte.

chap2 – Interprétation CCS du Langage VOVHDL

Afin de faciliter une spécification au niveau système et d'en permettre la vérification, le langage VOVHDL (Vérification Oriented VHDL) a été défini comme un sur–langage de VHDL. Dans une première partie, nous présentons les caractéristiques de ce langage et en particulier les protocoles de communication. En effet, ces derniers ont été introduits pour synchroniser les processus, afin de permettre l'élaboration d'une spécification de haut niveau intégrant la synchronisation de base de VHDL par signaux ou événements. Le langage VOVHDL peut être compilé puis simulé au moyen d'outils VHDL.

Nous présentons ensuite la manière dont nous avons interprété formellement le langage VOVHDL et comment nous avons modélisé cette interprétation selon une sémantique d'exécution synchrone. Cette interprétation a été réalisée au moyen du formalisme des Systèmes à Transitions Etiquetées. Nous distinguons l'aspect communication et l'aspect comportement d'un processus.

chap3 – La modélisation

Afin de pouvoir appliquer ces principes au composant FICOMP, le modèle d'interprétation CCS du langage VOVHDL a été implanté au moyen de l'outil ASA+. Nous rappelons dans ce chapitre les principes de l'outil utilisé, puis, nous montrons comment nous avons appliqué les principes de modélisation du chapitre 2 à un modèle hiérarchique en conservant une approche structurelle et comportementale. Enfin, nous décrivons comment nous avons solutionné le problème particulier de modélisation de l'environnement.

chap4 – La sémantique de communication du langage VHDL

En vue de modéliser directement une description VHDL selon les principes du chapitre 2, nous nous sommes plus particulièrement intéressés à la sémantique de communication du langage VHDL. La première partie de ce chapitre consiste à nous situer par rapport aux travaux existants dans le domaine, puis nous présentons brièvement les aspects de VHDL liés à sa sémantique de communication. Enfin, nous décrivons comment nous avons étendu les principes de notre approche structurelle et comportementale à la sémantique VHDL.

LE PROJET FICOMP

CHAPITRE 1

Nous présentons dans ce chapitre le contexte industriel dans lequel se situe notre travail.

Dans une première partie, un état des lieux du projet nous permet de présenter brièvement le composant FICOMP en terme de fonctionnalité au sein de son environnement sur la carte de communication à laquelle il doit être intégré.

L'objectif global du projet industriel est double : il s'agit non seulement de concevoir le composant mais encore de valider et de vérifier la conformité de cette conception par rapport à une spécification, en l'occurrence le projet de norme CEI FIELDBUS, au cours des différentes phases de conception. Notre travail s'insère dans cette deuxième tâche.

Nous définissons les objectifs de validation qui consistent dans un premier temps à valider une partie de l'architecture hardware du composant, puis à valider la conformité du comportement de l'ensemble du composant vis à vis des modèles de la norme. Dans cette intention, une démarche de conception est alors bien définie.

Nos objectifs de validation nous positionnent dans un domaine charnière entre le monde des concepteurs dont le langage de description est la norme VHDL et celui des Algèbres de Processus qui engendre des outils de validation performants. Afin de combler une partie du fossé qui sépare ces deux univers, nous nous proposons de définir un langage orienté vérification et nous présentons les chaînes d'outils nécessaires pour se diriger soit vers la berge où l'on s'exprime en VHDL , soit vers celle où l'on fait de la validation.

I – LE CONTEXTE DU PROJET FICOMP

I.1 – Les enjeux du projet

Aujourd'hui interconnectés en fil–à–fil, les constituants d'automatisme que l'on trouve au sein d'un processus (automates programmables, capteurs, actionneurs) pourront demain communiquer sur la base d'un réseau de communication, offrant ainsi une plus grande qualité de services ainsi qu'une plus grande ouverture et évolutivité. Un tel réseau, dit de terrain (FIELDBUS), se distingue des réseaux de communication type télécommunication ou informatique par les caractéristiques suivantes :

- il doit permettre un grand nombre de connexions,
- il doit être dimensionné afin de transmettre un grand volume d'informations. Cependant ces informations (température, pression, consigne...) sont de petite taille ; leur codage ne nécessite que quelques octets seulement,
- il doit garantir les performances “ temps réel ” que nécessite le processus,
- il doit répondre à des exigences fortes de sûreté de fonctionnement et de sécurité ; en particulier, il doit supporter la redondance du médium de communication,
- il doit enfin répondre à des critères de coût non seulement en termes de prix mais aussi en termes de conditionnement ; le passage à cette nouvelle technologie doit pouvoir se faire selon un coût raisonnable, d'autant qu'il touche l'ensemble des constituants d'automatisme du processus.

Le projet FICOMP* consiste à développer les différents éléments Hardware et Software, permettant la connexion au réseau de communication, spécifié par la norme internationale CEI FIELDBUS (CEI = Commission Electrotechnique Internationale).

L'enjeu de ce projet consiste donc à se positionner sur ce marché potentiel au plus tôt et avec la solution la plus adéquate répondant aux exigences d'un tel réseau de communication.

C'est pourquoi, les concepteurs se sont orientés vers le développement d'un ASIC (nommé FICOMP pour FIeldbus COMPonent) micro–programmé. Il convient en effet de garantir l'évolutivité d'un tel composant, grâce à sa partie logiciel, dans la mesure où il a été décidé de lancer le projet avant même la stabilisation de la norme. De plus, plusieurs normes de réseaux de terrain existent déjà (en particulier la norme française FIP et la norme allemande PROFIBUS) ; la conception du composant FICOMP doit permettre de supporter ces différentes normes et conduire ainsi à des équipements multi–standards.

Afin de faire face à ces enjeux techniques et économiques, ce projet a été organisé sous couvert d'une Assurance Qualité, qui repose sur deux éléments primordiaux concernant le composant FICOMP :

- la mise en oeuvre de techniques formelles de validation et de vérification de la conception au niveau système,
- la mise en oeuvre d'une chaîne complète de CAO (Conception Assistée par Ordinateur) et de synthèse pour l'obtention du composant à partir d'une spécification VHDL au niveau RTL (Register Transfer Level).

FICOMP Projet ESPRIT n° 6526 ; partenaires : CEGELEC/ EDF/ MARCONI/ SEMI_CONDUCTORES/ SOFTING et la collaboration du Politecnico di Torino pour les activités de modélisation.

I.2 – Une réalisation industrielle

I.2.1 – La norme CEI

Afin de permettre le déploiement industriel du nouveau concept ”réseau de terrain”, les industriels ont consenti un effort considérable à la normalisation. En tant que norme de communication, la norme FIELDBUS ne pouvait échapper à l'architecture conceptuelle préconisée par la norme OSI.

La norme OSI (Open Systems Interconnection) de l'ISO (International Standard Organisation) est une norme internationale proposant de décrire les interconnexions des systèmes ouverts selon une structuration de base en couches. On considère que chaque système ouvert est composé d'un ensemble ordonné de sous–systèmes que l'on représente dans un ordre vertical depuis la couche de plus bas niveau (la plus proche du support physique) vers celle du niveau supérieur (la plus abstraite). L'ensemble des sous–systèmes de même rang constitue une couche du modèle de référence. Un sous–système est constitué d'une ou plusieurs entités. Une entité n'a d'interactions qu'avec les niveaux immédiatement supérieur et inférieur.

Le modèle de référence de la norme OSI est un modèle en sept couches : depuis la couche Physique jusqu'à la couche Application et passant successivement par les couches Liaison de Données, Réseau, Transport, Session et Présentation. Pour des raisons d'efficacité (temps de réponse, sécurité, accessibilité aux variables) la norme associée aux réseaux de terrain est une norme en trois couches seulement : il s'agit des couches Physique, Liaison de Données et Application.

La couche Physique

Elle doit fournir les moyens mécaniques, électriques, fonctionnels et procéduraux qui sont nécessaires à l'activation, au maintien et à la désactivation des connexions physiques. Ces connexions physiques sont destinées à transmettre des éléments binaires (transmission de niveau bit) entre entités de liaison. Cette couche a pour objectif de conduire les éléments binaires jusqu'à leur destination sur le support physique. Plusieurs normes proposent les adaptations nécessaires selon le support physique (paire torsadée, fibre optique, radio).

La couche Liaison de Données

Elle offre des services à la couche supérieure, la couche Application, et utilise les services de la couche inférieure : la couche Physique. La couche Liaison de Données assure le contrôle de l'accès au médium en réception ou en émission (sous couche MAC) et doit résoudre les problèmes de conflit d'accès entre le réseau et la couche Application. Afin de répondre aux exigences temps réel, la couche Liaison de Données FIELDBUS a la particularité d'intégrer la base de données du processus au sein même de l'architecture de communication. Elle est ainsi constituée d'un ensemble de tampons produits et consommés. Ces tampons contiennent les dernières valeurs mises à jour soit par l'utilisateur, soit par le réseau. Elle offre deux types de services de transmission : les échanges de variables identifiées et les transferts de messages.

La couche Application

En tant que couche la plus élevée du modèle de référence, elle doit donner au processus d'application le moyen d'accéder à l'environnement OSI. Cet environnement est lié au système représenté. Il s'agit généralement d'un ensemble qui peut inclure un ou plusieurs ordinateurs, les logiciels associés, des périphériques, des terminaux, des opérateurs humains, des processus physiques ou des moyens de transfert d'informations. Chaque fois que des communications ont lieu dans l'environnement OSI, c'est au travers de la couche Application que tous les paramètres à spécifier concernant le processus d'application sont portés à la connaissance de l'environnement OSI.

I.2.2 – L'environnement du composant sur le réseau

Le projet consiste donc à développer un environnement complet de communication depuis le niveau utilisateur jusqu'à celui de réseau de terrain. FICOMP est un co–processeur de communication. Il est placé sur une carte de communication qui comprend son propre processeur (le processeur hôte). Le processeur du PC sur lequel est insérée la carte sera appelé processeur utilisateur.

Nous décrivons ci–dessous les interconnexions du co–processeur de communication avec l'utilisateur d'une part et le réseau d'autre part.

La communication du composant FICOMP vers l'utilisateur se fait au moyen d'un bus système sur lequel le processeur hôte et la mémoire de la carte échangent des informations au moyen d'une vingtaine de signaux.

Le composant FICOMP reçoit et émet des informations sur le réseau de terrain par l'intermédiaire d'un coupleur.

Il est également relié à un bus de mémoire privée qui lui permet d'accéder à une RAM (Random Access Memory) qui est une base de données et à un programme sous forme d'une PROM (Program Read Only Memory). La base de données est une base de données du réseau. Elle permet par un mécanisme de recherche dichotomique d'identifier rapidement une donnée qui passe sur le réseau et de déterminer si elle concerne ou non le co–processeur. En particulier, les informations stockées dans cette base de données sont des adresses ou des tailles de données.

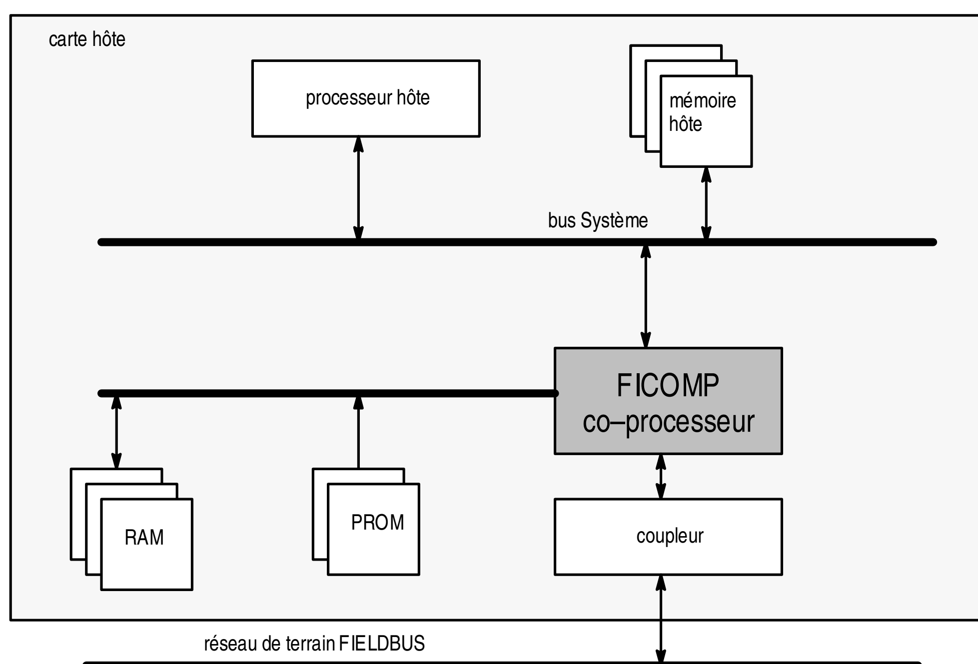


Figure 1 : L'environnement FICOMP et sa carte de communication

I.2.3 – Le composant

a) Ses principales caractéristiques

FICOMP est un co–processeur de communication gérant principalement les fonctionnalités de la couche Liaison de Données. Il doit disposer de deux interfaces afin de communiquer d'une part avec le processeur hôte de la carte et d'autre part avec le réseau. Les requêtes de l'utilisateur et les événements du réseau peuvent être totalement asynchrones. Le circuit doit résoudre tout conflit, lors d'accès aux bases de données qui peut en découler. C'est pourquoi le composant FICOMP est composé d'un CPU bi–tâche et bi–contexte qui doit réagir en temps réel. Lorsqu'une tâche devient prioritaire, un basculement de tâche doit pouvoir être réalisé en zéro cycle d'horloge.

En outre, ce CPU est un CPU en tranches ce qui permet une évolutivité plus grande. Pour des besoins de rapidité de transfert d'informations, il est à micro–instructions câblées et dispose d'une mémoire de communication privée. Il est conçu sur la base d'une architecture micro–programmée RISC (Reduced Instruction Set Circuit). L'unité centrale contient une mémoire micro–programmée constituée d'une ROM et d'une RAM.

L'architecture FICOMP décrit un circuit intégré composé de quatre blocs fonctionnels : l'interface utilisateur (UI), l'accès à la mémoire directe (DMA), l'interface réseau (NI) et le CPU. C'est par l'intermédiaire de ces quatre blocs que le composant FICOMP gère les échanges d'informations sur les trois bus (bus système, bus réseau et bus mémoire).

Ces quatre blocs communiquent entre eux au moyen de FIFO.

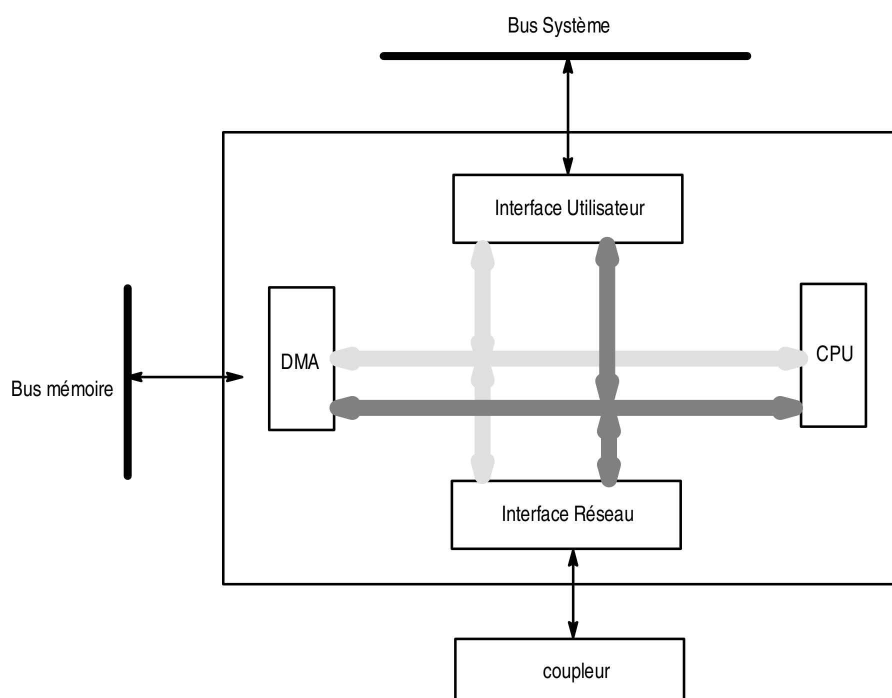


Figure 2 : Architecture du co–processeur FICOMP

b) Sa fonctionnalité

L'interface utilisateur décode les requêtes de l'utilisateur (ordres d'écriture, de lecture, d'exécution) et il contrôle les transactions entre le composant FICOMP et les éléments de la carte hôte. Pour cela, il communique avec le DMA et le CPU.

L'interface réseau gère l'émission ou la réception d'informations issues du réseau par l'entremise d'un support de communication qui sera suivant l'implémentation choisie un coupleur/câbles, un coupleur/fibres optiques ou un coupleur/radio. Le coupleur est composé d'un émetteur, d'un récepteur et de machines d'états qui assurent des opérations de temps critique.

L'unité centrale effectue l'exécution du micro–code en supervisant l'activité des autres blocs. Ce bloc, constitué du CPU, a été conçu pour supporter l'exécution de deux tâches concurrentes. L'objectif est de pouvoir exécuter les requêtes de l'utilisateur et les événements du réseau de manière totalement indépendante. L'unité centrale adresse des registres internes et des FIFO et dialogue avec l'extérieur au moyen de bus (Bus CPU et Bus DMA) ; sur chaque bus transitent des commandes, des données et des lignes d'états (status line).

L'accès à la mémoire privée externe est agencé par le DMA. Les données associées sont chargées dans des FIFO qui se comportent comme des tampons entre le DMA et les autres unités. Une fois initiés par l'unité centrale, les transferts de la mémoire vers l'utilisateur ou le réseau sont supportés de manière autonome par le DMA. Les informations vers les périphériques sont gérées par sept voies. L'accès aux informations sur ces sept voies se fait selon le mécanisme du tourniquet permettant ainsi une gestion des flots de données de manière équitable sur chacune d'elles.

Les différents flots de données peuvent se regrouper en quatre catégories :

- Les échanges entre l'utilisateur et la mémoire (par UI et DMA) : 1 voie,
- Les échanges entre le réseau et la mémoire (par NI et DMA) : 2 voies,
- La communication entre le CPU et la mémoire (par le DMA) : 4 voies. Il s'agit en fait de gérer les informations des deux tâches concurrentes T0 (vers le réseau) et T1 (vers l'utilisateur). Il y a deux voies pour chacune de ces deux tâches.
- La recherche dichotomique dont la gestion est prioritaire.

I.2.4 – L'originalité de FICOMP

Ce composant présente des aspect originaux que nous rappelons brièvement.

La base de données est variable à l'intérieur du réseau et est donc placée à un niveau très bas dans le réseau (couche de niveau deux), ceci pour des raisons d'efficacité : il s'agit en effet de garantir des besoins temps réel.

Le CPU est un CPU double tâche.

Le DMA est autonome : une fois initialisé par le CPU, il agit de manière indépendante. Aussi il faudra gérer le parallélisme entre DMA et CPU.

Les deux interfaces évoluent de manière asynchrone ; deux problèmes importants sont à considérer : la gestion du parallélisme et la modélisation de la priorité d'une requête réseau par rapport à une requête utilisateur.

I.2.5 – La partie logiciel : le micro–code

La partie logiciel est composée de deux logiciels exécutés en parallèle qui correspondent à deux tâches indépendantes. Ces deux tâches s'exécutent de manière totalement asynchrone. La première est associée à la gestion de l'utilisateur, la seconde gère le réseau.

Le CPU est vu comme une boîte comportementale qui appelle les autres modules (DMA, interface réseau, interface utilisateur) comme des ressources. Il est vu comme une interface dont le logiciel traduit l'utilisation séquentielle des ressources que constituent les autres blocs.

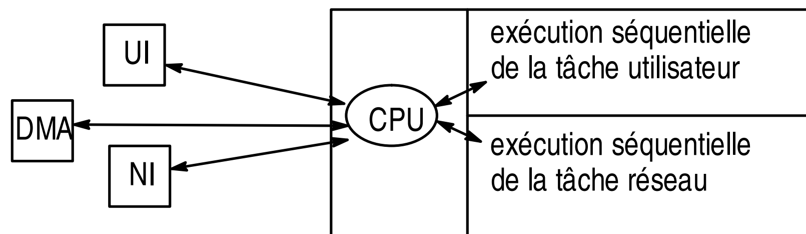


Figure 3 : Le CPU, interface entre le micro–code et les autres blocs de l'architecture

Cependant, cette interprétation oblige le CPU à gérer la synchronisation des deux parties : Hardware et Software. Comme nous le verrons plus loin, la modélisation de la partie Hardware est exécutée dans un langage de haut niveau (VOVHDL) dont la sémantique d'exécution se compte suivant un certain pas de simulation (δ_{VOVHDL}). La partie micro–code est modélisée suivant le même principe dans un langage de haut niveau PL/F1 mais dont le pas de simulation n'est pas forcément compatible avec celui de VOVHDL. Il s'agit donc de synchroniser l'exécution des deux parties pour obtenir le comportement global.

II –DES OBJECTIFS DE VALIDATION

II.1 – Une démarche de conception

II.1.1 – Une description au niveau système

Il est essentiel de décrire un système au moyen d'un langage spécifique dont les règles sémantiques et syntaxiques assurent une interprétation non ambiguë. Ainsi, une description formelle permet de définir une spécification dans un langage autre que le langage naturel. De plus, les techniques de description formelle possèdent un formalisme mathématique sous–jacent qui fournit un moyen de démontrer les propriétés essentielles d'un système. Dans notre application, il était essentiel de partir d'une modélisation au niveau système. Comme la spécification de départ est issue d'une norme, cela implique un point de départ de haut niveau qui correspond à une approche système.

II.1.2 – D'une démarche descendante à une réalité ascendante

La conception d'un composant se fait selon une démarche descendante depuis une spécification de haut niveau vers une implantation de plus bas niveau. Il aurait été souhaitable de pouvoir conserver une démarche descendante de bout en bout du projet. Or, les concepteurs ont souhaité réutiliser certaines spécifications extraites d'un autre composant, le FULLFIP qui supporte la norme française FIP. Le composant FULLFIP ayant été anciennement conçu sans chaîne CAO VHDL, des outils de conversion automatique ont été utilisés afin d'extraire du layout FULLFIP certaines parties décrites en VHDL au niveau RTL. La conception du composant FICOMP s'est donc faite au niveau RTL et a consisté à retravailler certaines parties du composant FULLFIP et à les compléter afin d'obtenir un composant beaucoup plus performant (citons par exemple le fait que le DMA soit entièrement autonome).

II.1.3 – Un compromis Software/Hardware

L'implantation du co–processeur de communication est réalisée suivant un compromis entre une solution totalement Hardware et une solution entièrement Software. La raison est que la partie Software garantit une certaine évolutivité afin de suivre les modifications de la norme, non encore stabilisée au début du projet. De son coté, la partie Hardware se justifie par les besoins de performance du composant (en terme de rapidité) ; citons par exemple le décodage d'adresse réalisé par recherche dichotomique câblée.

II.1.4 – Une vision d'ensemble

Le schéma suivant nous montre les interactions entre le monde des concepteurs et le monde plus abstrait des modélisateurs.

Depuis les documents normatifs, écrits en langage naturel (en anglais), un ensemble de modèles du protocole a été défini. Ce sont ces modèles de la norme qui ont pu être validés dans le cadre d'un projet Eureka (Eu68). Ce premier travail de modélisation a permis aux concepteurs de définir une spécification du protocole en distinguant une partie Software et une partie Hardware. A partir de ces spécifications, les concepteurs :

- d'une part, intègrent certaines spécifications du composant FULLFIP déjà existant (obtenu par extraction automatique) à la spécification Hardware. La spécification RTL ainsi obtenue est alors à un niveau de description trop bas pour effectuer une validation au niveau système. Aussi une démarche d'abstraction (donc ascendante) est mise en place afin d'obtenir une spécification système de la partie Hardware.
- d'autre part, décrivent la partie Software dans un langage de haut niveau (PL/F). Cette description peut ensuite être compilée de manière automatique pour fournir le micro–code exécutable ;

A partir de la modélisation système du Hardware, des techniques de validation d'architecture peuvent être appliquées. Puis la spécification Software peut être intégrée à cette modélisation afin de vérifier le modèle des concepteurs par rapport aux modèles issus de la norme.

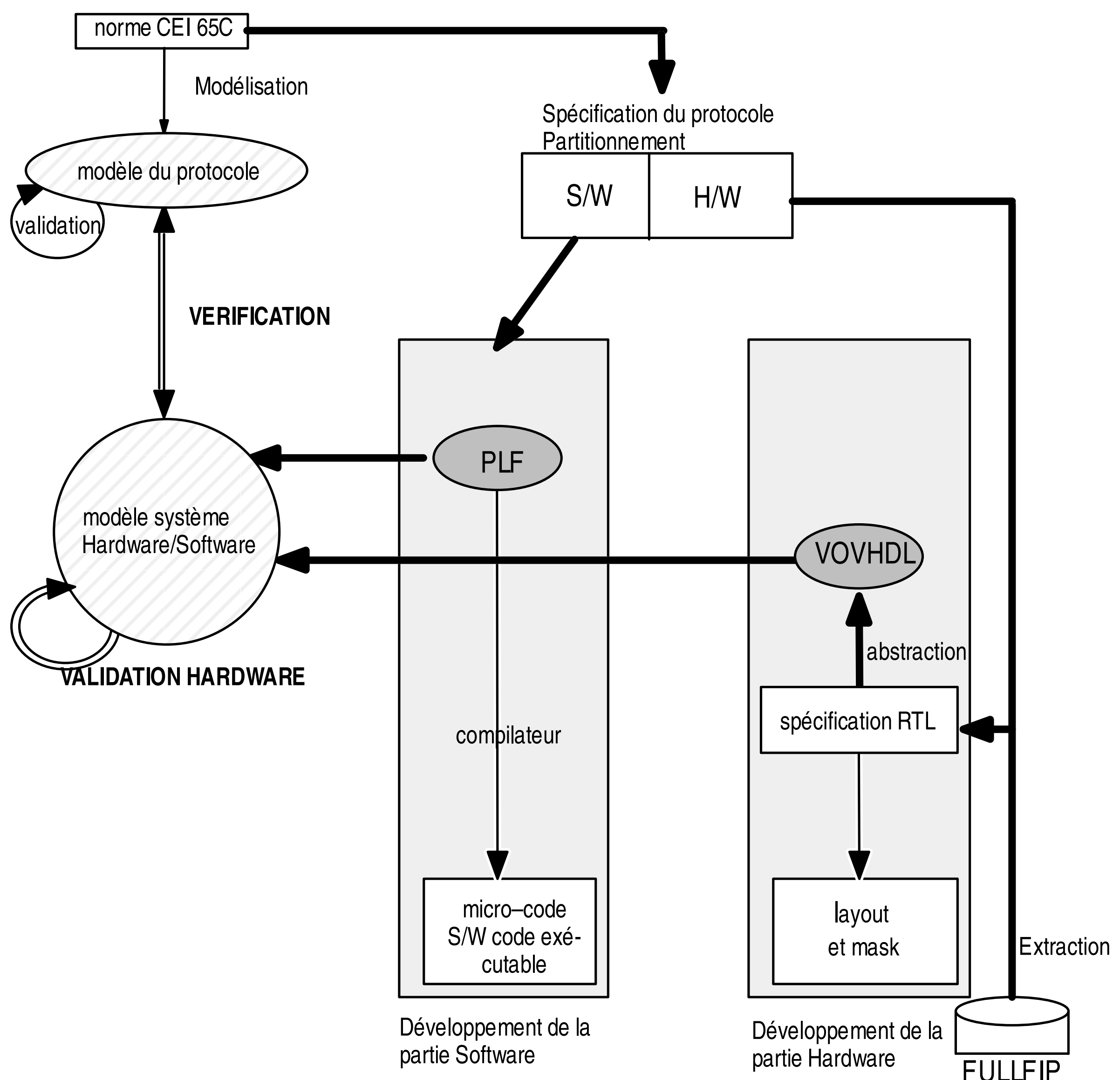


Figure 4 : Relations entre activités de validation/vérification et activités de conception

II.2 – Modélisation et validation de la couche Liaison de Données de la norme

Les opérations de validation effectuées dans le projet FICOMP se positionnent essentiellement au niveau de la couche Liaison de Données. La modélisation et la validation de la norme de la couche Liaison de Données ont été réalisées dans le cadre du projet Eu68. Les résultats de ces travaux constituaient une donnée d'entrée du projet FICOMP.

La modélisation peut se faire en distinguant les deux aspects de la norme : d'une part les mécanismes mêmes du protocole (validation des mécanismes d'échanges), d'autre part les services offerts par la couche Liaison de Données à la couche Application (vérification de la conformité des mécanismes de protocole vis à vis de la spécification des services offerts).

C'est notamment pour ces besoins de validation et de vérification qu'une approche Algèbre de Processus a été adoptée. Ce choix implique l'utilisation des Algèbres de Processus comme sémantique des outils de preuves employés dans la suite du projet.

II.3 – La validation d'architecture

II.3.1 – Une spécification au niveau RTL

En vue de valider l'architecture du composant, nous avons besoin d'une description au niveau système. La modélisation de la spécification RTL, obtenue suite à la démarche mixte descendante/ascendante, nous conduirait à une machine d'états constituée de beaucoup trop d'états et de transitions pour être manipulée par les outils de validation. En outre, les niveaux de description de la spécification RTL d'une part et des modèles du protocole d'autre part sont trop éloignés pour être vérifiables l'un par rapport à l'autre. Aussi, il a été indispensable de définir un niveau de spécification d'architecture plus abstrait. Des principes d'abstraction depuis le niveau RTL vers un niveau système ont donc été appliqués.

II.3.2 – Les principes d'abstraction

Le modèle abstrait sur lequel est appliqué le procédé de validation doit refléter les caractéristiques principales issues du comportement observable externe du système. C'est pourquoi, les mécanismes de bas niveau peuvent être ignorés du moment qu'ils ne modifient en rien le comportement observable du système. L'obtention d'un tel modèle provient d'une méthodologie d'abstraction en trois étapes. Cette méthodologie d'abstraction est connue sous le terme de trois quart types d'abstraction : temporelle, de données et fonctionnelle.

Dans un premier temps, des règles de simplification ont été définies. Le choix de ces règles provient du fait que certaines caractéristiques du circuit ne sont pas des caractéristiques à prendre en compte au niveau système. Citons par exemple les règles du type : 'Les opérations de circuit synchrones ne sont pas prises en compte'. La raison est qu'une horloge globale n'est pas à modéliser au niveau système. Une autre règle est : 'Le mode d'accès 8 bits n'est pas modélisé'. En effet, au niveau système la modélisation fait appel à un type abstrait.

Dans une deuxième phase, il s'agit de réduire le modèle. La démarche consiste à analyser les signaux situés en interface de chacun des blocs élémentaires qui constituent le système. Pour chaque signal, il faut déterminer s'il peut être supprimé, modifié ou réduit et cela par rapport aux règles de simplifications définies précédemment. Ce procédé de réduction induit deux étapes supplémentaires : la suppression de registres et la suppression de signaux internes.

En dernière étape, il s'agit d'analyser la sémantique du langage initial de spécification de manière à pouvoir exprimer le comportement du système selon une sémantique de plus haut niveau. Dans le cadre de notre étude, l'abstraction s'est appliquée depuis une description VHDL au niveau RTL vers une description VOVHDL (cf chap2). Cette étape a suivi également une méthodologie précise. La décomposition structurelle du modèle a été optimisée en tenant compte des aspects fonctionnels et du degré de parallélisme interne : le modèle a été décomposé en blocs fonctionnels en tenant compte des multiples activités des signaux. Par exemple, de manière à éviter des structures trop complexes, le mode de communication multi–receive de VOVHDL a été utilisé. Puis la caractérisation de la communication entre blocs entraîne la définition des canaux de communication avec leurs émetteurs, leurs récepteurs et le protocole de communication associé. Dans le cadre de VOVHDL ces protocoles sont d'un niveau d'abstraction élevé et le choix d'un protocole ou d'un autre lors de cette phase d'abstraction suppose la connaissance du comportement de chaque bloc vis à vis de cette communication. Enfin, une abstraction comportementale est traitée en déterminant l'évolution des variables internes et des sorties en réponse à des événements d'entrée. Le comportement de chaque bloc terminal d'une part est cohérent avec les spécifications VHDL initiales et d'autre part est facilement traduisible en un langage du niveau système (en l'occurrence VOVHDL).

II.3.3 – Les objectifs de validation

L'objectif de validation consiste à vérifier que le modèle de l'architecture système offre les propriétés attendues.

Une tâche de validation traite deux sortes de propriétés :

- les propriétés intrinsèques directement liées au formalisme mathématique utilisé pour exprimer la spécification. Ces propriétés concernent les aspects dynamiques de la spécification. Citons par exemple la vérification de l'absence d'états de blocage ou puits, la vérification de la réinitialibilité, la détection de boucles.
- les propriétés externes représentatives du comportement attendu du système. Dans ce cas la tâche de validation est plutôt vue comme une tâche de vérification et est basée sur les mêmes techniques : il s'agit des techniques de bi–simulation, satisfactions de propriétés, équivalence de tests et pré–ordre.

Les modèles des trois blocs principaux Interface Utilisateur, Interface Réseau et DMA on été obtenus selon le mécanisme d'abstraction présenté précédemment. Ces modèles sont décrits selon la sémantique du langage VOVHDL et sont traduits dans le langage de l'outil de validation ASA+ (basé sur le formalisme des réseaux de Pétri et la sémantique de l'Algèbre CCS). Des scénarios sont déterminés et des techniques de Model Checking permettent de les valider sur le graphe d'états obtenu à partir de la simulation du modèle. Ces scénarios sont déterminés en fonction d'une utilisation nominale du composant (par exemple le respect d'un certain ordre des requêtes). Dans un premier temps, il s'agit de déterminer des benchmarcks significatifs. Cependant, en vue d'appliquer des techniques de Model Checking il est nécessaire de fermer le modèle par le micro–code et de déterminer des scénarios en fonction de ce micro–code.

II.4 – Un objectif de vérification par rapport à la norme

II.4.1 – Le modèle Hardware/Software

Afin de valider l'ensemble du composant et non plus seulement la partie Hardware, il est nécessaire de fermer le modèle par la partie micro–code. Cette fermeture est différente de celle citée dans le paragraphe précédent . En effet, il s'agit ici non plus de considérer des benchmarks associés au comportement spécifié du micro–code, mais d'intégrer le comportement réel du micro–code, issu de sa modélisation Software. Pour obtenir le modèle global, il nous reste à intégrer la partie Software à la partie Hardware validée précédemment.

La partie Software est décrite dans un langage de haut niveau (PL/F) afin d'appliquer une méthodologie descendante. A partir d'une description de la partie micro–code en PL/F, un compilateur crée de manière automatique le code exécutable. Mais l'intérêt de la formalisation PL/F est de permettre l'intégration du comportement de la partie Software au composant Hardware, sur lequel la vérification est possible. La démarche d'intégration d'une description PL/F dans un modèle en vue de la vérification est la même que pour la partie Hardware sur le langage VOVHDL. Il s'agit donc d'exprimer la sémantique de communication de PL/F dans la sémantique de communication de l'Algèbre CCS (fondement de l'outil ASA+).

II.4.2 – La vérification

Le but de la vérification est d'établir que l'implantation est correcte vis à vis de la spécification c'est à dire qu'il existe une relation entre deux entités qui prennent en compte des aspects significatifs.

Indépendamment de la forme dans laquelle est donnée soit la spécification, soit l'implantation, différentes relations peuvent être définies : l'équivalence de trace [KO, 78], l'équivalence observationnelle [Mi, 89], des propriétés temporelles [BCDM, 86] ou des relations de tests [DNHe, 84].

Notre objectif de vérification est d'établir que le produit des concepteurs reflète les exigences fonctionnelles de la norme. Il s'agit de vérifier que le comportement global (c'est à dire le composant partie Hardware et partie Software) satisfait aux spécifications de la norme.

En particulier, le fonctionnement des deux tâches concurrentes et la gestion des autres blocs par le CPU doivent être vérifiés de telle sorte que ce qui est décrit conduise à un fonctionnement correct du système.

La norme se caractérise à la fois par les services (représentatifs des échanges vis à vis de l'environnement) et par les mécanismes (fonctionnement propre des protocoles). Les services peuvent être obtenus après projection observationnelle des mécanismes.

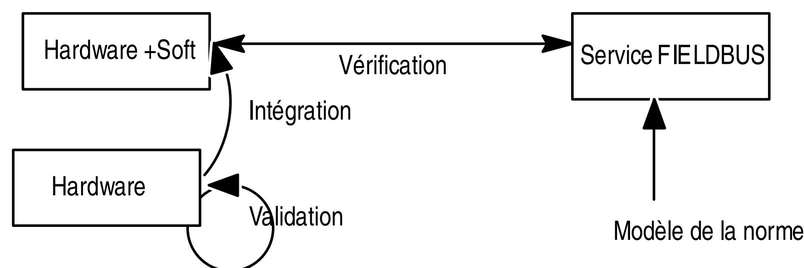


Figure 5 : La vérification par rapport aux modèles de la norme

En final, il s'agira donc de vérifier que les services du modèle Hardware/Software sont conformes à ceux définis dans la couche Liaison de Données de la norme.

III – DU CONTEXTE INDUSTRIEL AUX MOYENS DE VALIDATION

III.1 – VHDL : une norme incontournable

III.1.1 – Une norme internationale

VHDL (VHSIC Hardware Description Language) est un langage de description de matériel qui a été mandaté par le DoD (Ministère de la Défense américain) lors d'un programme d'étude sur les circuits intégrés de type VHSIC (Very High Speed Integrated Circuits). C'est un langage issu d'une autre norme : le langage ADA.

Dans le domaine des circuits intégrés, les changements rapides de technologie, en vue de réduire le cycle de vie des produits sans pour autant modifier le cahier des charges, ont conduit le DoD à promouvoir un langage dédié à la conception du Hardware mais indépendant de la technologie utilisée. Ce langage approuvé par IEEE est reconnu comme norme internationale. La première version de la norme VHDL est parue en 1987 sous la référence IEEE_1076. Cette version a été révisée et approuvée en 1993. Il devient de plus en plus difficile de concevoir un circuit ASIC sans une documentation associée décrite en VHDL. En effet, de nombreux outils de CAO ont pour langage de référence la norme VHDL.

III.1.2 – Des caractéristiques propres

VHDL est un langage de conception de systèmes, de cartes électroniques et de circuits intégrés ; à ce titre, il dispose de différents niveaux de description, depuis le niveau système jusqu'au niveau porte logique.

Dans une conception méthodologique descendante, une même fonction définie dans un environnement intégré peut être décrite à différents niveaux. On peut décliner la définition de cette fonction depuis une phase de spécification attachée à une description comportementale de haut niveau vers une phase de conception du circuit décrite dans un mode structurel.

L'organisation d'une description VHDL se fait autour de la notion de composant (couple entité/architecture). Chaque composant ou entité comporte une description de ses interfaces avec l'extérieur (les ports d'entrée et de sortie) en plus de sa propre structure interne. Pour un composant, la description interne peut se faire soit sous forme comportementale soit sous forme structurelle par assemblage d'autres composants, soit sous forme mixte par l'utilisation de fonctions de type comportemental avec des fonctions de type structurel. Ces sous–ensembles ou composants font l'objet de déclarations qui définissent les entrées/sorties qui les relient entre eux.

La description d'un système à un haut niveau d'abstraction permet de simuler son comportement. Le lien d'une description VHDL à un simulateur logique permet à l'ingénieur de concevoir son système dans une approche méthodologique hiérarchisée descendante. Cela devient primordial lorsque la complexité des circuits grandit. Un simulateur VHDL voit chaque processus comme une boîte noire qui réactualise l'état de ses sorties en réponse aux événements produits sur les entrées, compte tenu des caractéristiques dynamiques et des paramètres temporels du modèle.

III.1.3 – Pour des besoins industriels

Dans le cadre d'un projet multi–partenaires où plusieurs concepteurs de différentes entreprises sont amenés à échanger des informations, le langage commun VHDL est un moyen sûr de transmettre ces informations en minimisant les distorsions. Il fournit un moyen d'échange et de dialogue efficace entre l'ingénieur qui conçoit les fonctions du système conformément aux spécifications techniques du cahier des charges et le technicien qui réalisera et testera le système.

Une autre motivation industrielle est de pouvoir définir des circuits de manière modulaire et de pouvoir créer des bibliothèques de composants en vue d'une réutilisation.

Enfin, dans le projet FICOMP où une implantation ASIC est choisie, le langage VHDL fournit une description macroscopique du circuit. Cette description de haut niveau nous permet alors de conserver une liberté dans le choix du fabricant du circuit ainsi que dans le choix de la technologie employée.

Ce langage supporte toutes les phases de conception Hardware : développement, simulation, documentation, synthèse, test, vérification.

III.2 – Les Algèbres de Processus et le monde de la vérification

Les Algèbres de Processus ont surtout été développées pour modéliser les comportements concurrents des processus (des agents dans la terminologie des Algèbres de Processus). Les notions de hiérarchie, de modularité, de communication, d'abstraction et de parallélisme sont rigoureusement définies dans un formalisme mathématique qui supporte des algorithmes de preuve.

Les Algèbres de Processus telles que CCS sont des systèmes de description formelle qui permettent d'exprimer au mieux les besoins de la vérification formelle. En effet, elles sont le support de beaucoup d'outils de preuve (d'équivalence ou de propriétés) et elles permettent l'extension à de nouveaux opérateurs.Tout d'abord, nous nous situons au niveau système et à ce stade nous devons pouvoir gérer les actions échangées avec l'extérieur, partitionner les échanges de manière fonctionnelle, gérer la synchronisation, avoir la possibilité de masquer des informations à différents niveaux d'abstraction.

Dans la sémantique des Algèbres de Processus, un processus ou un agent est vu comme une boîte noire qui échange avec son environnement des événements ou des actions au moyen de ports de communication. Les agents peuvent être reliés entre eux par des canaux de communication, matérialisés par la connexion de deux ports de deux processus différents. Si une action n'est pas observable par l'environnement, elle est dite interne : dans ce cas, le processus a évolué de manière interne sans communication vers son environnement. Un ensemble d'opérateurs est défini et permet de faire évoluer ou de définir le

comportement d'un ensemble d'agents suite à une succession d'événements. Trois types de sémantiques peuvent être définis sur les opérateurs d'une Algèbre de Processus [Le, 90].

La sémantique opérationnelle est donnée par la définition d'axiomes et de règles sur chaque opérateur afin de construire un modèle de chaque processus. Ces modèles sont généralement donnés sous forme de graphes: il s'agit de Systèmes à Transitions Etiquetées, d'automates ou de graphes de synchronisation. Différentes relations d'équivalences sont alors définies afin d'identifier deux graphes différents. Citons par exemple l'équivalence de trace (ou langage) [Man 74] ou l'équivalence de bisimulation [Par 81].

Une sémantique axiomatique consiste à définir un ensemble d'axiomes et de règles d'inférence. Chaque axiome est une équation dont les deux membres sont des expressions algébriques sémantiquement équivalentes. Des sémantiques axiomatiques ont été définies pour CCS[Hem 85] et ACP [BeK, 88].

Enfin une sémantique dénotationnelle permet d'exprimer un comportement sous forme d'un objet d'un domaine mathématique et de définir le comportement d'un système comme une fonction d'expression comportementale de sous–systèmes. Les procédés récursifs et les notions de points fixes sont alors couramment utilisés. Des sémantiques dénotationelles sur CSP (sémantique des traces) et sur CCS ont été définies.

Dans notre travail, la sémantique opérationnelle de CCS [Mi, 80] est exprimée sous le formalisme des Systèmes de Transitions Etiquetées. Parmi d'autres Algèbres de Processus, nous pouvons citer : CSP [Ho, 85], Circal [Mil, 85], LOTOS [Br, 88], [NSF, 95] et ACP [BK, 85].

Les réseaux Prédicats/Transitions Etiquetés constituent un formalisme de plus haut niveau permettant de maîtriser la modélisation de systèmes complexes et dont la sémantique comportementale et compositionnelle est conforme à celle de l'Algèbre CCS [Llo, 90].

Les outils qui utilisent ces techniques permettent l'analyse des aspects comportementaux des systèmes séquentiels. Les modèles se présentent sous forme de modules hiérarchiques échangeant des informations sur des canaux de communication au moyen de ports d'interaction. Les modules communiquent entre eux selon le mode de composition par rendez–vous de l'Algèbre CCS.

III.3 – A mi–chemin : VOVHDL

III.3.1 – Le besoin d'un langage intermédiaire

Nous avons vu que la démarche des concepteurs de notre projet n'était pas une démarche descendante depuis une spécification de haut niveau vers une implantation finale mais plutôt une démarche mixte. Bien que la méthodologie d'une démarche descendante soit appliquée dans un premier temps, les besoins d'intégration de certaines parties du composant déjà existant ont obligé les concepteurs à une abstraction depuis une description de bas niveau vers une description VHDL au niveau transfert de registre (RTL).

De son coté, le monde de la vérification manipule des modèles basés sur la sémantique des Algèbres de Processus. Celle–ci est différente de la sémantique VHDL à un point tel qu'il est difficile de transformer directement un modèle VHDL en un modèle décrit en Algèbre CCS. Citons les différences essentielles entre VHDL et CCS.

- Une première différence réside dans la modélisation du temps. Alors que CCS est un langage asynchrone, la sémantique de simulation de VHDL (et non le langage proprement dit) impose un synchronisme.
- En second lieu, alors que la communication VHDL repose sur l'échange de données, les Algèbres de Processus communiquent par événements. Alors que les processus VHDL communiquent au

moyen d'affectations de signaux et d'états de blocage (WAIT ON), en CCS les échanges de message ne se font que par mode rendez–vous : pour qu'une information puisse être émise, il faut que les processus concernés soient à l'écoute de cette information.

Au niveau de la description comportementale des processus, la transformation VHDL vers un formalisme tel que les Machines à Etats Etendues ou les réseaux Prédicats /Transitions ne pose pas de grosses difficultés. Par contre, c'est à propos de la communication que se situe le noeud du problème : les sémantiques de communication VHDL et de l'Algèbre CCS sont très différentes. Afin de faciliter la rencontre de ces mondes, une passerelle a dû être définie : le langage VOVHDL. Celui–ci doit se placer à un niveau plus abstrait que le niveau RTL (Register Transfer Level) afin de se rapprocher des modèles issus de la norme.

III.3.2 – Cahier des charges

A partir d'une spécification décrite en VOVHDL, il doit être possible de définir des modèles utilisables tant dans le domaine de la vérification formelle que dans celui de la conception. Afin de réaliser cet objectif, le surlangage de VHDL que l'on définit doit vérifier les propriétés suivantes :

- offrir de puissantes caractéristiques de communication dédiées à la description de niveau système.
- être compilable selon le standard VHDL afin de pouvoir revenir à une conception basée sur VHDL pour une description de niveau inférieur. Cela suppose de conserver le caractère synchrone du langage.

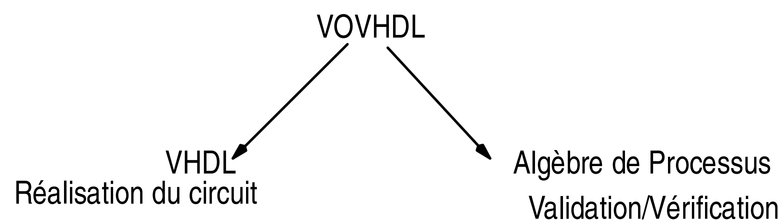


Figure 6 : VOVHDL : une passerelle entre conception et validation

Au–delà de ces deux caractéristiques, l'intérêt du langage VOVHDL réside surtout dans notre capacité à proposer un modèle d'interprétation sémantique de ce langage selon une Algèbre de Processus.

III.3.3 – De VOVHDL vers VHDL

Cette phase de traduction est nécessaire afin de pouvoir tracer la conformité des spécifications VHDL élaborées par les concepteurs par rapport à la spécification VOVHDL de plus haut niveau. Cette conformité repose sur la comparaison des traces de simulation obtenues. De plus, dans une démarche de conception descendante, il est important de pouvoir obtenir de manière automatique le code VHDL qui constitue l'entrée des outils de synthèse. Pour ce faire, une chaîne d'outils a été développée par le Politecnico di Torino. Nous rappelons ci–après les différentes étapes de cette traduction.

La première étape de cette chaîne d'outils consiste à traduire le source VOVHDL en un code VHDL syntaxiquement correct mais non exécutable (fanta_VHDL). C'est ce que réalise un pré–processeur qui vérifie notamment toutes les extensions syntaxiques, les conversions et les restrictions VHDL.

Description VOVHDL d'un canal multi–rendez–vous	Traduction syntaxique
CHANNEL C : **type_out type_in MRV**, ...,...;	**SIGNAL** C : **MRV_ch**;

Où MRV_ch est un enregistrement défini dans une base de données annexe PROTPAC.VHD

Figure 7 : Un exemple de traduction de VOVHDL vers fanta_vhdl

Une deuxième étape permet de relier une base de données qui contient la définition de tous les éléments sémantiques de VOVHDL (en particulier ceux concernant les protocoles de communication). On obtient alors la forme intermédiaire VTIP (arbre syntaxique décoré issu d'un outil industriel) compilable avec les outils VHDL standards.

Par exemple, la procédure Send selon le protocole multi–rendez–vous est traduite par le code VHDL suivant :

```
Procédure MRVSend (  signal rts : out bit;
                     signal data : out integer
                     signal ack : in bit;
                     x : integer) is
begin
        rts <=1;
        data <= x;
        wait on ack;
endMRVSend
```

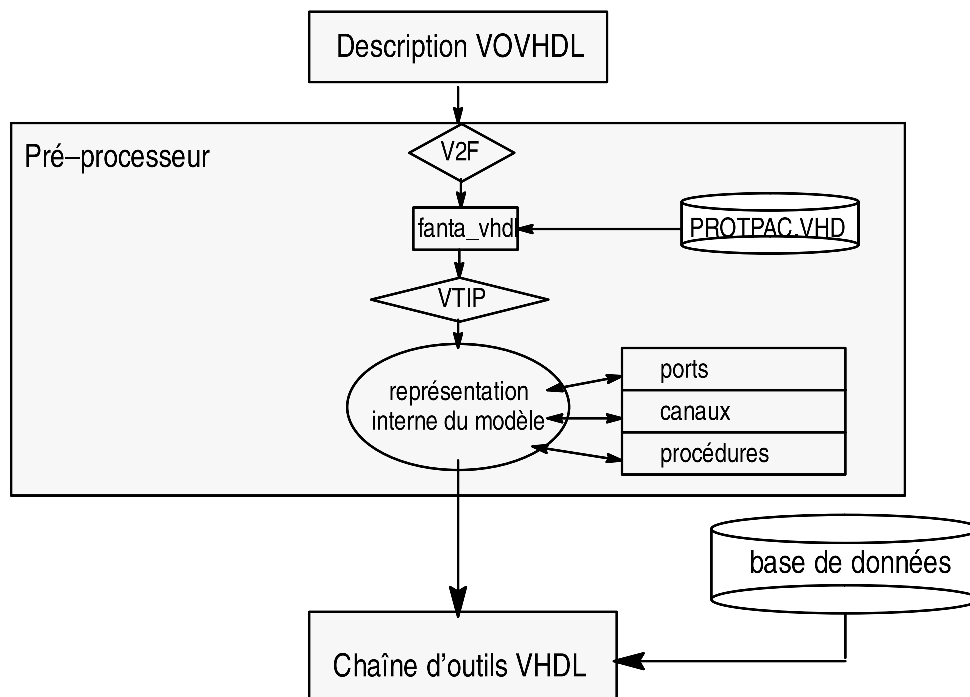


Figure 8 : La chaîne de traduction VOVHDL vers VHDL

Une forme interne du modèle est obtenue sous forme de code VHDL. Celui–ci sert de donnée d'entrée à une chaîne d'outils VHDL utilisée par les concepteurs. Parmi ces outils se trouve le générateur VTIP qui permet d'obtenir le code VHDL exécutable issu du code VOVHDL initial. Par ce processus de traduction, le code VOVHDL initial est ainsi simulable au moyen d'outils VHDL.

III.3.4 – De VOVHDL vers les outils de preuve

Dans le contexte du projet (Figure 4) et compte tenu des choix de vérification retenus (II.2) une description VOVHDL doit être traduite sous forme d'un modèle dont la sémantique est une sémantique opérationnelle des Algèbres de Processus.

Chaque processus VOVHDL est traduit de manière syntaxique en un fichier de code C. Le comportement des processus est donné sous forme de Systèmes à Transitions Etiquetées et le modèle global est obtenu par la composition des STE du modèle global. Le Système à Transitions Etiquetées que l'on peut alors obtenir est utilisé comme entrée des outils de vérification.

Le dépliage de ce STE nous permet d'obtenir une Machine d'Etats Etendue sur laquelle des simulations exhaustives et partielles peuvent être exécutées.

D'autre part, un ensemble d'outils de vérification (comme l'outil SEVERO implanté par le Politecnico di Torino) applique des techniques de bisimulation et de Model Checking symbolique en utilisant des Graphes de Décision Binaires (BDD). Ces derniers permettent de traiter de grands systèmes.

Les étapes de traduction sont les suivantes : à partir d'une description VOVHDL, un modèle ASA+ correspondant est extrait au moyen d'un analyseur syntaxique (V–parser). Une simulation exhaustive du modèle permet d'obtenir une description Machine d'Etats Etendue (MEE) de la spécification initiale VOVHDL. Une simple transcription dans un format adapté fournit alors le fichier d'entrée des outils de vérification.

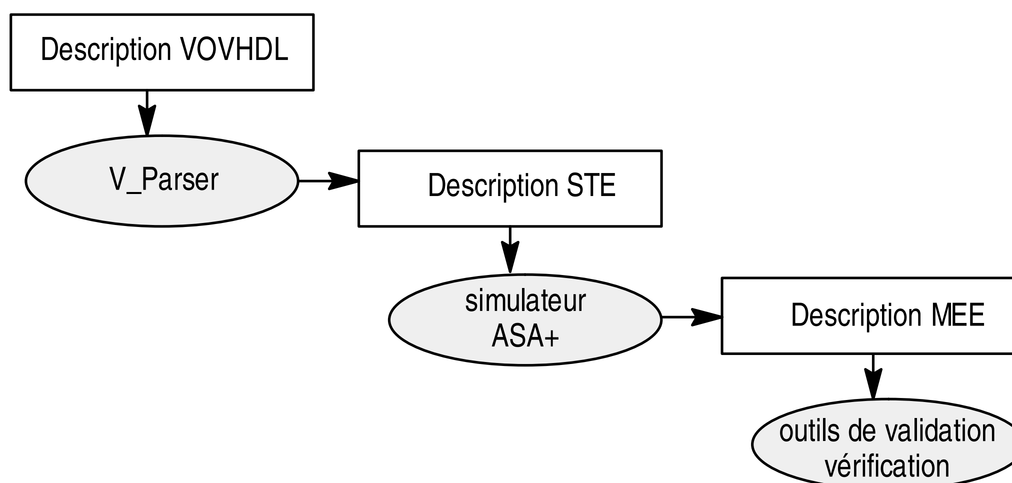


Figure 9 : La chaîne de traduction VOVHDL vers les outils de preuve

IV – LE POSITIONNEMENT PERSONNEL

IV.1 – Depuis une description VOVHDL vers un modèle vérifiable

La contribution de mes travaux à ce projet a consisté :

- ♦ d'une part à définir la sémantique du langage VOVHDL en collaboration avec le Politecnico di Torino ; il s'agit de définir un langage non seulement compatible avec le standard VHDL mais encore possédant une sémantique de communication de haut niveau qui permette d'obtenir un modèle vérifiable,
- ♦ d'autre part à définir un modèle d'interprétation sémantique du langage VOVHDL selon une sémantique Algèbre de Processus afin de valider des propriétés sur ce modèle à l'aide d'un outil de vérification formelle,
- ♦ enfin ce modèle d'interprétation a été utilisé dans le cadre du projet FICOMP et j'ai contribué à l'élaboration de scénarios qui ont été validés sur la partie architecture du composant.

IV.2 – Modélisation de la communication VHDL

Forte de l'expérience acquise sur VOVHDL, en particulier concernant la caractérisation de l'aspect synchrone, je me suis attachée par la suite à définir un modèle d'interprétation sémantique du langage VHDL selon la même méthodologie que celle utilisée pour VOVHDL. Pour cela, j'ai dû mettre en évidence la caractéristique synchrone de la sémantique de communication de VHDL.

L'intérêt de cette démarche est double :

- ♦ d'une part lors d'une méthodologie de conception complètement descendante, il est possible de raffiner une spécification depuis une description VHDL de haut niveau vers une description VHDL au niveau RTL puis vers les outils de synthèse automatique sans pour autant passer par un intermédiaire VOVHDL,
- ♦ d'autre part en vue de la validation, on peut être amené à vérifier à un plus bas niveau seulement une partie de composant décrite directement en VHDL .

Il est donc utile d'étudier la possibilité de construire un modèle vérifiable, directement à partir d'une description VHDL, sans passer par un langage intermédiaire.

VOVHDL ET SON INTERPRETATION SEMANTIQUE SOUS FORME D'UN SYSTEME A TRANSITIONS ETIQUETEES

CHAPITRE 2

Dans un premier temps, nous présentons le langage orienté vérification (VOVHDL) qui permet de faire le lien entre le domaine de la vérification et celui de la conception. Nous nous attacherons surtout à la sémantique de communication du langage qui est basée sur la notion de canaux de communication et qui gère un transfert d'informations au moyen de protocoles bien définis, de règles sur les émetteurs et récepteurs et de primitives de communication.

Après avoir mis en évidence le caractère synchrone de cette sémantique, nous montrons comment, au moyen d'une interprétation Système à Transitions Etiquetées, nous modélisons le comportement d'un système de processus décrits en VOVHDL en un ensemble de modules qui évoluent de manière synchrone et qui communiquent selon une sémantique CCS.

A cet effet, chaque processus est interprété sous forme d'un STE et la communication de l'ensemble est gérée par un unique module qui intègre les définitions des protocoles et les principes de la communication VOVHDL définis en début de chapitre.

Enfin, nous illustrons les propos de ce chapitre sur un petit exemple : depuis sa description VOVHDL jusque vers son implantation selon un modèle STE.

I – LA SÉMANTIQUE DE COMMUNICATION DE VOVHDL

I.1 – Introduction à VOVHDL

Le langage VOVHDL est une variante du langage VHDL. Il a été défini de manière à prendre en compte, d'une part, des restrictions nécessaires pour garantir la vérification d'un modèle selon l'Algèbre des Processus et d'autre part, des extensions du standard de communication pour définir des protocoles de plus haut niveau.

I.1.1 – Un modèle abstrait

Un système peut être décrit comme un ensemble de processus concurrents qui s'échangent des informations au moyen de canaux de communication. Une représentation de cette description peut alors être donnée sous forme de boîtes noires sur lesquelles sont définis des ports de communication. Ces ports de communication sont connectés entre eux au moyen de canaux.

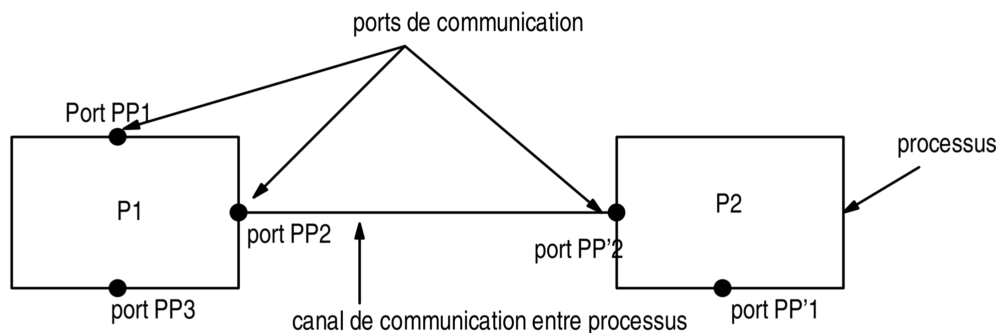


Figure 19 : Modèle abstrait d'une description VOVHDL

C'est par l'intermédiaire de ces canaux que des événements sont communiqués d'un processus vers un autre.

I.1.2 – Une structure hiérarchique

Dans une description structurelle, chaque instance d'un processus est représentée par un composant. Le composant est défini par l'entité et l'architecture associée. L'entité spécifie l'interface associée au processus. L'architecture spécifie le comportement du processus et elle peut être définie à son tour soit par un ensemble de composants qui communiquent au moyen de canaux (on a alors une description structurelle) soit par un unique processus VHDL (description comportementale).

La communication entre composants au même niveau hiérarchique se fait au moyen de canaux VOVHDL (les signaux VHDL plus traditionnels mais moins puissants ne sont pas autorisés). C'est par la définition de sa sémantique de communication que se caractérise VOVHDL.

Le niveau feuille est constitué soit d'une description comportementale sous forme d'un processus VOVHDL soit sous forme de composants instanciés à partir des blocs élémentaires d'une bibliothèque.

Il est important de remarquer que le parallélisme interne n'existe pas en VOVHDL. Si à un moment donné un module communique avec plusieurs canaux, on crée des agents supplémentaires (création du parallélisme de structure).

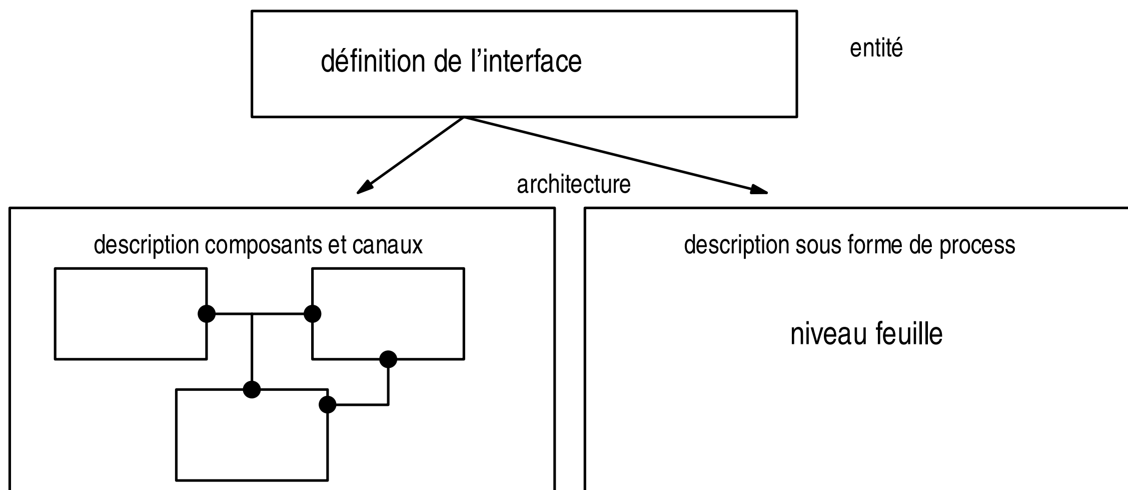


Figure 20 : Une vision abstraite d'une description d'un système

I.2 – Le support de communication

Ce modèle de communication permet d'exprimer la communication entre différents sous–systèmes décrits soit sous forme de modules VOVHDL soit sous forme de modules issus de la modélisation CCS. Nous précisons de manière formelle les principes de communication de VOVHDL.

I.2.1 – Un processus

Un processus est formellement défini par le triplet :

< PN, PC, PPL>

où PN (Processus_Nom) est le nom du processus, PC (Processus_Comportement) la description comportementale du processus donnée sous forme de corps de programme, et enfin PPL (Processus_Port_Liste) la liste des ports du processus. La description comportementale du processus peut être donnée soit sous forme d'une description fonctionnelle selon des techniques descriptives [KOO, 91], soit sous forme de composition structurelle de processus concurrents qui échangent des données et des synchronisations sur des canaux de communication.

Ainsi, nous définissons de manière formelle le processus P1 de la Figure 19 par :

< P1, addition, (PP1, PP2, PP3)>,

où addition est un nom correspondant à la partie comportementale du corps du processus de nom P1 et qui effectue par exemple l'addition des deux valeurs courantes des ports PP1 et PP2 et renvoit le résultat sur PP3.

I.2.2 – Un port

C'est par l'intermédiaire des ports que les informations s'échangent d'un processus à l'autre. Un port est défini par le quadruplet :

< PPN, PPS, PMT, PCP>

dont chacun des éléments correspond respectivement au nom du port (Port_Processus_Nom), à son sens (c'est à dire s'il est défini en entrée ou en sortie : Port_Processus_Sens), au type des messages qui transitent par ce port (Port_Message_Type), au protocole de communication (Process_Communication_Protocole) qui le régit.

Ainsi le port PP1 du Processus P1 de la Figure 19 sera défini :

< PP1, Input, Integer, MRV>

C 'est un port qui prend en entrée des entiers et le protocole de communication du canal sur lequel il se trouve est le Multi–Rendez–Vous.

I.2.3 – Un canal

Entre deux processus, l'information est transmise par l'intermédiaire des ports mais ce sont les canaux qui gèrent la circulation de l'information d'un processus vers un autre. Un canal gère une communication multi–points et a besoin de connaître un certain nombre d'informations du genre : quels émetteurs et récepteurs sont concernés par la communication? Un canal est donc défini par les six éléments suivants :

< CN, CT, CPr, L_Em, CR, L_Re, DR>

- son nom (CN Canal_Nom)) ;
- son type (CT Canal_Type), c'est à dire le type des messages qui peuvent être gérés par le canal
- son protocole (CPr Canal_Protocole) : cela peut être MULTI_RENDEZ_VOUS (MRV), CALL&RETURN (CALL), VSIG_value, VSIG_event (où VSIG signifie "VHDL signal").
- la liste des émetteurs actifs du canal (**L_Em**) ;
- La règle que doivent vérifier ces émetteurs (**règles de composition CR**) ; cette règle peut être de trois types :

  – **ALL** : tous les émetteurs du canal doivent être prêts et transmettre la même valeur à ce pas de simulation, dans le cas contraire, les émetteurs qui sont prêts sont bloqués en attendant que tous soient prêts et émettent la même valeur. Si la valeur émise n'est pas la même, la communication est bloquée et impossible : c'est un cas de dead–lock.

  – **ONE** : dès qu'un émetteur du canal est prêt, il peut émettre la valeur sur le canal, c'est une exclusion mutuelle. Dans le cas où plusieurs émetteurs sont prêts, un parmi ceux–là est choisi de manière non détérministe et les autres sont bloqués.

  – **SOME** : de façon non déterministe, un sous–ensemble des émetteurs du canal (ensemble d'émetteurs prêts à transmettre la même valeur) est défini et c'est cet ensemble qui réalise la communication. Si il n'y a pas au moins un émetteur présent, il n y a pas de communication possible.
- La liste des récepteurs actifs sur le canal (**L_Re**) ;
- La règle que doivent vérifier ces récepteurs (**règles de Répartition DR**) ; cela peut être :

  – **ALL** : tous les récepteurs doivent être prêts pour recevoir le message du canal, sinon ceux qui sont prêts sont bloqués en attendant que tous soient présents. Au cas où aucun émetteur n'est présent la communication ne peut pas avoir lieu.

  – **ONE** : De manière non–déterministe seulement un récepteur parmi ceux qui sont prêts reçoit le message, les autres ne sont pas concernés par la communication. Si aucun émetteur n'est présent, il n y a pas de communication.

  – **SOME** : tous les récepteurs qui sont prêts reçoivent la valeur du canal. Les autres ne sont pas concernés.

I.3 – Une sémantique de communication dirigée par des protocoles

I.3.1 – Les protocoles de communication

Les protocoles jouent un rôle clé dans le modèle de communication VOVHDL. Nous donnons une brève description de la sémantique de chacun d'entre eux. Cependant chacun d'eux peut être caractérisé de manière formelle suivant le modèle :

< TBC, #, RBC>

où TBC définit la condition de blocage sur les émetteurs (deux valeurs sont possibles, B pour bloquant et NB pour non bloquant) , # la dimension de la ressource que constitue le canal, RCB la condition de blocage sur les récepteurs (B ou NB).

Les quatre types de protocoles définis l'ont été en fonction des besoins des concepteurs au niveau système. Ils représentent des types de communication relativement usuels pour les concepteurs de cartes de communication.

MRV : La communication peut être établie quand tous les émetteurs (selon la règle de composition CR) sont prêts à émettre et tous les récepteurs (selon la règle de répartition DR) sont prêts à recevoir. La communication est réalisée simultanément par tous les processus. Si l'une des conditions, soit sur les émetteurs, soit sur les récepteurs n'est pas vérifiée alors les processus qui sont prêts sont bloqués en attendant les autres. Un tel protocole est donc bloquant sur les émetteurs et les récepteurs. Lorsque la communication se réalise, elle a lieu dans un pas de simulation et donc le canal n'a pas besoin de disposer de ressource. Un tel protocole se définit formellement par <B,0,B>. Ce protocole est particulièrement utilisé pour la description à un haut niveau de synchronisation de structures indépendamment de leur implantation. Les règles de composition et de répartition all, one, et some sont autorisées pour ce protocole.

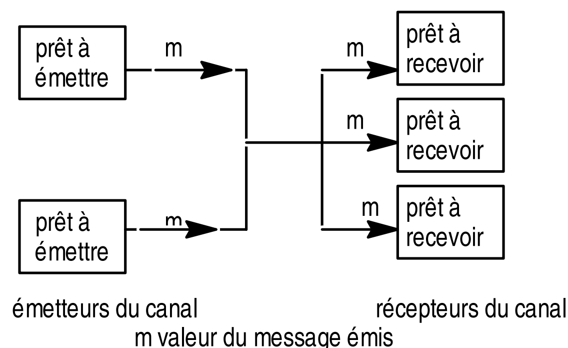


Figure 21 : Réalisation possible d'une communication selon le protocole MRV

CALL : L'émetteur envoie un message et réveille le récepteur qui attendait un message. C'est seulement quand le récepteur acquitte la fin de sa tâche que l'émetteur reprend son activité. Le protocole call&return est facilement interprété en terme de deux communications MRV <B,0,B>: l'une associée à l'appel, et l'autre correspondant à l'acquittement (Figure 22). Lorsque deux processus écrivent sur un même canal, la requête de communication est maintenue jusqu'à ce que le précédent appel ait eu un retour. Ce protocole est utile pour les appels multiples et concurrents d'une procédure partagée dont le litige est arbitré par le canal. Il est en particulier utile pour les appels du micro–code de la partie Software depuis la partie Hardware.

Dans ce cas de protocole, les trois règles CR sur les émetteurs sont possibles (ALL, ONE, SOME). Par contre, deux règles de répartitions DR sont retenues : ALL, tous les récepteurs sont prêts et sont activés par l'appel (cela permet de modéliser le déclenchement simultané de plusieurs procédures concurrentes), et ONE, un seul récepteur est actif (cela modélise le cas où un serveur est libre parmi un ensemble de serveurs).

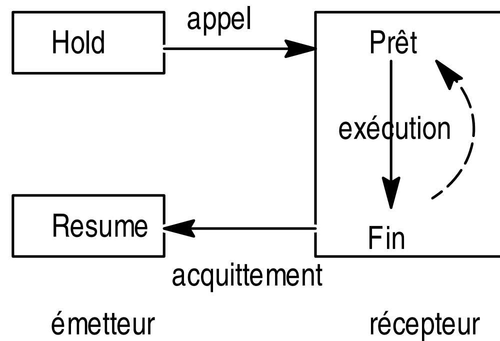


Figure 22 : Le mécanisme du protocole CALL

VSIG : Les émetteurs transmettent (suivant la règle CR) un message sur le canal qui peut être vu comme un tampon contenant un message conservant la dernière valeur écrite. Les deux modes de réception suivants peuvent être alors définis :

- réception dirigée par la valeur (protocole VSIG_value) : le récepteur lit à tout moment le dernier message écrit (équivalent à un signal de référence, où les récepteurs ont une valeur tampon (poll latch value)). Un tel protocole n’est ni bloquant sur les émetteurs, ni sur les récepteurs. Il nécessite une ressource : un tampon de dimension 1 qui contient la dernière valeur du canal. Le modèle formel d’un tel protocole sera <NB, 1, NB>.

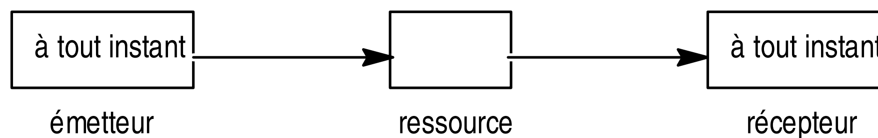


Figure 23 : Le protocole Vsig–val

- réception dirigée par événement (protocole VSIG_event) : le récepteur attend jusqu’à ce qu’une nouvelle valeur soit écrite puis se réactive avec la nouvelle valeur lue. Cela correspond à la condition du WAIT ON du langage VHDL. Il est utilisé pour modéliser des systèmes dont l’évolution est faite par interruption (interrupt_driven systems) et des récepteurs sur front montant. Dans le cas où plusieurs récepteurs sont concurrents, le résultat de la valeur du canal est non déterministe. C’est un protocole de représentation formelle <NB, 1, B> : la dimension de la ressource est 1 et il peut y avoir blocage sur la réception.

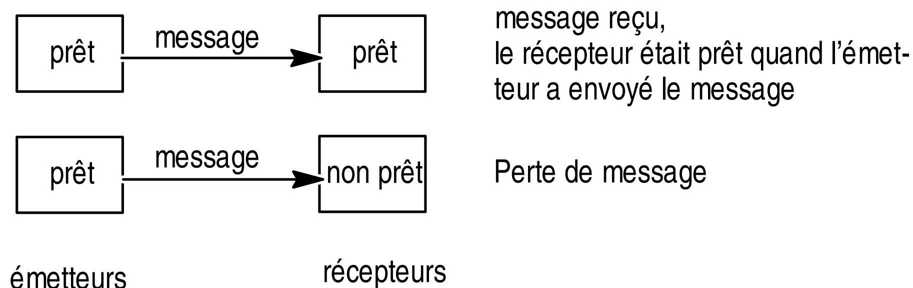


Figure 24 : Protocole VSIG–event

I.3.2 – Les primitives de communication SEND et RECEIVE

Un transfert d'information entre deux modules est explicitement déclaré dans le corps du programme VOVHDL au moyen de deux primitives de communication que sont les procédures SEND et RECEIVE .

- La procédure SEND (ch1, val_out) dans le corps de programme d'un module signifie que le module est prêt à émettre sur le canal 'ch1' la valeur 'val_out'.
- La procédure RECEIVE (ch1, val_in) dans le corps de programme d'un processus signifie que le processus est prêt à recevoir sur le canal 'ch1' la valeur de la variable 'val_in'.

Ainsi, dans une spécification VOVHDL, tous les événements de communication correspondent à l'appel soit d'une procédure SEND, soit d'une procédure RECEIVE.

Il est possible de considérer une émission multiple (Multi_SEND) ou une réception multiple (Multi_RECEIVE), mais dans ce cas l'interprétation de la communication doit être atomique.

Par exemple, dans le cas où il y a une émission multiple des trois processus P1, P2, P3 sur un canal ch1 et si la règle CR est ALL , si cette règle est vérifiée et si la valeur émise est val_em alors la procédure multi_send sera équivalente à l'émission d'un seul émetteur envoyant la valeur val_em.

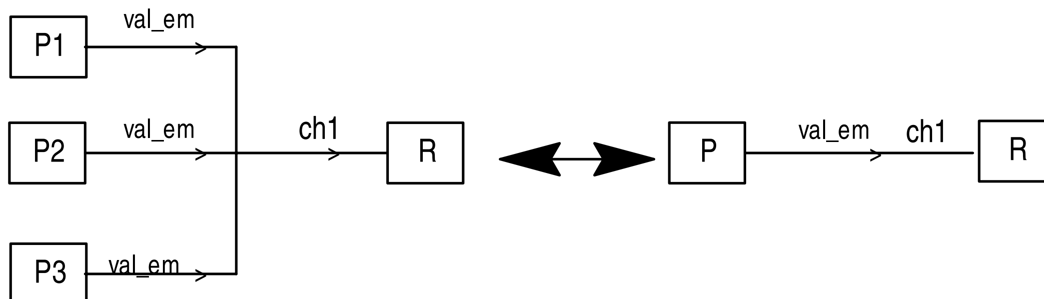


Figure 25 : vision atomique d'une émission multiple

I.3.3 – Les transferts d'informations

Les transferts d'informations entre émetteurs et récepteurs ne sont réalisés que si les propriétés du canal sont vérifiées ; c'est à dire si les règles de composition, de répartition et de protocole sont vérifiées. Par exemple, si un canal est caractérisé par le protocole MRV, la règle de composition ALL et la règle de répartition ONE, alors la communication est possible si et seulement si au même pas de simulation, tous les émetteurs du canal sont prêts à émettre la même valeur et au moins un récepteur parmi ceux du canal est prêt à recevoir. Dans le cas contraire, les modules, que ce soit en réception ou en émission, restent bloqués jusqu'à la satisfaction de toutes les règles.

Etant donné que dans le cas du protocole VSIG_val notre modèle était restreint à des canaux de communication avec un seul émetteur, nous pouvons noter que ce protocole ne conduisait jamais à une condition de blocage ni du coté émetteur, ni du coté récepteur.

I.4 – Une sémantique de communication synchrone

Afin de pouvoir garantir une description conforme à celle des concepteurs, il est nécessaire de définir un langage dont la sémantique est synchrone. En effet, les concepteurs utilisent des outils de synthèse et il est important de pouvoir obtenir des descriptions comparables en termes de pas de simulation.

De manière à interpréter une spécification VOVHDL suivant une vue de simulation telle qu'en VHDL, la sémantique de communication de VOVHDL hérite des caractéristiques synchrones de VHDL. Pour cela, nous définissons un pas de simulation δ_{VOVHDL} comme l'intervalle de temps qui sépare deux événements de communication.

Tel le fonctionnement d'un échantillonneur bloqueur, le déroulement d'un programme VOVHDL peut être décomposé en trois étapes de simulation :

- Au début du pas de simulation, les valeurs d'entrées de tous les canaux sont échantillonnées et bloquées.
- Puis chaque processus non bloqué reprend son activité jusqu'au nouvel événement de communication qui peut être un état **SEND** ou un état **RECEIVE**. Cette activité correspond aux traitements effectués par le processus entre deux évènements de communication, afin de faire évoluer sur un état interne les valeurs de sorties en fonction des valeurs d'entrées.
- Puis, quand tous les processus sont stabilisés, la possibilité du transfert d'information est évaluée suivant les règles de composition, de répartition et du protocole. Dans le cas favorable où la communication est possible, la valeur contenue par le canal est propagée vers les récepteurs pour le prochain pas de simulation.

En cas de communication impossible, les modules concernés sont bloqués jusqu'à la troisième étape du pas de simulation suivant de façon à évaluer de nouveau les règles du canal et ainsi de suite jusqu'à leur satisfaction. De cette manière, un processus peut être dans un état d'attente durant plusieurs pas de simulation δ_{VOVHDL}.

II – LES PRINCIPES DE MODELISATION

II.1 – Les moyens de modélisation

II.1.1 – Qu'est ce qu'un STE?

Un Système à Transitions Etiquetées est un graphe dont les noeuds correspondent aux états du système et les arêtes correspondent aux transitions. Toute arête est étiquetée par le nom de l'action arrivant en connexion avec la transition [VT,91].

Un système à transitions étiquetées est un quadruplet [DR,92] : STE = (S, Σ, Δ, s_0) où

- S est un ensemble dénombrable d'états,
- Σ est un ensemble d'actions observables,
- $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$, est l'ensemble des transitions qui permettent de passer d'un état vers un autre au moyen d'une action. Cette action est soit un élément de l'ensemble des actions Σ, soit une action interne notée τ n'appartenant pas à Σ,
- s_0 élément de S est l'état initial du système à transitions étiquetées.

Un système à transitions étiquetées est généralement vu comme un ensemble de processus (ou d'agents selon la terminologie des Algèbres de Processus) qui exécutent des actions selon des règles de transitions spécifiées par Δ.

II.1.2 – La conservation des traces

De manière à préserver le double objectif : d'une part, de conception basée sur VHDL, et d'autre part, de vérification basée sur CCS, nous devons considérer deux traductions. D'un côté, la traduction d'une spécification VOVHDL en un script VHDL, d'un autre côté, son interprétation selon la sémantique CCS. Afin de garantir la validité de la vérification elle–même, ces traductions doivent préserver la conformité entre la spécification VHDL et le modèle CCS. Le critère de conformité minimal est l'équivalence de trace vis à vis des événements de communication de VOVHDL (qu'ils soient externes ou internes). Ils doivent donc être observables en VHDL et en CCS (Figure 26).

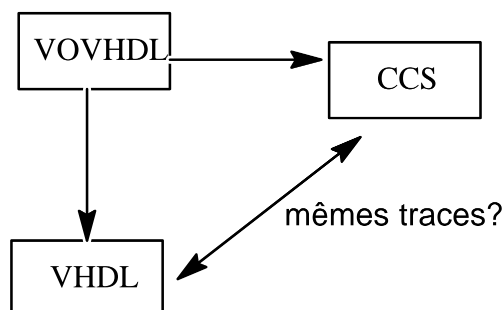


Figure 26 : Relations entre les formalismes

VOVHDL et VHDL ont tous les deux une sémantique d'exécution synchrone. Les caractéristiques de VHDL facilitent la traduction : en effet, on peut décomposer chaque pas de simulation VOVHDL en plusieurs pas de simulation VHDL. Cela introduit alors des transitions internes. C'est la raison pour laquelle les protocoles VOVHDL sont modélisés en VHDL de manière à respecter chaque pas de simulation VOVHDL. Les traces des événements VOVHDL sont ainsi préservées lors de la traduction VHDL.

Afin d'obtenir le même résultat en ce qui concerne la traduction CCS, deux aspects sont à considérer :

- l'interprétation de chaque processus VOVHDL en termes d'un modèle STE, en tenant compte des événements de communication
- l'interprétation de la sémantique de communication VOVHDL en termes d'un module comportemental qui lui même est décrit sous forme d'un STE.

Ainsi, deux processus VOVHDL P1 et P2 composés selon la sémantique de communication VOVHDL sont interprétés selon la sémantique CCS (Figure 27) par la composition parallèle de :

- la représentation STE de chaque processus
- la représentation STE du module comportemental de communication B_{VOVHDL_COM}

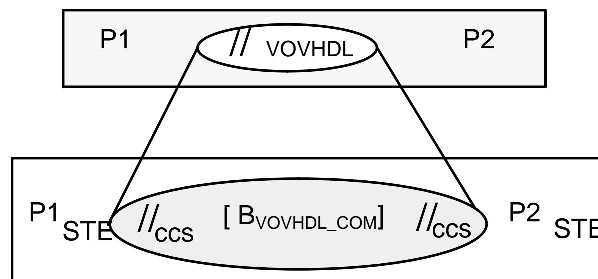


Figure 27 : Composition parallèle de processus VOVHDL en CCS

Chacun des trois processus $P1_{STE}$, $P2_{STE}$ et B_{VOVHDL_COM} communique alors selon le mode rendez_vous de la sémantique CCS.

II.2 – Interprétation STE de chaque processus

II.2.1 – Interprétation STE d'un processus VOVHDL

Si nous considérons un processus VOVHDL comme une boîte noire avec des ports d'entrées et sorties, nous pouvons observer ses événements de communication au cours de la simulation de la spécification VOVHDL. En théorie, il est possible d'interpréter les traces obtenues en termes de Système à Transitions Etiquetées.

Un tel STE associé à un processus VOVHDL est caractérisé par le fait que :

- chaque état correspond au moment immédiatement avant un événement de communication,
- chaque transition, étiquetée par l'événement de communication qui suit l'état courant correspond à un pas de simulation VOVHDL.

En outre, comme nous l'avons vu lors de la définition des protocoles VOVHDL, certains d'entre eux peuvent conduire à des situations de blocage des processus. Depuis un état donné (état_donné), nous nous trouvons donc devant deux situations :

- soit la communication a été possible et le processus a évolué : on a alors atteint un nouvel état (état_suivant) ;
- soit il y a eu blocage et on reste dans le même état : la transition qui est alors tirée est appelée transition boucle (Figure 28).

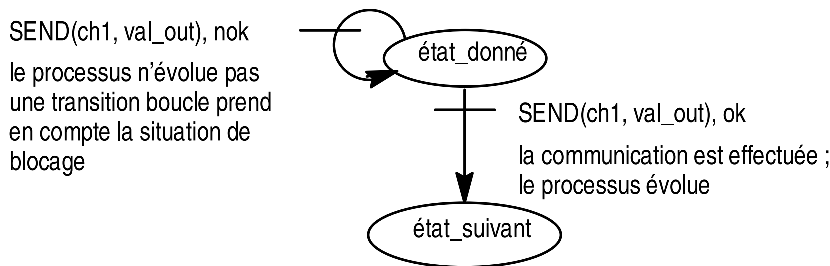


Figure 28 : D'un état STE vers le suivant

L'extension de l'étiquette de la transition par le prédicat ok ou nok permet d'identifier une situation de blocage (nok) ou non (ok).

Un processus VOVHDL pourrait être traduit en un STE constitué de plusieurs éléments tels que ceux décrits dans la Figure 28 en considérant que chaque état_donné d'un élément est un état_suivant de l'élément suivant.

Cependant, il nous a paru intéressant de distinguer d'une part le comportement d'un processus vis à vis de son environnement et d'autre part son évolution interne. Cette distinction permet de structurer notre modélisation et de maîtriser la complexité du modèle.

II.2.2 – La partie générique et la partie Particularisée d'un processus

Un processus VOVHDL peut en effet être décomposé en deux parties. D'une part, il y a le comportement du processus vis à vis de son environnement : le fait qu'il soit bloqué ou non et dans le cas où il communique, l'identification du flot de données. D'autre part, dans le cas d'une communication, le processus évolue de manière interne suite à la réception ou à l'émission d'une information. La partie Générique (il est à noter que le terme générique est inspiré des outils de modélisation et utilisé ici dans le même sens) correspond à la modélisation de l'évolution du processus par rapport à son environnement alors que la partie particularisée représente l'évolution interne du processus.

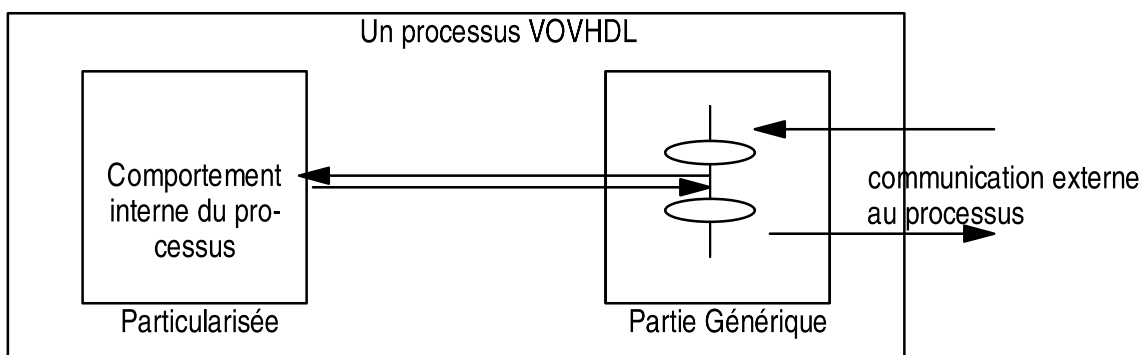


Figure 29 : La décomposition d'un processus

La partie Générique

La partie Générique est en fait la même pour tous les processus. Elle représente l'évolution d'un processus depuis l'instant précédant immédiatement un événement de communication (en fait une instruction SEND ou RECEIVE) jusqu'à l'événement de communication suivant. Le déroulement de l'exécution est le suivant :

- l'état du processus est le moment immédiatement avant un événement de communication ; il reste dans cet état jusqu'au moment où son environnement lui renvoie le résultat de la communication.

- Soit il y a eu blocage et dans ce cas le processus ne peut pas évoluer ; il reste dans le même état et reémet le même message de communication.
- Soit la communication permet au processus d’évoluer. Il y a alors appel de la partie particularisée : en fonction des résultats de la communication, le processus évolue et atteint une nouvelle primitive de communication. L’état des variables internes du processus a également changé.

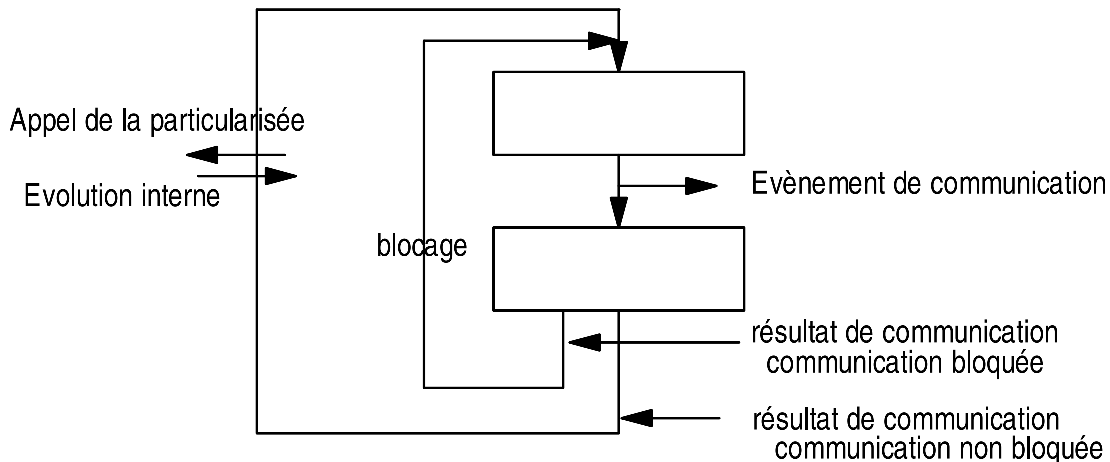


Figure 30 : Comportement de la partie générique d’un processus VOVHDL

La partie Particularisée

Le comportement interne de chaque processus peut être interprété sous forme d’un Système à Transitions Etiquetées. La machine d’état du système est en fait le graphe des états atteignables, mais dans la pratique, l’extraction d’un STE à partir d’une spécification VOVHDL se heurte au problème du développement de structures avec branchements conditionnels. Le calcul de toutes les transitions possibles d’un état vers un autre est rendu très difficile. Aussi, la machine d’états finis associée au processus est déterminée de manière dynamique. Chaque fois que la partie particularisée est appelée, le nouvel état du processus est déterminé en fonction des variables courantes du système (Figure 31).

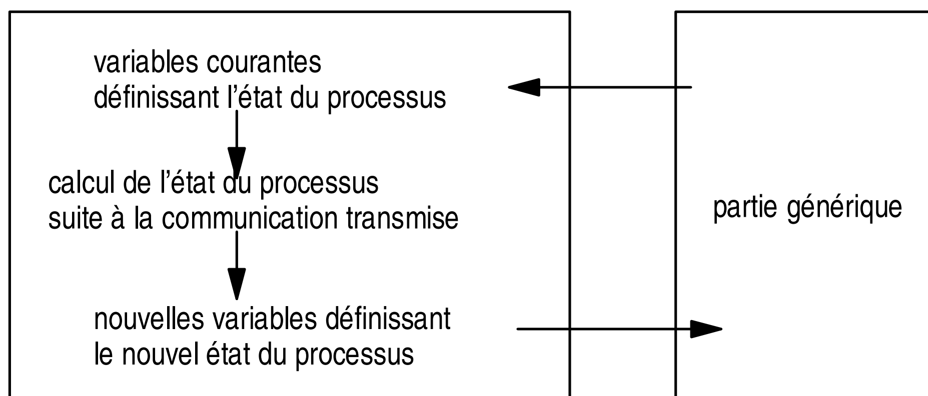


Figure 31 : Le rôle de la partie Particularisée

II.2.3 – Le modèle d'un processus VOVHDL sous forme de STE

Cette décomposition d'un processus en termes de partie Générique et partie Particularisée, nous permet de représenter un processus VOVHDL sous forme d'un STE bien particulier :

♦ il n'est plus constitué que d'un seul état : l'état en attente d'un événement de communication;

♦ depuis cet état seulement deux transitions sont tirables : une transition boucle (situation de blocage) et une transition principale (évolution interne du processus).

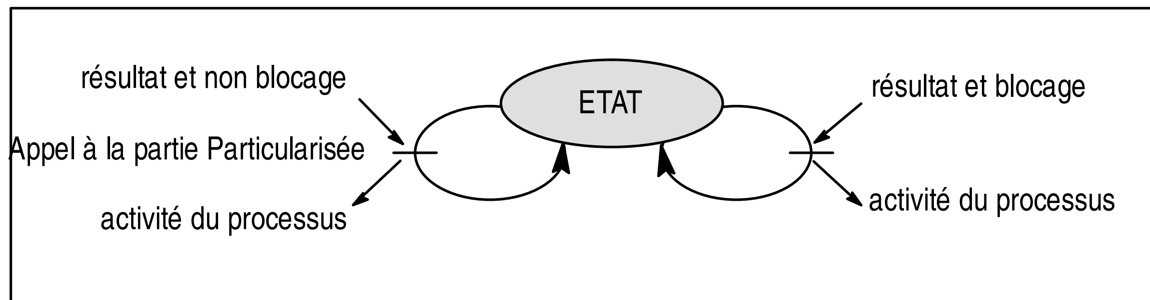


Figure 32 : Un seul état et deux transitions pour un processus

II.2.4 – Une interprétation synchrone : une transition atomique

Si nous considérons le modèle STE d'un processus en dehors du système complet, son comportement est non déterministe (l'une ou l'autre des deux transitions étiquetées est tirable). Cependant, la modélisation de la sémantique de communication (II.3) et la composition parallèle de tous les processus garantit qu'à chaque pas de simulation, seulement une des deux transitions étiquetées soit tirable depuis un état donné.

Nous considérons qu'un système VOVHDL évolue de manière synchrone si tous les processus évoluent vers un nouvel état de communication (ou éventuellement restent dans le même état pour les processus bloqués) en un seul pas de simulation.

La sémantique synchrone de VOVHDL force la synchronisation implicite des processus : tous les modules STE évoluent de manière synchrone :

♦ depuis leur état initial au même instant de départ,

♦ en tirant en même temps (dans un même pas de simulation) une transition.

L'interprétation synchrone de la sémantique d'exécution est garantie par le fait qu'à chaque pas de simulation une seule transition est tirable.

II.3 – Interprétation de la communication

II.3.1 – Une propriété fondamentale

En considérant l'interprétation STE du comportement d'un processus VOVHDL, nous devons mettre en évidence une propriété très simple mais néanmoins fondamentale :

A tout instant (c'est à dire à chaque pas de simulation), chaque processus est actif sur au moins un canal

Cette propriété reste encore vraie dans le cas d'une émission ou d'une réception multiple. Il est possible de tenir compte d'une multiple activité d'un processus sur plusieurs canaux durant un pas de simulation. Dans ce cas, nous interprètons l'échange multiple comme un événement de communication unique et atomique (cf I.3.2).

Au cours de l'évolution synchrone de tous les processus, le modèle de communication doit représenter leur composition parallèle en terme de rendez_vous (réussi ou non) des différentes activités sur chaque canal selon les règles de composition, de répartition et de protocole.

Une première approche nous conduirait naturellement à modéliser chaque canal par un module relié à ses émetteurs et à ses récepteurs. Mais l'analyse de cette solution, comme nous le montrons ci–après, nous a conduit à envisager une autre architecture.

II.3.2 – Un module de communication pour chaque canal

Le principe

Chaque canal de la spécification VOVHDL est remplacé par un module de communication qui réfléchit le comportement du protocole spécifié sur le canal (y compris les règles de composition et de répartition).

Ce module de communication est alors lié à ses émetteurs et récepteurs selon les règles de la communication CCS : le mécanisme du Rendez_Vous. Un canal VOVHDL est donc traduit par deux fils de communication rendez_vous et un module comportemental qui contient la définition des protocoles et des primitives de communication VOVHDL. Le module comportemental peut donc, à lui tout seul, gérer la communication du canal qu'il modélise.

Un exemple

Considérons le petit exemple suivant (Figure 33) où trois processus VOVHDL communiquent entre eux au moyen de quatre canaux VOVHDL. Chacun de ces canaux est défini en VOVHDL par la liste de ses émetteurs et de ses récepteurs ainsi que par le protocole et les règles de composition et de répartition qui régissent sa communication

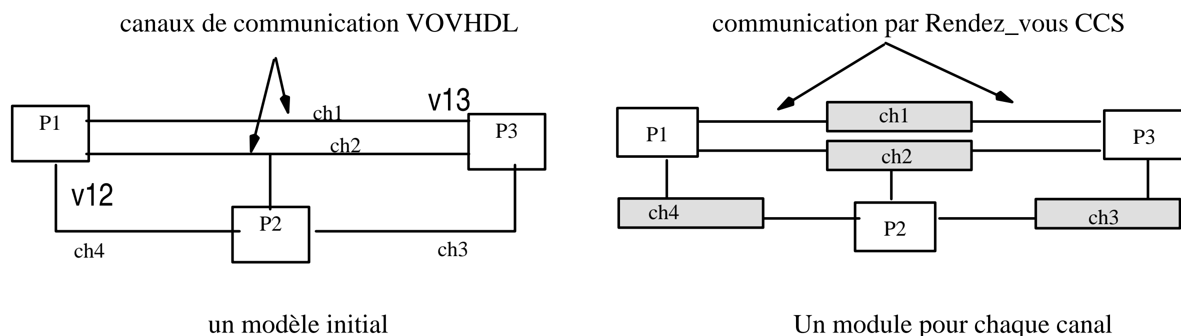


Figure 33 : Un module de communication pour chaque canal

Modélisation de la communication du système

Cette architecture peut être interprétée en termes de composition parallèle selon la sémantique de communication CCS. Chaque canal de communication engendre une règle de composition parallèle entre les modules impliqués par la communication.

- ♦ P1 || P3 || ch1 ; ensemble d'étiquettes [Send(ch1,_), Receive(ch1,_)]
- ♦ P1 || P2 || P3 || ch2 ; ensemble d'étiquettes [Send(ch2,_), Receive(ch2,_)]
- ♦ P2 || P3 || ch3 ; ensemble d'étiquettes [Send(ch3,_), Receive(ch3,_)]
- ♦ P1 || P2 || ch4 ; ensemble d'étiquettes [Send(ch4,_), Receive(ch4,_)]

L'écriture P1 || P3 || ch1 signifie que :

- ♦ chacun des modules P1, P3 et ch1 est modélisé sous forme d'un STE comme décrit précédemment,

- au moins une transition pour chacun des modules est tirable. Ces transitions pouvant porter les étiquettes de l'ensemble suivant : [Send(ch1,_), Receive(ch1,_)] ; (on peut recevoir ou émettre une valeur sur le canal ch1).

A chaque étape, un module de communication est en rendez_vous avec un sous ensemble des émetteurs et récepteurs de tout le système. Suivant les règles des protocoles, c'est lui qui doit faire en sorte que les processus évoluent en tirant soit leur transition principale, soit leur transition boucle. Ce module canal est en rendez_vous avec un processus actif par l'intermédiaire de l'une ou l'autre de ces deux transitions selon la nécessité de le bloquer dans le même état ou non.

Deux points de discussion

Cependant, parce qu'il n'y a, à chaque pas de simulation, qu'un sous ensemble de processus actifs sur un canal donné, le problème suivant est apparu : comment détecter le non rendez_vous d'un processus sur un canal. Par exemple, si P3 est actif sur ch3 et n'est pas actif sur ch1 au même pas, comment faire en sorte que le canal ch1 soit informé du fait que le processus P3 n'est pas en communication sur ce canal mais sur ch3 à un pas de simulation donné? Pour cela, afin d'éviter tout deadlock, le modèle comportemental du canal doit contenir autant de transitions que le nombre possible de sous–ensembles d'émetteurs et de récepteurs.

Un autre point peut être mis en évidence : le module de communication devient de cette manière non déterministe par le fait que deux transitions relatives à deux sous ensembles, l'un incluant l'autre, sont concurrentes. Par exemple, pour le canal ch2, il est nécessaire d'avoir une transition pour le cas où P1 et P2 sont actifs et un dans le cas où P1 seulement est actif. Mais alors, cette dernière transition est aussi tirable pour le cas où P1 et P2 sont actifs en même temps.
Pour garantir qu'une seule transition correspondant exactement au sous ensemble des processus actifs soit tirée, il est nécessaire de l'étiqueter par la non activité des autres processus. Cela implique que tous les processus sont liés à la boite de communication du canal. Ainsi, chaque processus doit répercuter un événement de communication sur tous les modules de communication des canaux, et informer ainsi un canal de son inactivité par le fait qu'il est actif sur un autre.

II.3.3 – Vers un module de communication UNIQUE

Une même vision pour tous les canaux

Notre étude nous amène à considérer une nouvelle architecture telle que celle décrite par la Figure 34 :

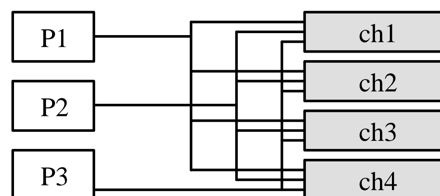


Figure 34 : Intercommunication entre processus et canaux

Ainsi, un pas de simulation est représenté par la composition parallèle qui correspond à la synchronisation d'une transition dans chaque processus et dans chaque module de communication. Par exemple, si P1 et P2 sont actifs sur le canal ch4 et P3 sur le canal ch3, l'évolution peut être représentée par la synchronisation d'une transition dans chaque processus et la transition correspondante dans chaque module de communication. Cette dernière transition est étiquetée par la composition des trois événements de communication.

En fait, nous remarquons qu'il s'agit de la même transition dans chaque module de communication. La seule différence réside dans le fait que le calcul des règles du protocole est distribué sur chaque module de communication qui supporte les caractéristiques de chaque canal (Figure 35).

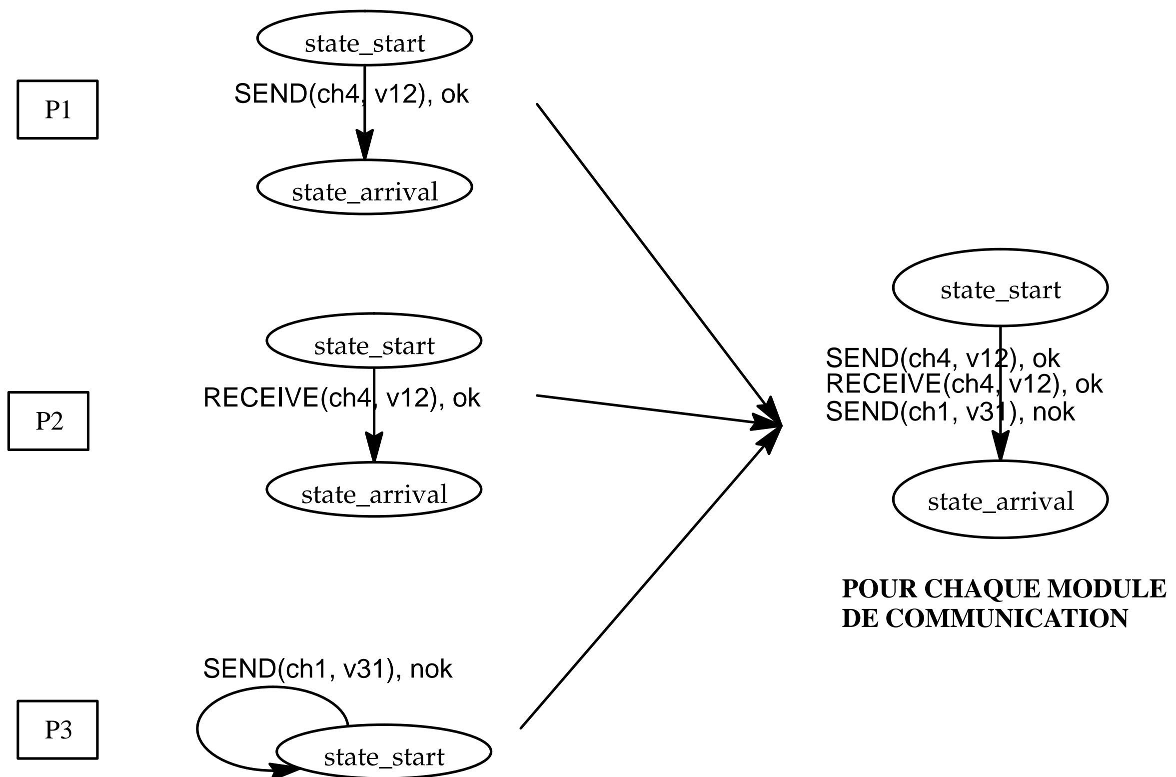


Figure 35 : Un pas de simulation

Chaque module comportemental de chaque canal doit contenir non seulement son activité au pas de simulation courant, mais encore l'activité de tous les autres canaux. Ainsi, tous les modules de communication sont identiques.

Un unique module pour l'ensemble des canaux

Nous définissons donc un nouveau modèle de communication constitué d'un UNIQUE module de communication connecté à tous les canaux (Figure 36).

Tous les processus sont reliés au module de communication et l'informe sur quel canal ils sont actifs.

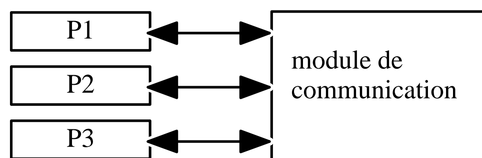


Figure 36 : Un unique module de communication

Ce module contient la définition de tous les canaux, c'est à dire les règles de composition, de répartition et de protocoles de chacun d'eux. Formellement, en terme de modèle STE nous devons considérer les points suivants :

- **l'aspect synchrone de VOVHDL, et la définition d'un pas de simulation δ_{VOVHDL}**
- **le modèle comportemental des différents protocoles autorisés par le langage VOVHDL.**

Le premier point est résolu par notre modélisation du module de communication avec seulement un état et une seule transition. En outre, ce choix garantit que le modèle STE n'introduit pas de transitions internes supplémentaires.

On définit un pas de simulation VOVHDL comme étant ce qui sépare deux événements de communication.

Le second point peut être représenté pour chaque protocole par un ensemble de transitions bouclant sur un unique état. Chacune d'elles représente chaque situation qui peut être rencontrée concernant l'activité ou non des processus et correspondant aux conditions de blocages.

Chaque fois que l'unique transition du module de communication est tirée un pas de simulation VOVHDL s'est écoulé. Chaque processus VOVHDL a évolué d'un ou de zéro événement de communication.

La modélisation STE du module de communication

Le module de communication doit recevoir en un pas de simulation VOVHDL l'activité de tous les processus. Lorsqu'il connaît ces activités sur un pas de simulation donné, il gère la communication sur tous les canaux et communique à chaque processus le résultat de la communication :

- ♦ soit une communication a été effectuée et une valeur a été transmise ou reçue,
- ♦ soit la communication est bloquée et le processus concerné reçoit un ordre de blocage.

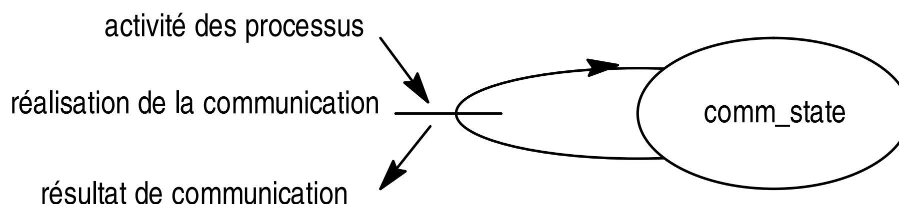


Figure 37 : Représentation STE de l'UNIQUE module de communication

Le rôle de la boite de communication est :

- ♦ de récupérer de manière synchrone l'activité de tous les processus : c'est à dire sur quel canal ils sont actifs et dans le cas où il s'agit d'un émetteur quelle est la valeur qu'il veut transmettre.
- ♦ pour chaque canal de communication, de s'assurer que les règles de composition CR, les règles de répartition DR et le protocole sont vérifiés.
- ♦ de mettre à jour la ressource de communication de chaque canal (par exemple un tampon dans le cas des protocoles de type VSIG)
- ♦ de transmettre à chaque processus, soit le déblocage avec la bonne valeur (la valeur par défaut 'nil' si c'est un émetteur, la valeur reçue si c'est un récepteur), soit le blocage si les règles CR ou DR ne sont pas vérifiées.

II.3.4 – Illustration sur un exemple

Par exemple (Figure 38) : pour un protocole MRV sur un canal ch1 entre deux modules P1 et P2, le processus P1 peut être prêt à émettre une valeur val_out sur le canal ch1 (c'est à dire être dans une condition de non blocage : la transition est étiquetée par 'ok'). Au même instant, le processus P2 peut être prêt à recevoir la valeur val_in sur le canal ch1.

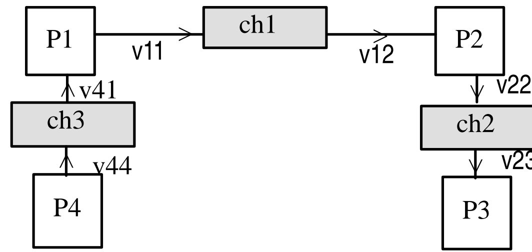


Figure 38 : Un exemple

Sur un même schéma, il y a deux sortes de transitions possibles : une correspondant au fait que P1 est dans un état de blocage et P2 peut être en mode réception ou émission sur un canal différent de ch1 ; d'un autre côté, une autre transition correspondant au fait que P2 est dans un état de blocage, mais P1 peut être émetteur ou récepteur sur un autre canal. C'est ce qu'exprime la formulation suivante :

$$\{\text{ SEND(ch1,v11, ok || RECEIVE(ch1,v12) ok}\}$$
$$\cup\{\text{SEND(ch1,v11), nok || EvtSt(chi,val_in) X} \mid i\neq 1\}$$
$$\cup\{\text{ EvtSt(chi,val_out), X || RECEIVE(ch1,v12) nok} \mid i\neq 1\}$$

où la variable X peut prendre la valeur 'ok' dans le cas où il n'y a pas de situation de blocage, ou 'nok' sinon. Val_in et Val_out sont respectivement les valeurs reçues ou émises sur un canal. EvtSt peut prendre indifféremment la valeur SEND ou RECEIVE

La transition est en fait étiquetée par un nombre d'événements de communication égal au nombre des processus. Ainsi, le produit Cartésien de l'ensemble des transitions de chaque canal doit être pris en compte.

L'intérêt d'une approche Prédicat dès que le modèle devient plus complexe

Dans l'exemple précédent, si nous ajoutons un troisième processus P3 relié à P2 par le canal ch2 de protocole MRV, et si nous ajoutons également un quatrième processus P4 relié à P1 par le canal ch3 de protocole MRV, l'ensemble des transitions possibles (les événements sont donnés dans l'ordre des processus P1, P2, P3, P4) est le suivant :

{SEND(ch1,v11) ok || RECEIVE(ch1,v12) ok
|| RECEIVE(ch2,v23) nok || SEND(ch3,v44) nok }
∪ {SEND(ch1,v11) nok||SEND(ch2,v22) ok
|| RECEIVE(ch2,v23) ok || SEND(ch3,44) nok }
∪ {RECEIVE(ch3,v41) ok || RECEIVE(ch1,v12) nok
|| RECEIVE(ch2,v23) nok || SEND(ch3,v44) ok }
∪ {RECEIVE(ch3,v41) ok || SEND(ch2,v22) ok
|| RECEIVE(ch2,v23) ok || SEND(ch3,v44) ok }

Dans cet exemple, nous remarquons que deux transferts d'informations concurrents peuvent se produire sur deux canaux concurrents (c'est le dernier cas).

Il est certain que dans le cas d'un système plus complexe, le calcul manuel de toutes les combinaisons possibles des transferts d'information est irréaliste. Aussi, une approche basée sur les prédicats est plus appropriée pour exprimer, de manière symbolique, les liens logiques entre les canaux sur lesquels les processus sont actifs et l'état (bloqué ou non bloqué) de chaque processus.

Enfin, le module de communication doit considérer le calcul des valeurs (val_in) de chaque canal, fournies en entrée aux processus, sur la base des valeurs (val_out) qui avaient été transmises par les processus émetteurs du canal. En fait, en raison du délai introduit par le pas de simulation, il est nécessaire de considérer une ressource pour un protocole tel que VSIG ; la valeur Val_in est ainsi calculée à partir de la valeur courante de cette ressource, qui elle–même évolue selon les valeurs transmises val_out.

Nous remarquons que le protocole MRV n'a pas besoin de ressources. En effet, le transfert d'information se produit au cours du même pas de simulation dans la mesure où émetteurs et récepteurs sont prêts.

II.4 – L'évolution de l'ensemble du modèle

Il est important de voir comment notre modèle se comporte dans sa globalité et notamment comment se synchronisent les différentes représentations STE des processus d'une part et du module de communication d'autre part.

A chaque pas de simulation, tous les processus envoient leur activité au module de communication. A partir de ces informations, c'est le module de communication qui contrôle le blocage ou non de chaque processus. Comme ce module commande le blocage ou non de tous les processus au cours du tirage de son unique transition, c'est lui qui synchronise l'évolution du système complet. C'est ce que nous représentons par la Figure 39 et la Figure 40 .

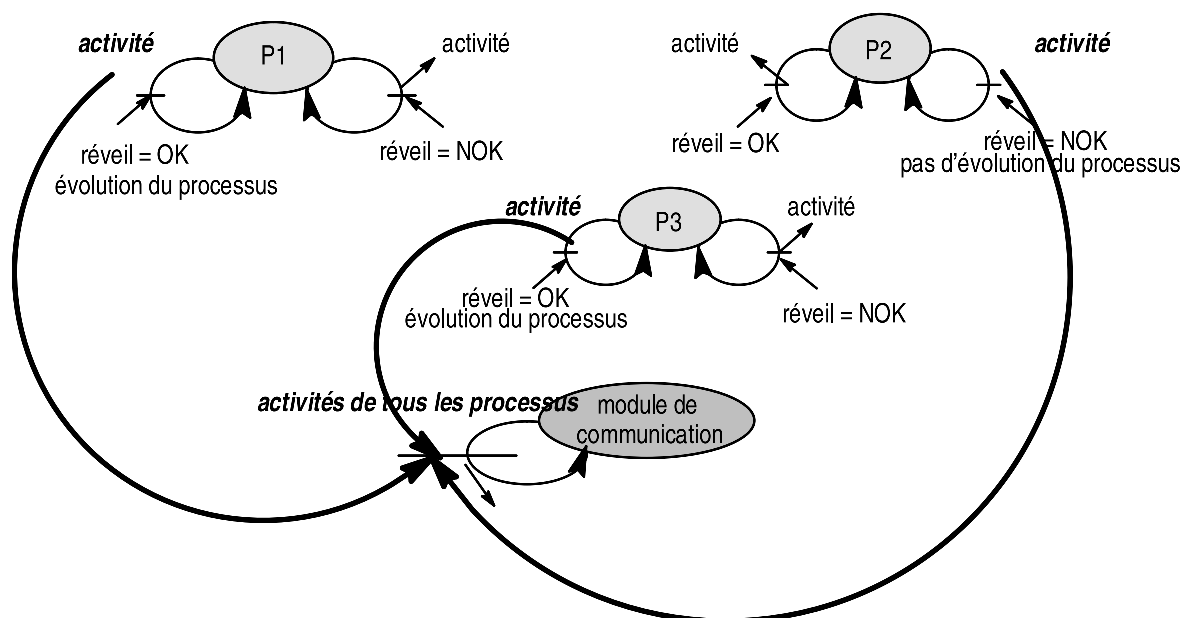


Figure 39 : Le module de communication reçoit l'activité des tous les processus

Pour des raisons de lisibilité, nous présentons deux schémas pour exprimer la séquentialité des actions mais en ce qui concerne leur observabilité, il y a atomicité (elles sont effectuées en un seul pas de simulation VOVHDL).

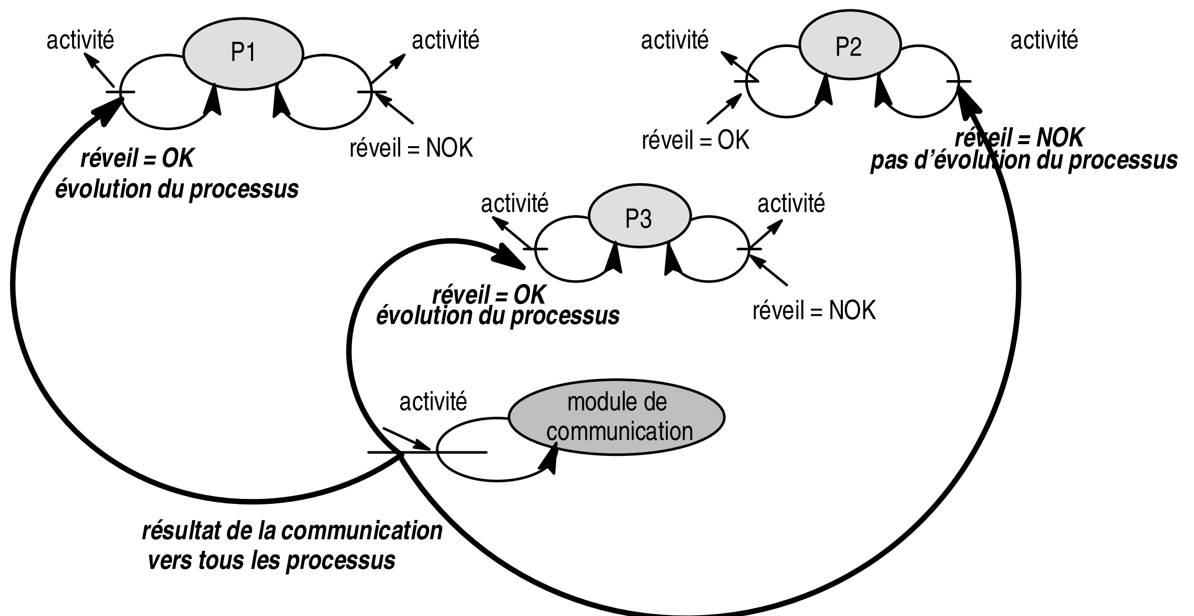


Figure 40 : Réception synchrone des résultats de communication par les processus

A chaque pas de simulation la transition du module de communication est tirée et fusionne avec une des deux transitions associées à chaque processus.

III – DANS LA PRATIQUE

L'objectif de cette partie est de faciliter la compréhension des concepts énoncés dans les parties I et II de ce chapitre.

Pour cela nous considérons le petit exemple suivant Figure 41 :

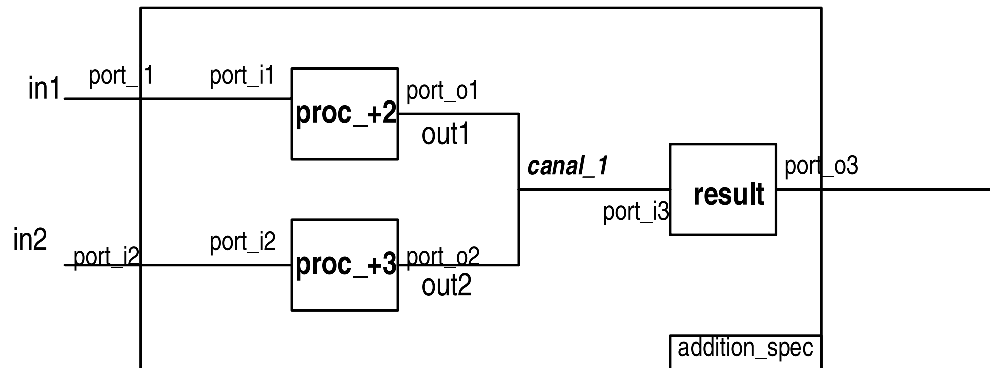


Figure 41 : un exemple

La fonction de ce petit exemple est la suivante :
la valeur *in1* est transmise au module *proc_+2* qui lui ajoute la valeur 2 si *in1* vaut 2 uniquement et renvoie la valeur obtenue *out1* sur le canal *canal_1*. Au cas où i*n1* est différent de 2, *out1* reçoit la valeur de i*n1*. La valeur *in2* est transmise au module *proc_+3* qui l'incrémente de 3 et transmet le résultat sur le *canal_1*. Le canal de communication *canal_1* est défini par le protocole de communication MRV avec pour règle de composition CR= all et pour règle de répartition DR =some.
Le module *résult* ne fait que stocker le résultat et émettre la valeur sur le port de sortie *port_o3*

III.1 – Une description VOVHDL

Nous supposerons que le formalisme sous lequel les concepteurs nous fournissent une description d'architecture est une description VOVHDL. En réalité dans le contexte de notre projet, les concepteurs ont obtenu cette description à partir d'un formalisme intermédiaire (cf chap1 II.3).

Une description VOVHDL se présente de la manière suivante :

Une base de données

Un fichier package contient la définition des types et des fonctions prédéfinies.

```
Package USER_PACKAGE is
        type  nb_pred_type is ( 0, 1, 2, 3, 4, 5, 6, 7,8, 9);
```

Dans notre exemple, nous nous limitons au transfert de messages entiers de 0 à 9. Nous supposerons que les valeurs des messages *in1* et *in2* sont inférieures à 6.

La description du système

L'entité permet de définir l'enveloppe de la boite *addition_spec* et l'architecture contiendra sa composition (ici trois processus). Chacun des processus sera défini de manière comportementale.

```
    ENTITY addition_spec is
            port (   port_i1 : in MRV  nb_pred_type;
                     port_i2 : in MRV  nb_pred_type;
                     port_o3 : out MRV nb_pred_type);
    end addition_spec

ARCHITECTURE structure of addition_spec is

        channel canal1 : nb_pred_def, MRV, all (proc_+2(port_o1), proc_+3 (port_o2)), some
(result (port_i3));

        component proc_+2
                port  ( port_i1 : in , MRV nb_pred_type;
                        port_o1 : out , MRV, nb_pred_type;)
                end component;
        component proc_+3
                port  (  port_i2 : in , MRV, nb_pred_type;
                         port_o3 : out, MRV, nb_pred_type);
                end component;
        component result
                port  ( port_i3 : in , MRV nb_pred_type;
                        port_o3 : out , MRV, nb_pred_type;)
                end component;
end structure
```

Figure 42 : Le module addition_spec

Chaque processus qui est un *component* de la structure de plus haut niveau *addition_spec* est alors décrit de manière comportementale :

```
    ENTITY proc_+2 is
            port  (  port_i1 : in , MRV nb_pred_type;
                     port_o1 : out , MRV, nb_pred_type;)
    end proc_+2

   ARCHITECTURE proc2arch of proc_+2 is
   begin
           process2 :
           process
           variable in1, out1 : nb_pred_type;
           begin
                   receive ( port_i1, in1);
                   IF in1 = 2  THEN out1 := in1 +2 ELSE out1 := in1;
                   send (port_o1, out1);
   end process;
   end proc2arch;
```

Figure 43 : Le processus Proc_+2

```
ENTITY proc_+3 is
        port (  port_i2 : in , MRV nb_pred_type;
                port_o2 : out , MRV, nb_pred_type;)
end proc_+3

ARCHITECTURE proc3arch of proc_+3 is
begin
        process3 :
        process
        variable in2, out2 : nb_pred_type;
        begin
                receive ( port_i2, in2);
                out2 := in2 +3;
                send (port_o2, out2);
end process;
end proc3arch;
```

Figure 44 : Le processus proc_+3

III.2 – Un processus VOVHDL sous forme STE

Chacun des processus VOVHDL aura une modélisation correspondante sous forme d'un système à transition étiquetées.

Le processus proc_+2 peut être traduit par le STE suivant :

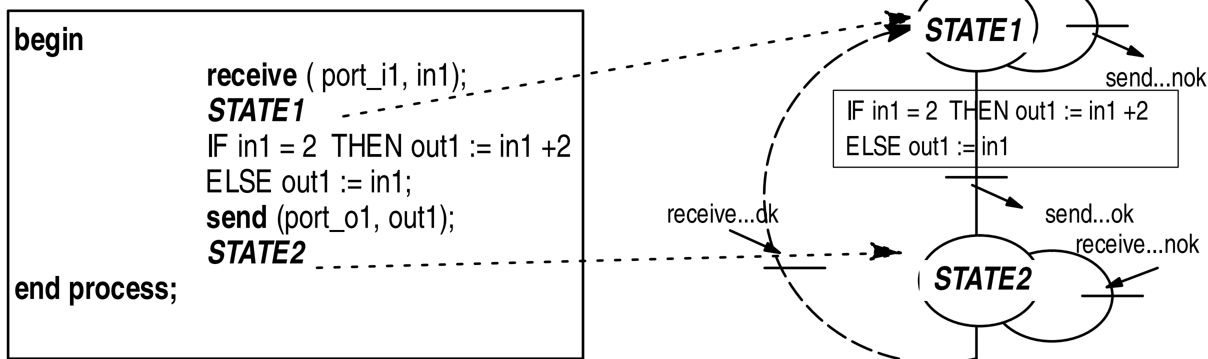


Figure 45 : Un processus VOVHDL déplié sous forme de STE

Notre principe de modélisation nous permet de modéliser ce processus au moyen d'un STE constitué du seul état : "attend_evt" (Figure 46). Chacune des deux transitions tirables depuis cet état correspond :

- soit à une évolution du processus, c'est à dire au calcul de valeurs de sorties "data_out" en fonction des valeurs d'entrées "data_in" d'une procédure C.
- soit à une non évolution du processus.

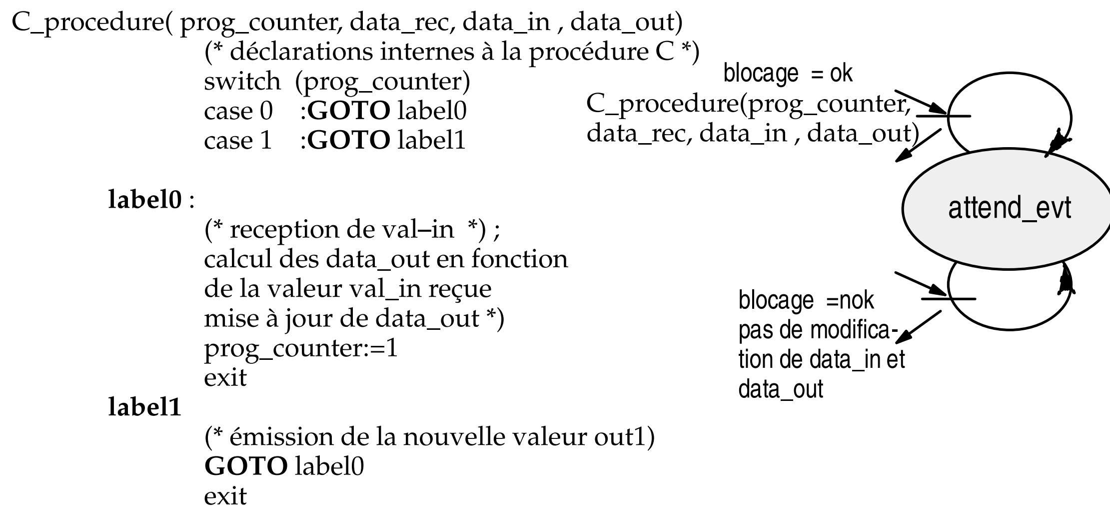


Figure 46 : Modélisation STE en utilisant la décomposition générique/particularisée

III.3 – Le module de communication

C'est lui qui réalise la synchronisation de tous les processus précédemment décrits. La structure est indépendante du système modélisé. Ce module est constitué d'un état unique sur lequel on boucle en appelant une fonction qui calcule les résultats de communication de tous les processus en fonction de leurs activités.

Cette fonction est donnée dans la suite sous forme algorithmique mais elle est complètement définie dans les fichiers comn.bdy de l'annexe 2 (ce fichier contient la définition complète d'une boite de communication selon une syntaxe pascal)

La fonction *simulation* reçoit en entrée l'activité de chaque processus (comme décrit au paragraphe précédent) et renvoit le résultat sous forme de blocage de processus ou de réalisation de la communication. Le module de communication a accès aux données qui caractérisent chaque canal.

Pour chaque canal :

rechercher les émetteurs et les récepteurs actifs du canal

vérifier les règles CR et DR du canal (en cas de non vérification bloquer les processus)

Pour chaque protocole, vérifier si la communication est réalisée : si oui mettre à jour les ressources et débloquer les processus sinon, bloquer les processus concernés.

LA MODELISATION

CHAPITRE 3

Les principes de modélisation mis en évidence au cours du chapitre 2 ont été implantés sur le cas particulier du composant FICOMP au moyen de l'outil ASA+.

Le chapitre 2 nous a permis de modéliser un système constitué de modules qui ne communiquent que sur un seul niveau. La description du composant FICOMP a nécessité la mise en oeuvre de ces principes sur une structure hiérarchique.

Après une présentation de l'outil, nous décrivons les aspects structurels et fonctionnels de notre modèle. Une architecture hiérarchisée et générique permet de modéliser un système constitué de modules qui communiquent entre eux et s'échangent des informations sur plusieurs niveaux.

Le comportement du modèle d'un processus, décrit de manière générique, et celui du module de communication nous donnent une vision fonctionnelle de l'ensemble.

Enfin, une modélisation particulière de l'environnement nous permet de réaliser l'interface entre une sémantique CCS et une sémantique VOVHDL.

I – PRESENTATION DE L'OUTIL ASA+ [Ve, 94] ,[BS, 95]

I.1 – Un outil industriel

I.1.1 – Des applications diverses

Afin de répondre à des besoins d'analyse comportementale de systèmes sur la base de techniques formelles de validation, vérification et test, l'outil ASA+ permet de décrire une large gamme de systèmes (d'une description macroscopique vers une description plus détaillée). Son environnement de simulation permet de vérifier des spécifications par rapport au besoin, d'évaluer des performances, d'évaluer des sûretés de fonctionnement et de générer des tests de validation.

Plusieurs applications ont montré à la fois la faisabilité et l'intérêt de ces techniques dans un contexte industriel (secteur de l'énergie électrique, du transport ferroviaire et spatial, de l'armement).

I.1.2 – Spécification et analyse : deux besoins

La motivation d'un tel outil est de réunir sur un même socle informatique les besoins liés à la spécification (ce qui nous conduit à la définition du langage) et à l'analyse (fortement induite par le formalisme).

La maîtrise de la complexité d'un système passe par une approche méthodologique associée non seulement à l'outil mais encore au domaine couvert. Le noyau de vérification repose sur le calcul du graphe des états atteignables qui doit être calculable donc fini. Dans le cas contraire, seul un graphe partiel est calculé.

C'est pourquoi, il est nécessaire de distinguer une utilisation soit en vue de besoins de spécification, soit en prévision des besoins d'analyse.

Dans le cadre de la spécification, l'objectif est de décrire le système de la manière la plus complète, en cohérence avec le niveau d'abstraction considéré. C'est une vision de maquettage en vue d'obtenir un modèle exécutable.

En ce qui concerne l'analyse, il s'agit d'apporter un maximum de garanties sur le comportement spécifié. Il s'agit d'extraire de la spécification un modèle (par simplification, abstraction, discrétisation...) pour lequel les vérifications de propriétés restent significatives.

I.2 – Le langage LSA+

I.2.1 – Une architecture statique

L'architecture d'un système est saisie au moyen de l'éditeur graphique SADT (Structured Analysis Design Techniques) Ce dernier permet d'établir des architectures fonctionnelles statiques. L'approche est hiérarchique et permet de maîtriser la complexité.

Un système est décrit sous forme d'un ensemble de boîtes connectées par des ports. Chaque module est défini sous forme d'une boîte à laquelle est associé un fichier qui contient son comportement. Ce fichier comprend un ensemble de transitions de la forme de celle décrite par la Figure 48.

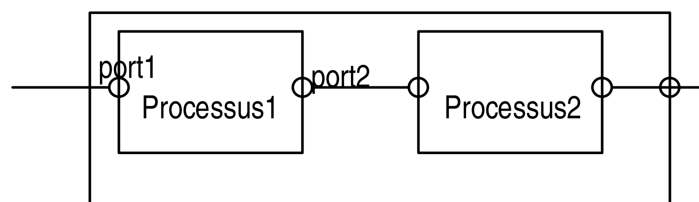


Figure 47 : La donnée d'une architecture sous ASA+

I.2.2 – Une sémantique de comportement

La sémantique de comportement est basée sur les Réseaux de Pétri Prédicats–Transitions [LLo 90]. Le langage possède une structure hiérarchisée de type SADT. Les corps des transitions sont décrits en Pascal.

```
TRANS nom_de_la_transition
        WHEN reception(message1, port1)
        FROM ETAT_DEP
        TO ETAT_ARR
        BEGIN
                corps de programme de la transition
        END;
        OUTPUT emission(message2, port2)
END;
```

Figure 48 : Une transition dans l'outil ASA+

La signification de la transition de la Figure 48 est la suivante :
lorsque le message *message1* est reçu sur le port *port1* et que le prédicat *condition* est vérifié alors la transition est tirable et le module passe de l'état ETAT_DEP à l'état ETAT_ARR et exécute le programme contenu dans le corps de programme. Il émet le message *message2* sur le port *port2.*

Le langage de l'outil, LSA+ (langage de programmation utilisé pour décrire les automates) intègre un sous–ensemble du langage ESTELLE normalisé à l'ISO [ES, 89] : cela offre la possibilité de concevoir des architectures de communication ; le contexte d'interprétation de l'outil est celui des automates communicants. Ce langage permet notamment de spécifier des architectures répétitives et d'exprimer la dynamique de comportement par le biais d'automates étendus.

En outre, ASA+ hérite du concept de généricité du langage ESTELLE : des bibliothèques de composants configurables et réutilisables sont alors définies.

I.2.3 – Une sémantique de communication

La communication est basée sur le mécanisme du rendez–vous de l'Algèbre CCS étendu au Rendez_Vous multiple. Il est ainsi possible de modéliser une communication synchrone au moyen du rendez vous. De plus, l'utilisation de macro fonctions au niveau des ports de communication permet de modéliser une communication asynchrone (réalisée au moyen de files d'attentes). Une communication en diffusion sera réalisée par extension du Rendez_Vous au Rendez_Vous multiple.

I.2.4 – Un système temps réel

La description directe des caractéristiques spécifiques des systèmes temps réel par des contraintes temporelles définies par des plages de tir associées à des distributions de probabilités [At, 94] [ACJV, 93] est possible. La sémantique du temps est celle des réseaux de Pétri temporisés stochastiques.

I.3 – Des moyens de spécification

La simulation nous permet avant tout de mettre au point des spécifications et d'évaluer des choix d'architecture. Nous avons besoin également de valider l'enchaînement des fonctions par rapport à l'expression du besoin, les procédures d'utilisation d'un système afin d'en permettre la formation des utilisateurs par une animation du modèle. Enfin, il est utile de valider l'interface d'un système avec des applications externes par une simulation hybride.

Pour cela, trois modes de simulation sont possibles :

- la simulation interactive (l'utilisateur choisit un stimulus externe à appliquer à la spécification et résout le non déterminisme entre transitions internes) ou automatique (choix aléatoire en cas de multiples possibilités) d'un modèle. Le simulateur offre alors des caractéristiques usuelles d'un debogueur telles que les points d'arrêt, la configuration de trace, la ré–exécution d'un scénario.
- la simulation avec prise en compte de contraintes temporelles. Un simulateur aléatoire permet une analyse quantitative. Celle–ci est effectuée sur la simulation intensive de modèles basée sur un algorithme de Monte–Carlo. Par exemple, l'analyse peut porter sur des critères de performances tels que le temps de réponse utilisateur, le débit d'un medium de communication, le taux d'utilisation d'une ressource, ou des critères de sûreté de fonctionnement tels que la fiabilité et la disponibilité du système ou le temps moyen de réparation d'un équipement.
- La simulation exhaustive permet d'obtenir le graphe des états atteignables du modèle. Celui–ci représente le comportement global du système. Cela consiste à explorer, à chaque instant, chaque transition tirable, selon chaque valeur possible (en fonction de leur typage) des messages en entrée de la transition.

I.4 – Des moyens d'analyse

I.4.1 – Validation et Vérification

L'analyse comportementale est supportée par un ensemble d'outils qui exploitent le graphe des états atteignables obtenu par simulation exhaustive. Ces outils sont choisis avec un objectif de validation (vérifier qu'il n'y a pas de dysfonctionnements tels que des états bloquants, des non–réinitialisations ou des codes morts) et de vérification de l'adéquation aux besoins. Il peut alors s'agir soit de propriétés portant sur la logique de fonctionnement (par exemple l'absence de doubles fautes) soit de génération de scénarios conduisant à des événements redoutés.

Les outils d'analyse peuvent être classés selon les trois groupes suivants :

- Ceux basés sur la logique d'ordre 1 : un analyseur logique vérifie des propriétés intrinsèques, telles que la présence d'états puits ou de cycles internes, ou évalue des expressions ou des invariants (formules logiques par rapport aux états et transitions du graphe d'états).
- Ceux qui utilisent des expressions de la logique modale : l'analyse du comportement repose sur l'évaluation de propriétés telles que la séquentialité d'événements ou le changement d'états, en termes d'inévitabilité ou de potentialité : est–il inévitable d'aboutir à tel état, après l'occurrence de tel événement ?
- Enfin ceux qui utilisent l'approche opérateurs équivalence. Des opérateurs de projection basés sur différentes équivalences (test, observationnelle, comportementale) [Fe, 88], [Ve, 89] permettent d'extraire un modèle comportemental complet qui dépend du point de vue adopté par l'utilisateur. L'intérêt de ces opérateurs de projection (observationnelle et comportementale) est d'être à la fois congruants à la composition et de conserver les propositions de logique modale. Il est ainsi possible de maîtriser la complexité d'un système (composition des projections plutôt que projection de la composition) et de démontrer une propriété du système sur une des abstractions de ce système.

I.4.2 – Génération de scénarios de test

En outre, un générateur de scénarios fournit une séquence d'entrée permettant d'atteindre une situation donnée. Cette situation est exprimée sous forme d'une expression logique.

Il permet de générer des scénarios de test de validation de manière automatique à partir des spécifications. Ces scénarios peuvent provenir soit d'une simulation manuelle et interactive, soit d'une simulation aléatoire, soit de l'analyse du graphe d'états (dans ce cas, tous les scénarios associés à un critère donné peuvent être calculés).

II – L'ASPECT STRUCTUREL DU MODELE

II.1 – Une structure générique et hiérarchique

II.1.1 – Une sémantique de communication distribuée

Une description VOVHDL est généralement définie sur plusieurs niveaux. Une solution serait de remettre à plat toute la communication et de la traiter globalement. Mais il semble intéressant de rester proche de la définition multi–niveau ne serait–ce que pour des raisons de visibilité de la structure. Aussi, afin de conserver la structure hiérarchique du circuit, nous devons être capable de traiter la communication correspondant à un niveau donné ainsi que la gestion de la communication provenant des niveaux inférieurs et supérieurs.

Cela est réalisable car la sémantique de communication de VOVHDL préserve la structure hiérarchique. En effet, si nous considérons un ensemble de processus VOVHDL (par exemple le système S de la Figure 49) eux–mêmes décomposés en sous–processus, il est possible d'analyser les échanges entre processus en conservant la structure.

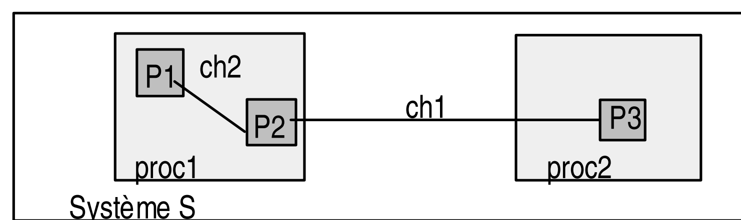


Figure 49 : Un exemple

La communication du système pourra être analysée selon les étapes suivantes :

1 – Analyse de la communication au niveau le plus bas : analyse indépendante des processus *proc1* et *proc2* (ces deux actions pouvant être traitées en parallèle). Pour *proc1* il s'agira de la communication sur *ch2* (communication interne à *proc1*) et *ch1* (communication externe à *proc1*).

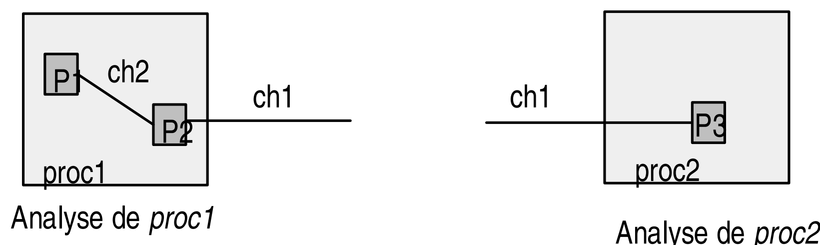


Figure 50 : Analyse de plus bas niveau

2 – Analyse de la communication au niveau supérieur (analyse de *système S*). Il s'agit de voir les modules *proc1* et *proc 2* comme des boîtes noires et en l'occurrence de considérer uniquement le canal de communication *ch,* qui devient un canal de communication interne à *système S* (Figure 51).

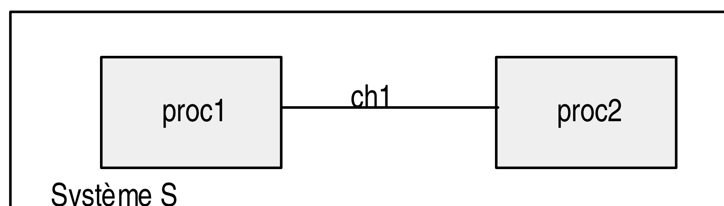


Figure 51 : Analyse de niveau supérieur

II.1.2 – Une structure générique

La modélisation d'une structure VOVHDL pourra être représentée de manière générique et chaque niveau sera constitué d'un ensemble de modules correspondant à l'ensemble des processus VOVHDL de ce niveau et d'un module supplémentaire : la boîte de communication du niveau. Celle–ci gèrera la communication interne et transmettra au niveau supérieur les informations concernant les canaux de communication externe. La structure obtenue est ainsi une structure récursive.

Nous considérons la généricité au sens où chaque module de communication a le même comportement et donc le fichier comportemental qui lui est associé est défini une fois pour toutes. De même, le module comportemental de chaque processus est identique par son comportement. Chaque processus ne se distingue que par sa partie Particularisée (cf chapitre 2, II .2.2).

Ainsi, chaque boîte de communication d'un niveau n donné est un module d'un niveau n+1.

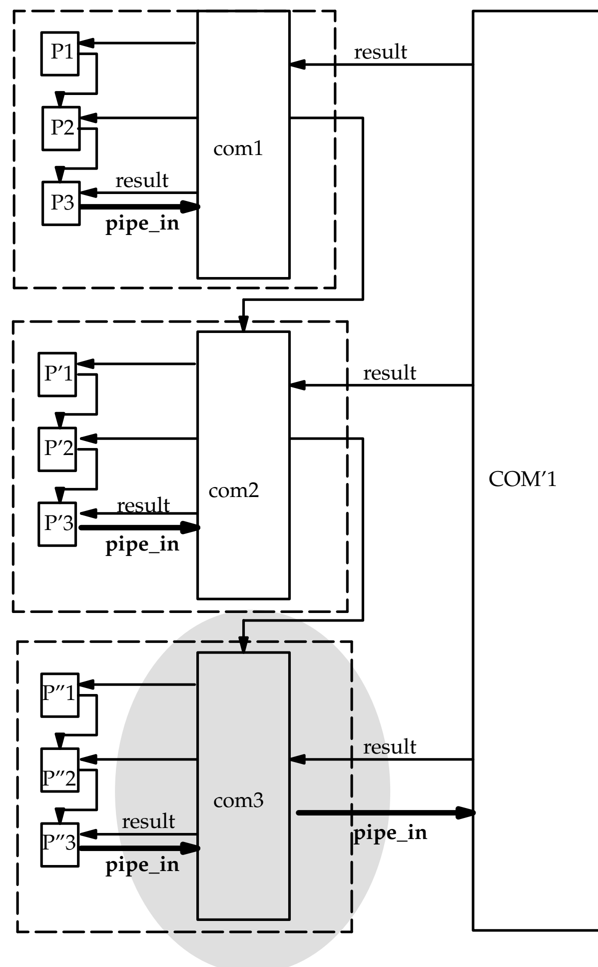


Figure 52 : Description hiérarchique multi–niveaux

La structure hiérarchique du circuit peut être préservée par la définition d'un modèle générique comme le schématise la Figure 52 . Chaque sous–ensemble représenté par un cadre en pointillé est un processus pour le niveau associé à la boite com'1.

II.1.3 – Conservation du caractère synchrone

Afin de préserver le caractère synchrone de la sémantique à un niveau donné, la boîte de communication de ce niveau doit recevoir de manière synchrone les activités de tous les modules du même niveau. Or l'outil ASA+ n'autorise qu'une seule condition de garde en entrée. C'est pourquoi nous avons défini un prin-

cipe de *pipe* qui connecte en chaîne tous les modules d'un niveau donné avec le module de communication et, par le principe de fusion des transitions, la réalisation de la communication à un niveau donné est synchrone.

Par exemple : une boîte de communication doit recevoir simultanément l'état d'activité des modules P1, P2 et P3. Or une transition ASA+ ne peut pas recevoir plusieurs événements de communication en entrée. Aussi, afin de conserver l'aspect synchrone, la structure est pipelinée de la façon suivante : le premier module envoie son activité à un deuxième module frère. Le deuxième module transmet au troisième l'activité des modules 1 et 2 et ainsi de suite jusqu'au dernier module qui transmet toute l'information au module de communication. Toutes ces occurrences de communication se sont déroulées en un seul pas de communication par le principe de la fusion des transitions.

A chaque niveau, un module *starter* permet d'initialiser les modules du circuit. Le premier module garde ainsi son caractère générique. Tous les modules *starters* d'un même niveau de profondeur de composition sont synchronisés sur la base d'un même signal d'horloge *CK*. La synchronisation est ainsi généralisée par niveau afin d'éviter un entrelacement dans l'exécution de ces modules.

A un niveau de profondeur donné de l'architecture, le modèle se présentera donc suivant le schéma de la Figure 53 qui représente l'implantation du module *starter* et l'enchaînement des communications pipelinées.

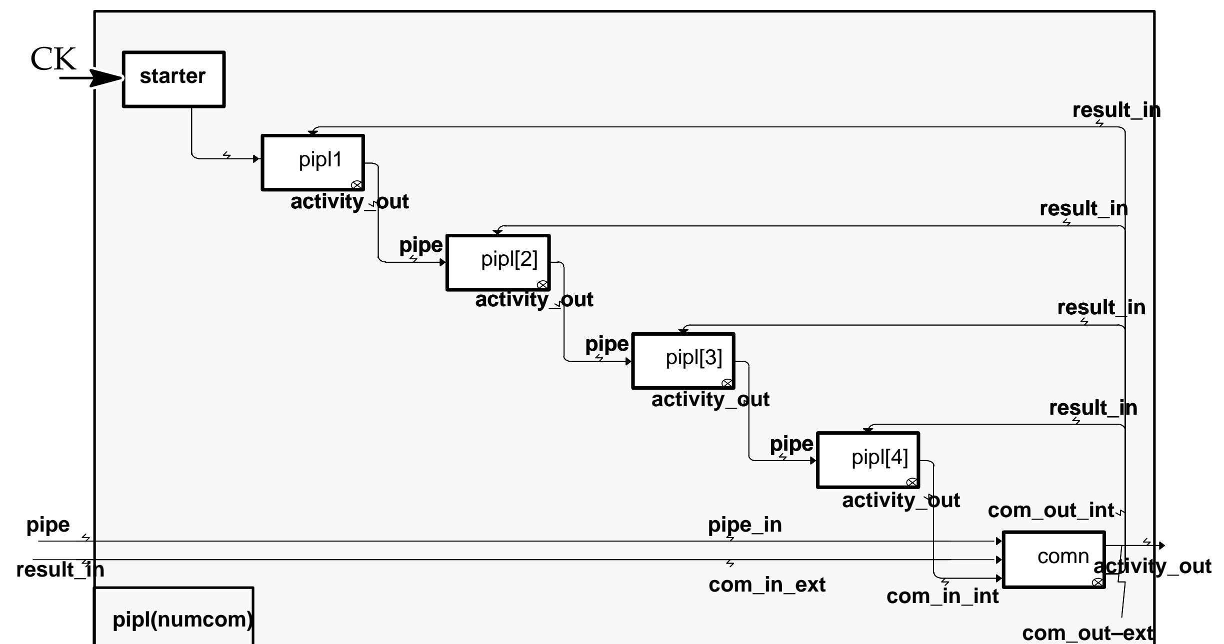


Figure 53 : Le modèle ASA+ à un niveau n

Si la boîte *pipl* est terminale, elle contient la définition du comportement d'un processus VOVHDL sinon, à un niveau plus haut, chaque boîte *pipl* est une structure *pipl(numcom)* où *numcom* est le rang du sous–bloc frère du niveau hiérarchique supérieur. Ce numéro doit être connu de la boîte de communication afin de gérer les informations sur la communication externe.

III – L'ASPECT COMPORTEMENTAL DU MODÈLE

A partir d'une activité engendrée par l'environnement, le flot d'informations se répercute dans les niveaux inférieurs jusqu'au niveau le plus bas. Chaque boîte de communication d'un niveau remonte l'information (résultat du calcul de la communication) et, par remontées successives, l'environnement récupère le résultat issu de l'activité initialement émise.

III.1 – La cellule de communication (comn.bdy)

III.1.1 – Son fonctionnement

C'est par elle que passe toute la communication. Sa fonction est d'une part de gérer la communication interne et d'autre part de transmettre l'information nécessaire à la communication externe (gestion des niveaux supérieurs et inférieurs).

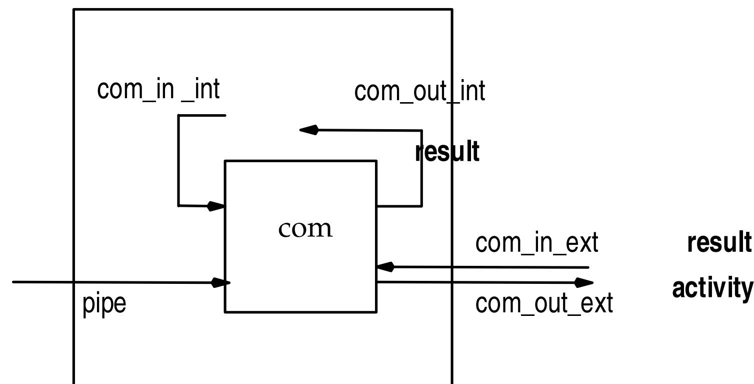


Figure 54 : Une cellule de communication

La cellule reçoit l'activité de tous les canaux VOVHDL de son niveau. En fonction de ces activités et des propriétés des canaux (règles de composition, règles de répartition, protocoles) elle doit réaliser ou non la communication. Aussi, pour chaque canal, l'activité des émetteurs (s'ils émettent ou non et si oui la valeur émise) et l'activité des récepteurs (s'ils sont prêts à recevoir un message ou s'ils sont bloqués) sont analysées. En fonction de cette analyse et des règles de communication du canal un résultat est produit (émetteurs et récepteurs peuvent être bloqués, un récepteur a reçu telle valeur).

III.1.2 – Son interprétation STE

La modélisation STE du comportement de cette boite de communication peut être réalisée au moyen de trois états et trois transitions comme le détaille la Figure 55.

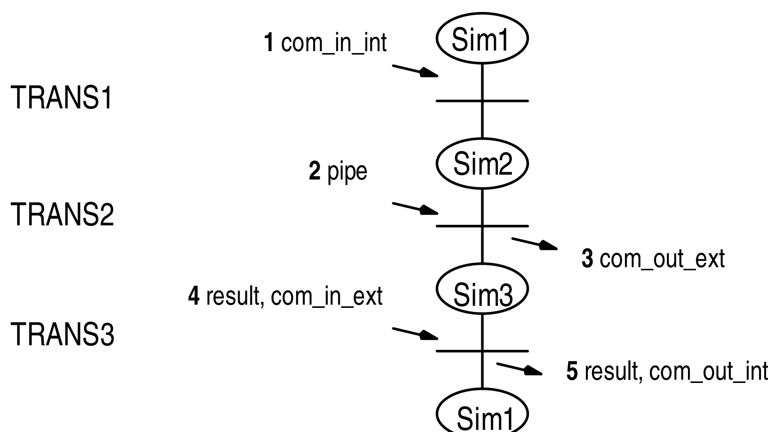


Figure 55 : Le fonctionnement d'une cellule de communication

III.1.3 – L'interprétation avec l'outil

La modélisation STE est représentée par les trois transitions suivantes extraites de l'annexe II qui contient le programme complet d'une boîte de communication.

Une transition se lit de la manière suivante : Les clauses WHEN et OUTPUT représentent respectivement les conditions des stimuli en entrées et en sorties, les clauses FROM et TO indiquent les états du système. Les opérations effectuées au cours du tir de la transition sont indiquées entre les termes BEGIN et END.

```
TRANS1
        WHEN int_com_in.mess(tab_act)
        FROM SIM1
        TO SIM2
  BEGIN
  simulation (tab_act, data_out_sup,tab_res_int, liste_rec_pred_tab, BUFF_IN_VECT);
  END;
```

La fonction simulation reçoit en entrée les variables *tab_act* : activité des processus du niveau et calcule le résultat de la communication (*tab_res_int* et *BUFF_IN_VECT*) et l'activité du niveau supérieur (*data_out_sup*). L'activité d'un processus émetteur consiste à signaler sur quel canal il émet et quelle est la valeur émise ou bien pour un processus en réception, s'il est prêt à recevoir ou non. Le résultat de la communication consiste à signaler si le processus émetteur est bloqué ou non et pour un processus récepteur quelle est la valeur reçue et sa condition de blocage ou déblocage.

```
TRANS2
        WHEN pipe_in.mess(tab_act)
        FROM SIM2
        TO SIM3
        BEGIN
        construire(data_out_sup, num_com, tab_act, tab_act_out);
        OUTPUT ext_com_out.mess(tab_act_out);
        END;
```

La fonction *construire* ajoute la contribution de cette boîte de communication (*data_out_sup*) au tableau (*tab_act_out*) contenant la contribution des modules frères d'indice inférieur (inférieur à *num–com*).

```
TRANS3
        WHEN ext_com_in.mess(tab_res)
        FROM SIM3
        TO SIM1
        BEGIN
        merger(tab_res_out, tab_res, num_com, tab_res_int);
        OUTPUT int_com_out.mess(tab_res_out);
        END;
```

La fonction *merger* regroupe les résultats issus des boîtes de communication de niveau supérieur avec celui de la boîte de communication interne à ce niveau (*num–com*) et le dirige vers les processus concernés.

III.2 – Le fonctionnement d'un module

III.2.1 – La partie Générique

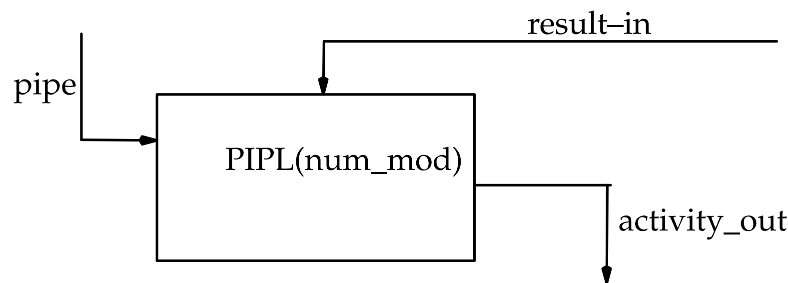


Figure 56 : Les entrées/sorties d'un module associé à un processus

Afin de préserver le caractère synchrone, nous avons vu que les processus se décomposaient en partie Générique et partie Particularisée. Le comportement de la partie générique sera le suivant :

- le module reçoit par l'intermédiaire du port '*pipe*' l'activité des modules frères précédents c'est–à–dire de numéro inférieur à *num_mod*.
- Le module transmet son activité et celle des modules précédents au module frère suivant s'il existe , sinon, à la boîte de communication.
- Lorsque la communication du niveau inférieur est traitée, le résultat relatif à la boîte *pipl(num_mod)* lui est directement retransmis par la boîte de communication du niveau supérieur (si les événements multiples en entrée ne sont pas autorisés en ASA+, en revanche les sorties multiples lors du tir d'une transition sont possibles).

III.2.2 – La partie Particularisée

Lorsqu'une boîte est terminale un comportement VOVHDL lui est associé. Ce comportement est géré par la partie Particularisée du processus VOVHDL. La partie particularisée est implantée sous la forme d'une procédure ré–entrante écrite en C.

L'exécution du modèle repose sur l'extraction de la machine d'état de la description VOVHDL de chaque processus. Or cette exécution revient à simuler le modèle pas à pas.

La simulation ASA+ est exécutée conjointement à l'exécution des routines C. Il s'agit du principe de la simulation conjointe. Au cours de l'exécution du modèle ASA+, celui–ci fait appel à l'exécution de la routine C. Le pas de simulation du coté ASA+ est lié à l'occurrence de deux événements de communication. A partir d'un événement de communication, le processus "passe la main" au simulateur ASA+ qui se déroule suivant les étapes décrites au paragraphe précédent puis rend son résultat au processus. Celui–ci en fonction des variables d'entrées (transmises par le simulateur ASA+) et en fonction du point où il était resté stoppé reprend son exécution jusqu'à la rencontre d'un nouvel événement de communication ; Il transmet alors au simulateur les nouvelles valeurs de ses variables et se bloque sur cet événement de communication.

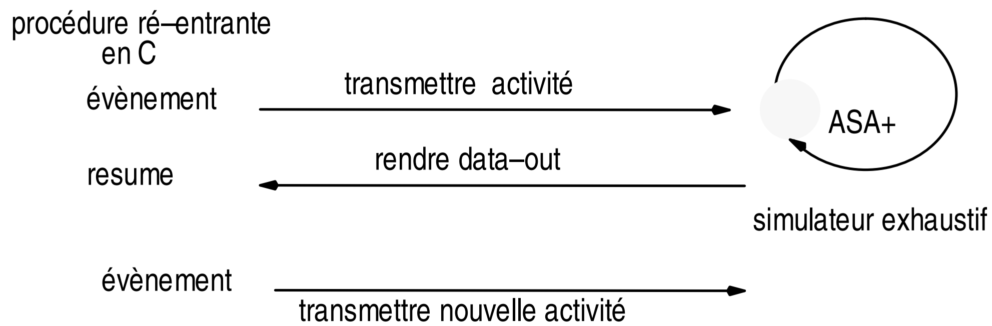


Figure 57 : La simulation conjointe

Le module ASA+ gère la communication et l'appel à la procédure C. Après avoir reçu l'activité des modules frères et transmis celle–ci ainsi que sa propre activité, le module passe dans un état où il reçoit le résultat de la boîte de communication. Si le résultat de la communication est tel que le module doive rester bloqué, alors il émettra de nouveau au pas suivant la même activité. Sinon, il y a appel à la procédure C et la nouvelle activité est calculée pour le pas suivant et est transmise en entrée de la procédure C représentative de l'exécution du processus.

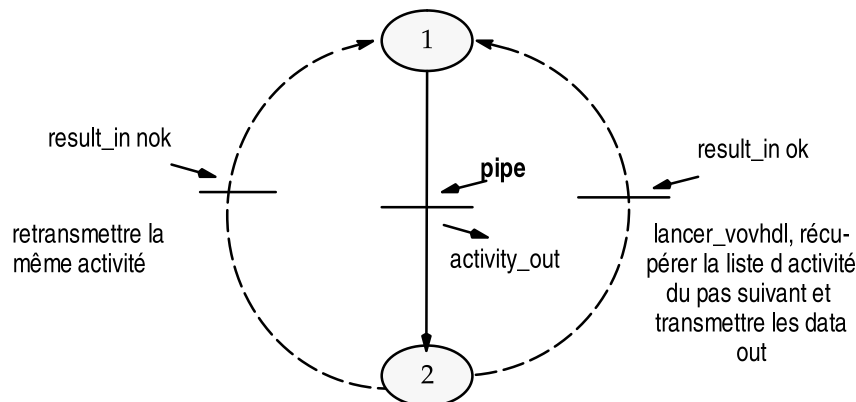
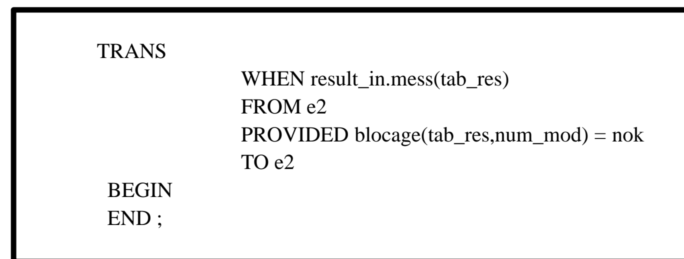


Figure 58 : Appel à la procédure C vu depuis un module

La description du comportement d'un processus (extraite de l'annexe II) est rappelée ci–dessous :

```
TRANS
        WHEN result_in.mess(tab_res)
        FROM e2
        PROVIDED  blocage(tab_res,num_mod) = ok
        TO e1
BEGIN
        data_in := tab_res[num_mod];
        appel_de_proc_C(prog_counter,data_rec,data_in,data_out);
 END;
```

La fonction *appel_de_proc_C* est la procédure ré–entrante écrite en C. Elle calcule les nouvelles variables du processus (*data_out*) en fonction des valeurs fournies par le simulateur ASA+ (*data_in*) et en fonction de l'état où la procédure avait été stoppée (*prog_counter* et *data_rec*).



```
TRANS
        WHEN result_in.mess(tab_res)
        FROM e2
        PROVIDED blocage(tab_res,num_mod) = nok
        TO e2
BEGIN
END ;
```

Si il y a une condition de blocage sur le processus, la procédure ré–entrante C n'est pas appelée : le processus n'évolue pas.

III.2.3 – La fusion des transitions à un niveau n donné

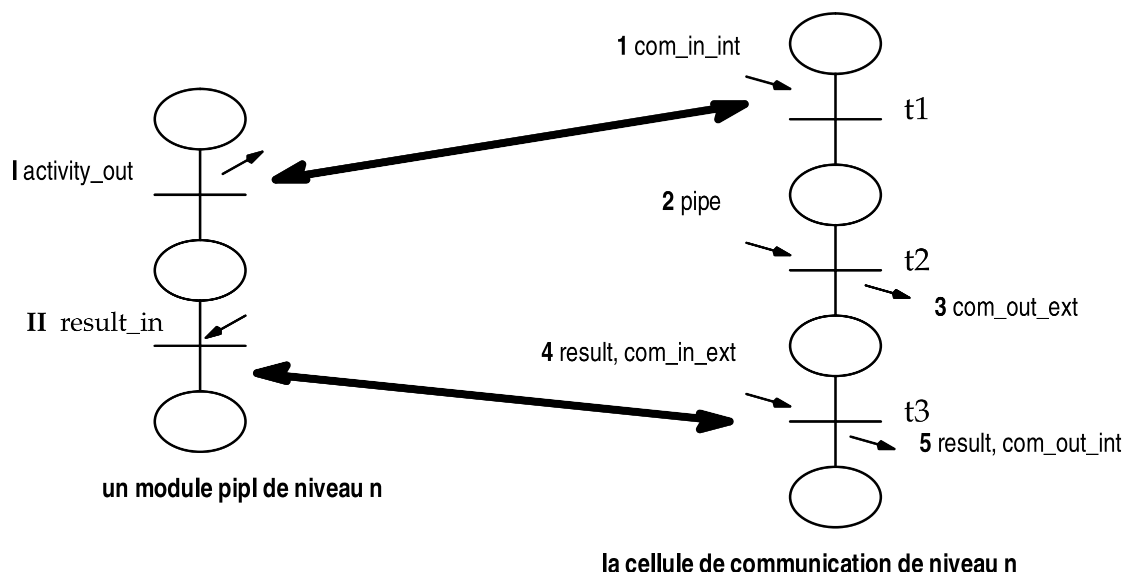


Figure 59 : Fusion de transitions

La fusion des transitions est en fait issue du fonctionnement d'un module de communication au niveau n. Son fonctionnement est développé suivant les cinq étapes suivantes réalisées en seulement trois transitions :

1– Réception des activités de communication de même niveau.

2– Les informations provenant des modules frères du niveau supérieur sont reçues. Elles traitent des canaux externes pour cette cellule de communication. Il s'agit des activités issues des autres modules de communication situés au même niveau structurel (c'est–à–dire les modules de communication de niveau n).

3– On extrait la communication externe depuis les activités fournies en 1 et on introduit ces informations au moyen de la connexion '*pipe*' au niveau supérieur. La cellule de communication de niveau n+1 intègrera ces données pour calculer la communication au niveau supérieur. La communication interne est réalisée suivant la définition du protocole et le résultat est envoyé au niveau supérieur. Ce résultat est concaténé avec les informations reçues en 2.

4– La cellule de communication du niveau supérieur renvoie les résultats sur les activités extérieures.

5– Le résultat du niveau supérieur est complété par le résultat de la communication et transmis aux processus du niveau courant.

Les transitions **2** et **3** et **4** et **5** de la boîte de communication fusionnent avec les niveaux supérieurs. On remarque donc que chaque module de communication transmet les activités des modules internes concernant les activités externes.

Les transitions I et 1 fusionnent ainsi que II et 5 sur le même niveau structurel.

III.2.4 – Un pas de simulation contrôlé

Si nous désignons les deux transitions tirables d'un module *pipl* par I et II et si nous notons chacune des trois transitions possibles d'une boite de communication d'un niveau de hiérarchie j par les couples (j, t1), (j, t2), (j,t3) nous avons les étapes suivantes :

- Dans un premier temps, chacune des transitions I fusionnent avec une transition de niveau (j, t1), cette fusion est notée (I // (j, t1))
- Puis, les transitions (j, t2) et (j+1, t1) fusionnent

Ces deux étapes correspondent à une phase aller où les communications remontent d'un niveau feuille vers le niveau le plus extérieur du circuit. Le nombre de groupe de transitions fusionnées est égal au nombre de niveaux hiérarchiques introduits.

- Une troisième étape correspond à la fusion de toutes les transitions t3 de toutes les boites de communication avec la transition II d'un module terminal soit : II // (1, t3) // ... // (j, t3) // (j+1, t3).

Cette dernière étape correspond à une étape retour où tous les résultats des communications redescendent au niveau des processus.

En conclusion , un pas de simulation VOVHDL correspond à N + 1 étapes de simulation du modèle où N est le niveau de profondeur de la structure hiérarchique.

IV – L'ASPECT ENVIRONNEMENT

IV.1 – Un problème de fermeture d'environnement

Il s'agit en fait de gérer la complexité du modèle. En effet, le nombre de combinaisons possibles des variables d'entrées est déjà prohibitif en vue de l'obtention d'un modèle de taille raisonnable. Le problème qui se pose alors est un problème de fermeture de modèle. Il s'agit de restreindre les combinaisons des entrées possibles en fonction de données réalistes sur le système.

L'environnement relatif à un circuit et son comportement vis–à–vis du mode extérieur doivent être modélisés. Il semble intéressant de définir à ce niveau la communication de la même manière que celle définie à chaque niveau de description : c'est à dire gérée par un module qui contient la définition de la sémantique de communication. Mais persiste alors un problème.

En effet, comment adapter l'interface de communication entre la sémantique de communication des protocoles VOVHDL et la sémantique CCS ? En particulier, certains protocoles VOVHDL induisent des situations de blocage qui ne sont pas compatibles avec le fait que l'environnement CCS soit totalement coopérant.

IV.2 – Le modèle

Afin de répondre à cette difficulté, l'environnement est décomposé en deux modules, l'un permettant l'adaptation entre les ports de communication VOVHDL du circuit et une interface CCS, le second correspondant à la modélisation du comportement proprement dit de l'environnement. Cette modélisation est ainsi facilitée dans la mesure où elle repose sur la sémantique directe du langage de l'outil ASA+.

Cette facilité est notamment appréciée pour l'élaboration de plusieurs cas de figure d'environnement.

En outre, il convient de prendre en compte, dans le cas du composant FICOMP, chacune de ses trois interfaces (partie micro–code, partie utilisateur et partie réseau) par la modélisation de trois environnements pouvant évoluer parallèlement.

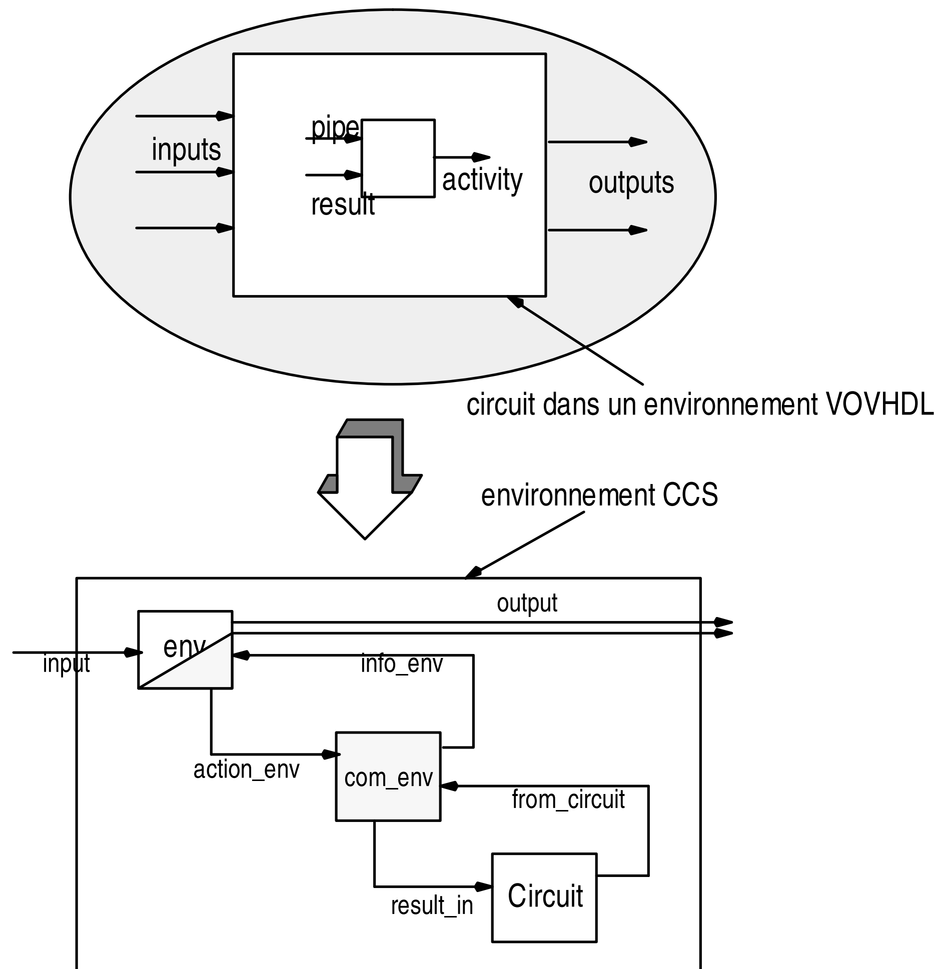


Figure 60 : L'environnement VOVHDL en CCS

IV.2.1 – Le module *com_env*

Ce module est un module spécifique ; en effet, contrairement à ce qui est réalisé dans le circuit, l'environnement et le circuit ne sont pas considérés au même plan. Ceci afin de tenir compte d'un environnement totalement coopérant.

- L'environnement récupère a priori ce que peut faire le circuit et analyse son activité possible.
- L'environnement s'adapte et calcule ce qu'il peut émettre en fonction de l'activité du circuit. En fonction de cette activité, certaines combinaisons des valeurs d'entrées de l'environnement sont possibles ou pas. Le module de communication effectue alors un pré–traitement de l'activité de l'environnement éliminant ainsi des configurations sans signification par rapport à la réalité du composant.

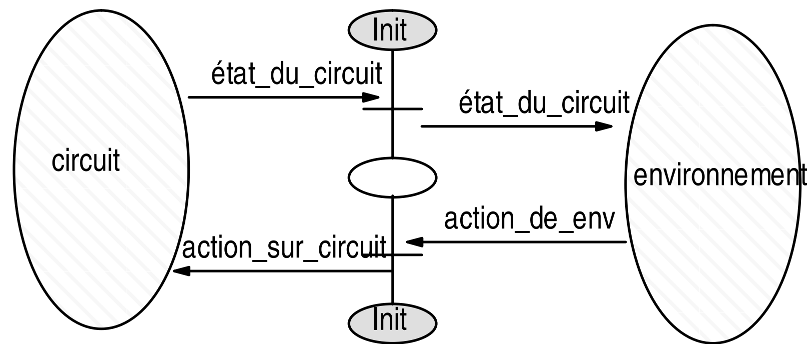


Figure 61 : Les transitions du module com_env

IV.2.2 – Un contrôleur d'environnement.

Le contrôleur permet de traiter le problème d'adaptation à l'environnement CCS. En effet, c'est lui qui sert d'interface de traduction entre les sémantiques VOVHDL et CCS. Il tient compte des éventuelles conditions de blocage induites par un protocole VOVHDL. L'environnement doit donc être récepteur de l'état du circuit et de la génération éventuelle des conditions de blocage. Ces conditions de blocage ne posent un problème que sur les entrées. Le seul protocole de VOVHDL qui puisse entraîner des conditions de blocage en entrée est le Rendez–Vous. Il est donc nécessaire que l'environnement puisse acquérir a priori l'activité du circuit sur une entrée en Rendez–Vous.

Le contrôleur d'environnement autorise alors le tirage d'une transition correspondant à une activité de l'environnement en fonction de l'état du circuit. De plus, un mécanisme de gestion de priorités permet de n'être dans la situation de blocage que si aucune autre situation n'est possible. Soit le circuit est prêt et l'environnement tire une transition correspondant à une activation de l'environnement. Soit le circuit n'est pas prêt (aucune transition de l'environnement ne peut être tirée) alors la transition de priorité la plus faible correspondant à l'inactivité du circuit est tirée.

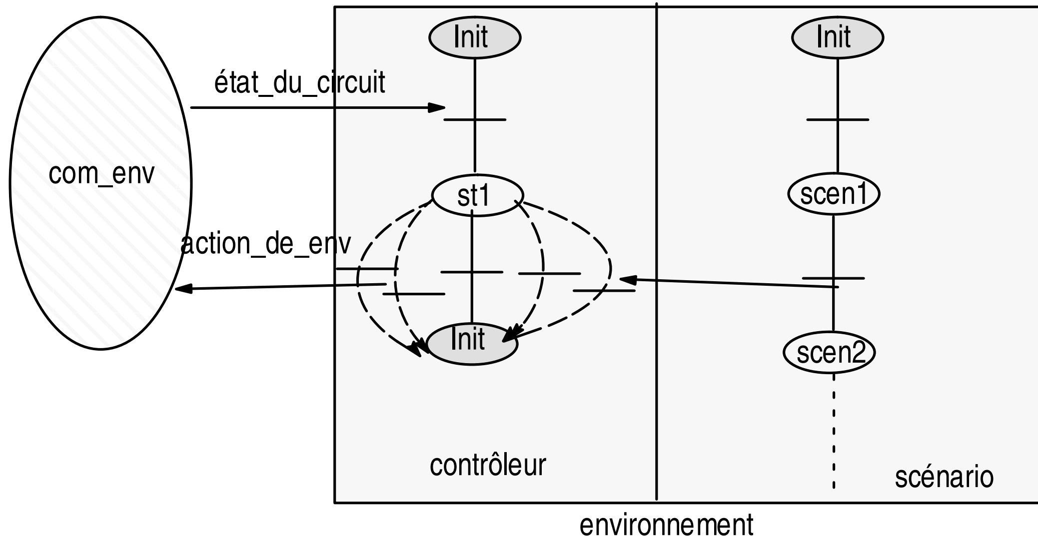


Figure 62 : La décomposition du module environnement

IV.2.3 – Un module scénario

Un deuxième module, connecté à l'environnement contient toutes les transitions des scénarios possibles et permet de dérouler un scénario prédéfini. Chaque fois que le contrôleur active ce module, une action de l'environnement est autorisée, un pas du scénario est déroulé et l'activité est communiquée à la boîte de communication via le contrôleur.

Chacune des transitions de la boucle du contrôleur (c'est à dire l'ensemble des transitions qui permettent de passer de l'état *st1* à l'état *Init*) est fusionnable avec une transition du scénario. Les différentes transitions de la boucle correspondent aux différentes valeurs possibles des messages émis par le scénario. De cette manière, à une étape donnée du scénario, le non–déterminisme est limité. La taille du modèle obtenu peut être ainsi considérablement réduite en éliminant les possibilités peu réalistes.

Leur décomposition en deux sous–modules permet de faire évoluer l'un ou l'autre de manière indépendante : on peut changer un scénario en fonction de ce que l'on veut vérifier, on peut ajouter ou éliminer une transition du contrôleur.

V – L'APPLICATION PRATIQUE

La mise en oeuvre en ASA+ du modèle qui vient d'être exposé a été appliquée au projet FICOMP de la manière suivante :

Le constructeur industriel fournit une spécification abstraite du circuit en précisant les protocoles de communication des ports d'entrées/sorties, complétée par une description comportementale donnée sous forme algorithmique (cf annexe 1).

Cette spécification initiale permet d'écrire la description VOVHDL. Dans le projet FICOMP, cette description a été réalisée manuellement par les chercheurs du Politecnico di Torino.

Un traducteur automatique, réalisé lui aussi à Turin, sur des spécifications que nous leur avons communiquées, produit la partie spécifique du modèle de chaque processus sous la forme d'un programme C.

Ce programme est alors relié avec les modèles de la partie générique du processus et du noyau de communication que nous avons écrits pour constituer le modèle ASA+ complet. C'est ce modèle complet qui est soumis à l'outil de vérification. L'ensemble de la démarche est schématisé par la Figure 63.

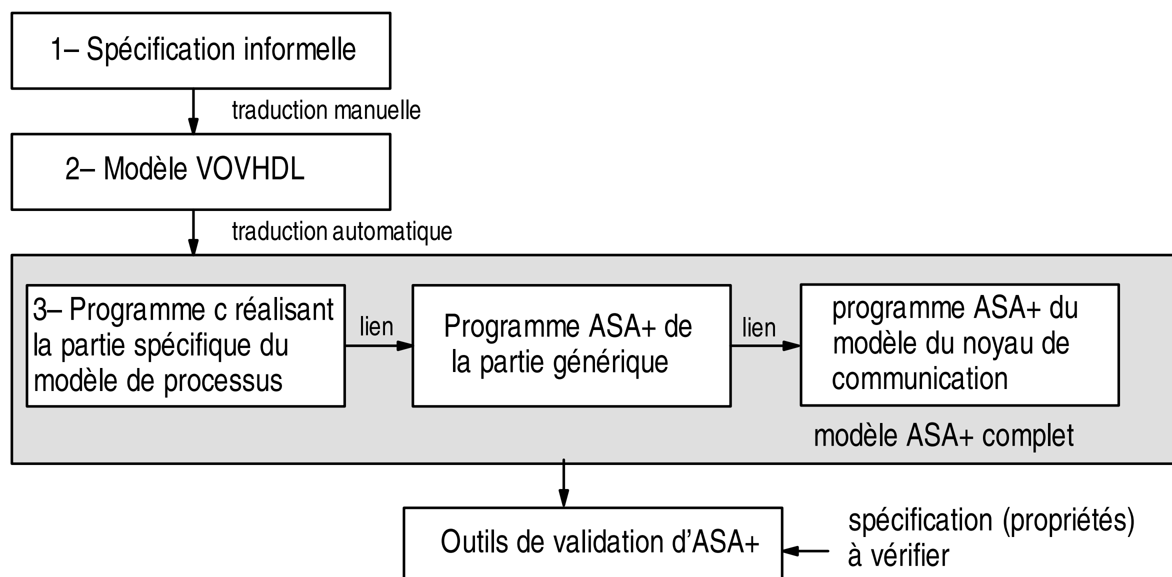


Figure 63 : La démarche de l'application pratique

L'annexe 1 montre sur un exemple simple la forme que prennent les descriptions 1, 2 et 3 d'un module du circuit.

INTERPRETATION DE LA SEMANTIQUE VHDL

CHAPITRE 4

Le but de ce chapitre est de déterminer comment à partir d'une description VHDL, nous pouvons obtenir un modèle basé sur l'algèbre CCS afin d'utiliser des outils de validation et de vérification tels que ASA+.

Dans le domaine de la conception Hardware de circuit, une description au niveau système qui utilise un langage avec une sémantique de communication de haut niveau (telle que celle utilisée par VOVHDL) facilite la modélisation dans un langage orienté Algèbre de processus.

Cependant, à un niveau de description plus bas, on peut être conduit à valider et vérifier une description VHDL.

Nous nous proposons donc d'appliquer à une description VHDL la même méthodologie que celle, présentée dans un chapitre précédent, appliquée au langage VOVHDL. La différence entre ces deux formalismes réside essentiellement dans leur sémantique de communication ; il ne s'agit plus d'une communication au niveau système (primitive de communication) mais d'une sémantique de plus bas niveau (signaux et wait).

Il est alors nécessaire de définir un modèle formel de la sémantique de communication de VHDL. La formalisation de la sémantique VHDL est une préoccupation non triviale qui a orienté des recherches dans des domaines différents. Aussi, dans une première partie, nous rappellerons les champs d'investigation et les groupes d'études menées autour de ce problème.

Dans un deuxième paragraphe, nous rappellerons les concepts de VHDL en insistant plus particulièrement sur les éléments qui caractérisent la communication et la synchronisation ainsi que sur les mécanismes d'exécution d'un modèle de simulation VHDL.

Enfin, une dernière partie présentera la méthodologie de modélisation de ce mécanisme de simulation VHDL en terme de Systèmes à Transitions Etiquetées.

I – INTRODUCTION–POSITIONNEMENT–ETAT DE L'ART

I.1 –Un modèle formel de la sémantique VHDL : un besoin

Bien que VHDL soit utilisé à titre de langage standard dans des domaines très différents, il existe néanmoins un besoin commun de définir formellement la sémantique de ce langage.
Des travaux de plus en plus nombreux concernent ce problème. La définition de concepts mathématiques et rigoureux correspond donc à un besoin qui s'exprime notamment dans les domaines suivants :

- ♦ La fourniture d'une aide à la compréhension du Manuel de Référence du Langage et la localisation de ses éventuelles inconsistances,
- ♦ Une sémantique de VHDL orientée simulation est nécessaire afin de déterminer les fondements des outils de simulation,
- ♦ Une sémantique associée à la synthèse est également nécessaire pour définir les implications Hardware de VHDL,
- ♦ Enfin, la définition d'une sémantique qui permette l'utilisation d'outils de vérification est également indispensable.

Nous présentons ici les principales approches qui nous paraissent émerger en insistant plus particulièrement sur celles orientées Réseaux de Pétri ; cela nous permettra de positionner notre approche.

I.2 – Les différentes approches en fonction de leur motivation

I.2.1 – Les motivations

Ces différentes orientations qui sont autant d'interprétations du Manuel de Référence (LRM) correspondent à deux classes d'approches.

La première consiste à exprimer formellement la sémantique de VHDL. Cette formalisation a pour objectif :

- ♦ soit de pouvoir vérifier directement des propriétés sur le langage. Citons par exemple les travaux de SALEM sur la définition de sémantique formelle de primitives temporelles de VHDL (chap5 [Sa, 92], [BS, 95]) ou la définition d'une sémantique fonctionnelle d'un sous ensemble de VHDL ([BFK, 93], [BFK, 95]),
- ♦ soit de vérifier des propriétés d'un simulateur abstrait du langage, c'est ce qu'envisage [Ru, 95].

Ces approches sont basées sur des formalismes fonctionnels ou dénotationnels et utilisent des démonstrateurs de théorèmes généraux.

La seconde classe correspond à des approches qui ont pour but de traduire des spécifications VHDL dans un formalisme permettant des manipulations formelles au moyen d'outils spécifiques à des fins de vérification de circuit par des techniques d'équivalences de description ou de Model Checking. Ces approches sont fortement guidées par les outils et permettent in fine d'obtenir une machine d'états finie.

Ces deux grandes classes sont loin d'être totalement déconnectées dans la mesure où les approches fonctionnelles peuvent être en fait une formalisation de machine d'états finie et où des démonstrateurs de théorèmes peuvent aussi être employés pour démontrer des équivalences de description et des propriétés logico–temporelles.

Pour clarifier la présentation, les travaux sont regroupés en deux grandes classes et dans chaque classe, les différents travaux sont présentés sous forme d'un tableau comparatif.

Les éléments de comparaisons sont les critères suivants :

- le formalisme utilisé,
- la manière d'interpréter les processus utilisateurs,
- la modélisation du noyau,
- la manière dont sont modélisés les signaux,
- la synchronisation de l'ensemble (représentation des processus et du noyau),
- les restrictions que chaque formalisme impose par rapport au standard VHDL complet,
- le respect ou non du δ cycle VHDL,
- les outils mis en oeuvre au moyen de ce formalisme.

I.2.2 – Les formalismes en vue de l'utilisation d'outils spécifiques

A) Nous avons classé dans un premier groupe les approches qui utilisent le formalisme des réseaux de Petri.

Les travaux du TGI de Madrid [OC, 93a], [OC, 93b] et [OI, 95] présentent les mécanismes de traduction d'une description VHDL sous forme d'un réseau de Petri ; chacun des éléments de la sémantique VHDL (processus utilisateurs, noyau, signaux, fonctions de résolution et liens de communications) sont traduits sous forme d'un réseau de Petri. Le modèle global est obtenu par la composition des RdP de chacun des éléments. La sémantique de communication est implicite et représentée par la fusion des transitions.

De même que les travaux précédents,[DDH&, 93], [DJS, 93] et [DJS, 95] ont pour objectifs de définir une sémantique formelle de VHDL en vue de développer un environnement de vérification dans le cadre du projet ESPRIT FORMAT 6128. Aussi, ces travaux présentent une méthode pour traduire de manière automatique une description VHDL en un modèle d'états fini [DHP, 95]. A partir de ce modèle, des outils de vérification automatique basés sur les BDD peuvent être utilisés ainsi que des outils de preuve interactives.

[En, 94] définit également chacun des éléments de la sémantique de VHDL sous forme de réseaux de Petri. Des techniques de manipulation de BDD seront appliquées pour obtenir une machine d'états utilisable par des outils de vérification.

B) Une approche FSM

Un second groupe comprenant les travaux de BULL [DO,93] et de l'université de Grenoble ([DB, 95a] et [DB, 95b]) se caractérise par une approche FSM. Dans les deux cas, les descriptions sont synchronisées par une horloge mère. Cela impose de fortes restrictions aux descriptions pour garantir que toutes les variables d'état soient chargées sur front d'horloge.

L'approche de BULL a pour objectif de prouver l'équivalence de deux architectures pour une même description avec l'aide d'un outil totalement automatique basé sur les BDD. A ce jour, seulement l'outil permettant de comparer deux machines ayant les mêmes états est commercialisé.

L'approche de l'université de Grenoble consiste à déduire d'une description VHDL une machine abstraite. Cette machine est obtenue par composition de machines abstraites issues des processus et par distribution de la fonctionnalité du noyau sur ces machines abstraites. Des techniques d'équivalence et de model checking sont alors utilisables sur la machine abstraite ainsi obtenue.

C) Les langages synchrones

Qu'il s'agisse du langage SIGNAL ([Be, 94] et [BCG, 93]), LUSTRE ou FOCUS ([FM, 95]), l'objectif est de définir des règles de traduction automatique d'une description VHDL dans le langage synchrone concerné afin d'utiliser les outils de vérification basés sur ce même langage.

D) Les Algèbres de Processus

Afin de permettre l'utilisation d'outils basés sur la sémantique des Evolving Algebra (EA), [BGM, 94] présentent une interprétation de la sémantique VHDL. Les processus et le noyau sont modélisés selon la sémantique des EA et les liens de communication sont également définis.

De notre côté, nous interprétons la sémantique de VHDL en termes de comportement et de communication mais en vue de l'utilisation d'un outil de vérification basé sur la sémantique CCS ([BSBCP,94], [CCPBS,93b]).

approches / critères	RdP	FSM	Langages Synchrones	Notre approche
Le formalisme	**Réseau de Petri (RdP)** **Evolving Algebra (EA)**	**Machine de Mealy**	**Lustre** **Signal** **Focus**	**Système à Transition Etiquetées et Multi–rendez–vous**
L'interprétation du processus	**instruction par instruction**	**Bull : mise sous forme canonique** **Grenoble : mise sous forme canonique puis instruction par instruction**	**Instruction par instruction**	**Mise sous forme canonique et extraction du comportement en termes de communication (approche globale et observationnelle)**
La modélisation du noyau	**Olcoz : explicite** **Encrenaz: explicite** **Damn : implicite**	**Implicite : produit cartésien des machines d'états qui reflète le comportement du noyau**	**Implicite : composition du formalisme**	**Explicite**
La réalisation de la synchronisation	**Par place partagée et simplification du graphe**	**Produit cartésien des FSM**	**Composition du formalisme**	**Multi–rendez–vous isomorphe à la fusion des transitions**

approches / critères	RdP	FSM	Langages Synchrones	notre Approche
La modélisation des signaux	**Olcoz : Un RdP par pilote Encrenaz : un RdP par pilote**	**De 0 à 2 variables d'état par pilote et synchronisation par ces variables**	**Lustre : un signal par signal et un process par attribut**	**Variables globales partagées entre processus et noyau**
Les restrictions par rapport au standard *(*) (**)*	**Encrenaz : pas de clause AFTER (uniquement des 0 delay)**	**Pas de clause AFTER** **Synchronisation sur front d'horloge**	**signal : 1 wait par processus** **Lustre : pas de delay inertiel**	*(*) (**)*
Atomicité, respect du δ cycle	**OUI**	**NON**	**OUI**	**OUI pour le formalisme NON pour outil**
Les outils	**Model Checking et équivalence**	**Model Checking et équivalence**	**Model checking et équivalence** **Signal : équivalence observationnelle**	**Model checking, équivalence, projection et test**

(*) restrictions communes aux approches de ce tableau : uniquement les types finis sont considérés et pas de généricité

(**) restrictions communes à toutes les approches : pas de type réél, pas de type acces, pas de fichier, pas de variable globale, pas de type physique autre que TIME.

I.2.3 – Les formalismes en vue de l'utilisation de démonstrateurs de théorèmes généraux

Ces formalismes sont regroupés en trois types d'approches :

Tout d'abord ceux qui expriment la sémantique de VHDL en utilisant une logique d'ordre 1. Citons les travaux de [Sa, 92] et de [Ru, 95].

Puis viennent ceux qui travaillent avec la logique d'ordre supérieur.

- ♦ L'approche Flow Graph (FG) de [RK, 95] a pour objectif de définir un formalisme qui permette de faire le lien entre une description VHDL et la logique d'ordre supérieur afin d'obtenir un procédé de vérification automatique d'une description VHDL. Le démonstrateur utilisé est HOL.
- ♦ Van Tassel a également pour objectif l'utilisation du démonstrateur HOL et c'est la raison pour laquelle il s'est penché sur l'étude d'une sémantique opérationnelle d'un sous–ensemble de VHDL [VT, 95].
- ♦ Beth Levy se place également au niveau de la logique d'ordre supérieur mais en vue de l'utilisation d'un autre démonstrateur : "SVDS" [LFMM, 92].

La dernière approche concernera la sémantique dénotationnelle. Un début de formalisation de la sémantique du noyau VHDL a été présenté par [Da, 93]. mais des travaux plus complets ont été abordés ; [BS, 95] a présenté la définition dénotationnelle d'un sous–ensemble synchronisé par une horloge de VHDL et [BFK, 95] présente également une définition dénotationnelle et une logique de programmation de Hoare d'un sous–ensemble de VHDL.

approches / critères	Logique d'ordre 1	Logique d'ordre supérieur	Approche dénotationnelle
Le formalisme	**Fonction**	**Fonctionnelle**	**Fonctionnelle**
L'interprétation du processus	**Fonction**	**Relations d'ordre supérieur**	**Fonctionnelle**
La modélisation du noyau	**Fonction**	**Relations d'ordre supérieur**	**Fonction**
La réalisation de la synchronisation	**Implicite**	**Implicite**	**Implicite**

approches / critères	Logique d'ordre 1	logique d'ordre supérieur	Approche dénotationnelle
La modélisation des signaux	**Fonction**	**Fonction du temps**	**Fonction**
Les restrictions par rapport au standard (**)	**Russinoff : transport Salem: uniquement 0 delay**	**μ VHDL de Van Tassel**	**Salem: pas de delay different de 0 + synchronisation par horloge et delay transport Delgadok Kloos : pas de δ delay**
Atomicité, respect du δ cycle	**Oui**	**Oui**	**Oui**
Les outils	**Démonstrateurs de théorème et techniques inductives Russinoff : simulateur formel de VHDL en B&M**	**Démonstrateurs de théorèmes**	**Partiellement démonstrateurs de théorèmes Delgado Kloos : moteur formel pour prouver des propriétés**

(**) restrictions communes à toutes les approches : pas de type réel, pas de type acces, pas de fichier, pas de variable globale, pas de type physique autre que TIME

II – LA SÉMANTIQUE DE COMMUNICATION VHDL

II.1 – Une brève présentation de VHDL

II.1.1 – Les caractéristiques de VHDL

Un langage de description structurelle multi–niveaux et hiérarchique

VHDL permet au concepteur de décrire un composant de manière hiérarchisée. En effet, il est possible de décliner la description fonctionnelle de haut niveau d'une entité en une ou plusieurs descriptions détaillées de plus bas niveau. Une même entité avec une même interface peut posséder différents niveaux de description. Ces niveaux peuvent aller d'une conception au niveau système (micro–processeurs) jusqu'à un niveau de description plus détaillé (niveau porte logique).

La modularité – les unités de conceptions

Une unité de conception VHDL peut être vue tantôt comme une boîte noire (on ne décrit que les ports et les interactions de cette boîte avec l'extérieur), tantôt comme une boîte blanche (on précise le fonctionnement interne de la boîte).
Il existe cinq unités de conception qui sont classées en unités de types primaires et secondaires. Les unités primaires décrivent ce que fait l'unité de conception. Il s'agit des spécifications de paquetage, des spécifications d'entités et des configurations. Les unités de type secondaire (les corps de paquetage et les corps d'architecture) décrivent comment est fait l'intérieur de la boite entité.

En fait, une unité de description VHDL est constituée de deux éléments : l'*interface* et le *corps* de la description. L'interface définit les variables de communication avec l'environnement extérieur et les paramètres génériques qui régissent la structure et le fonctionnement de l'unité de description. Les valeurs données à ces paramètres (instances) permettent d'utiliser l'unité sous plusieurs formes. Le corps décrit l'organisation interne et/ou les opérations de l'unité.
Une spécification de paquetage contient les programmes, types et objets manipulés par le corps de paquetage sans entrer dans les détails de la réalisation. L'algorithmique sera contenue dans le corps de paquetage.
De manière analogue, une spécification d'entité constitue l'interface de la boîte entité et le corps est donné par une ou plusieurs architectures.
Une architecture peut être de trois types (ou une combinaison de ces trois types) : structurelle, flot de données ou comportementale. Le bloc est un élément de structuration fondamental de l'architecture : il permet de hiérarchiser la description. (cf II.2.3)

Une configuration est liée à la notion de composant (voir plus loin le paragraphe composant et blocs). Lorsqu'une architecture est décrite sous forme structurelle, la configuration exprime, pour chaque exemplaire du composant de la structure, quel couple (entité/architecture) lui est associé.

L'héritage d'ADA.

Une originalité de VHDL réside dans l'utilisation d'instructions inspirées de celles du langage de programmation ADA pour décrire le comportement des entités sous forme de description algorithmique. Aussi, VHDL conserve les caractéristiques suivantes et permet :

- d'effectuer des descriptions à un haut niveau d'abstraction, en particulier au niveau spécification système,
- de définir de nouveaux types abstraits,
- de créer des modèles abstraits (entité/spécification) et de définir plusieurs instances pour un même modèle,

- de définir des paquetages : unités de bibliothèques qui contiennent des éléments pouvant être utilisés dans d'autres unités de conception,
- de modéliser des tâches concurrentes.

II.1.2 – Les objets manipulés

Nous rappelons brièvement la syntaxe et la fonctionnalité des quatre classes d'objets manipulés en VHDL : les constantes, les variables, les fichiers et les signaux.

constantes

Une constante est définie par son mot clé suivi de son nom , de son type et de la valeur de la constante (celle–ci est une expression statique). Une constante est calculée lors de l'élaboration et peut être une expression complexe (incluant des appels de fonctions). On peut également définir des constantes à valeurs différées : la constante est déclarée dans une spécification de paquetage et sa valeur n'est pas définie. Cela permet d'utiliser la constante de manière abstraite. C'est un moyen de masquer l'information. Le type de la constante peut être de tout type ou sous–type prédéfini ou déclaré à l'exception du type pointeur ou de tout type complexe ayant un type pointeur parmi ses éléments.

constant nom_de_cte : type_ou_sous_type {:= valeur_de_cte};

variables

C'est un objet lié à l'algorithmique et à la séquentialité. Une variable peut être modifiée au cours de la simulation et la valeur présente (contrairement aux signaux) peut également être modifiée (par affectation). Une variable est définie de manière similaire à la constante par son mot clé, son nom, son type ou sous–type et éventuellement sa valeur d'initialisation. Tous les types sont autorisés y compris le type pointeur. Une valeur d'initialisation peut ne pas être donnée explicitement : toutes les variables ont une valeur d'initialisation par défaut. La valeur d'initialisation d'une variable peut être une expression. Dans ce cas la valeur initiale de la variable est déterminée à chaque élaboration. Le domaine d'utilisation des variables est restreint aux sous–programmes et aux processus.

variable nom_de_var : type_ou_sous_type {:= valeur_d'init};

fichiers

Un fichier est en fait une variable VHDL sans possibilité d'affectation. Si des processus peuvent lire un fichier mais pas le modifier par une écriture, celui–ci est alors en mode **in**. Inversement, un fichier accessible en écriture mais pas en lecture est de mode **out**. Le nom logique du fichier est le nom sous lequel il est connu de l'environnement.

file nom_de_fic : sous_type_de_fic **is** (mode) nom_logique_de_fic;

signaux

Les signaux sont spécifiques à la description de matériel. Ils servent à modéliser les informations qui passent sur les fils, les bus ou, d'une manière générale, qui transitent entre les différents composants d'une description de matériel. Cet objet assure donc la communication (cf II.2.1). Un signal est statique (rémanent). Il existe indépendamment de sa zone de visibilité.

Un signal est *résolu* s'il possède une fonction de résolution dans sa déclaration. C'est un signal qui peut avoir plusieurs sources et donc être soumis à des affectations concurrentes. La fonction de résolution résout le conflit possible et calcule la valeur du signal en utilisant toutes les valeurs des sources (à l'exception de celles qui sont déconnectées).

Une affectation de signal est dite *gardée* si cette affectation s'effectue sous le contrôle d'une expression booléenne. Les deux mots clés "bus"ou "register" qui peuvent figurer dans la déclaration du signal déterminent deux types de signaux gardés. Cette distinction est en fait liée à la description de matériel et

n'est visible que lorsque tous les signaux sont déconnectés : un signal de type registre gardera la dernière valeur qui lui a été affectée alors qu'un signal de type bus prendra une valeur prédéfinie.

Un signal gardé ne peut être affecté que sous le contrôle d'un signal booléen GUARD. Si la valeur du signal GUARD vaut "True" le signal gardé peut être affecté sinon il est déconnecté. Les blocs gardés (cf II.2.3) permettent également de garder certaines affectations de signaux.

signal nom_du_sig : {fct_reso} type_ou_sous_type {contraintes} {**register/bus**}{:= valeur_init};

II.2 – Les éléments de communication et de synchronisation

II.2.1 –Signaux et pilotes : éléments de communication

La communication entre blocs, composants et processus s'effectue en VHDL au moyen d'objets de type signaux. Ceux–ci sont caractérisés non seulement par une variable représentant la valeur du signal à un moment donné mais encore par une suite de couples (instant, valeur) qui symbolise les états futurs du signal. Un tel couple est appelé transaction ou encore élément d'onde. Cette suite est appelée pilote du signal.

Les signaux sont des objets ayant un historique de valeurs. Chaque signal possède un pilote par processus où il est affecté. Un même signal peut donc posséder plusieurs pilotes. Il doit alors être résolu.

1– Les différentes affectations de signaux

Les valeurs d'un signal évoluent suivant son affectation. Une affectation de signal ne change pas la valeur présente d'un signal mais sa ou ses valeurs futures (son pilote). Il existe différents modes d'affectation, de type séquentiel ou concurrent :

Affectation par forme d'onde

C'est le cas de l'affectation suivante où A prendra la valeur de B dans 10 ns, puis '1' dans 20 ns (10+10), puis le résultat de l'expression "D AND C" dans 50 ns (20+30).

A<= B **after** 10 ns, '1' **after** 20 ns, D AND C **after** 50 ns

La valeur de la transaction peut être une constante, la valeur présente d'un signal ou le résultat d'une expression. Une affectation par forme d'onde peut être séquentielle ou concurrente. L'effet est le même lorsqu'il y a une seule affectation. Dans le cas où deux affectations à un même signal sont exécutées au cours du même cycle de simulation, la différence est la suivante :

- Dans le cas séquentiel, (à l'intérieur d'un même processus), la deuxième affectation modifie le même pilote que la première et peut donc détruire des transactions créées par la première. C'est le cas de l'exemple suivant :

```
Process
        Begin
                ...
                A<= B after 10 ns, '1' after 20 ns, D AND C after 50 ns;
                — traitement sequentiel sans wait—
                A<= C after 10 ns, '0' after 20 ns;
                wait on S;
        end process
```

- Dans le cas concurrent, les deux affectations constituent des processus distincts, chacune modifie son propre pilote et le signal doit être résolu. C'est le cas de l'exemple suivant où les deux processus affectent au même pas de simulation le signal A qui doit être un signal résolu.

```
P1 : Process
        ...
        Begin
                A<= B after 10 ns, '1' after 20 ns, D AND C after 50 ns;
                wait on S;
        end process;
P2 : Process
        ...
        Begin
                A<= C after 10 ns, '0' after 20 ns;
                wait on P;
        end process;
```

affectation à délai "delta"

Il existe une affectation particulière : dite "affectation à délai delta". C'est celle qui affecte un signal à un délai nul. Son écriture est la suivante :

A<=B **after** 0 ns ou plus simplement A<=B

Elle signifie que le signal A prend la valeur présente du signal B après un délai "delta". Un délai delta est un délai nul pour la simulation. Un délai nul n'a pas de signification du point de vue du concepteur, mais par contre, il permet de garantir le respect de la causalité au niveau de la simulation.

Deux modes de transmission

Deux modes de transmission de signaux peuvent être pris en compte : le mode TRANSPORT et le mode INERTIEL. Le mode TRANSPORT permet la transmission de toute impulsion de signal et ceci quelle que soit sa durée. Par contre, le mode INERTIEL, qui est le mode pris par défaut à chaque affectation, filtre les impulsions dont la durée est inférieure au temps de transmission.

Par exemple, signal1 <= REF **after** 10 ns : le mode inertiel est pris par défaut et toute impulsion sur REF de durée inférieure à 10 ns ne sera pas transmise à signal1. Si l'on désire que toute impulsion de REF soit répercutée sur signal1, il faut utiliser la syntaxe suivante : signal1<= **transport** REF **after** 10 ns.

Affectation concurrente conditionnelle

Elle peut prendre deux formes : la forme alternative ou la forme sélective. A chacune de ces formes correspond une instruction conditionnelle séquentielle contenue dans le processus équivalent.

Forme alternative : La transformation du signal dans le processus équivalent correspond à une instruction IF :

```
Signal <= { guarded} {transport}
        forme_onde1 when condition_bool1 else
        forme_onde2 when condition_bool2 else...
        ....
        forme_ondeN;
```

Forme sélective : cela correspond à une instruction CASE dans le processus équivalent :

```
with exp select
Signal <= { guarded} {transport}
        forme_onde1 when choix1;
        forme_onde2 when choix2;
        ...
        forme_ondeN when choixN;
```

2– Signaux explicites et implicites

Déclaration de signal

Le terme signal fait référence à un objet défini soit par déclaration explicite, soit par déclaration de ports. Il peut être défini d'une des manières suivantes :

entity XYZ (a,b : **in** Bit; c : **out** bit);
signal R, S, T : bit;
Component and_gate **port** (a,b : **in** bit; c : **out** bit);
...

Il existe d'autres types de signaux : les signaux implicites. Il peuvent provenir soit d'une condition de garde, soit d'un attribut.

Les signaux implicites définis sur attributs de signaux

L'attribut est défini comme une caractéristique associée à un type ou à un objet. Ici, notre propos ne concerne que les attributs de signaux. Un certain nombre d'attribut est prédéfini en VHDL mais le concepteur peut lui même déclarer et spécifier ses propres attributs. Un attribut est caractérisé par son préfixe (le signal S sur lequel porte l'attribut) et ses paramètres (en général temporels). Ceux–ci permettent le calcul de l'attribut.

Les attributs prédéfinis en VHDL peuvent être subdivisés en deux groupes : ceux dont le résultat est un signal et ceux pour lesquels c'est une fonction. Les attributs signaux sont S'DELAYED [(T)], S'STABLE [(T)], S'QUIET [(T)], S'TRANSACTION. Les attributs fonctions S'EVENT, S'ACTIVE, S'LAST_ACTIVE, S'LAST_EVENT, S'LAST_VALUE.

Le concepteur peut définir un attribut sur un signal mais le résultat sera uniquement de type constante.

Toute référence à un attribut DELAYED, STABLE, QUIET ou TRANSACTION crée, pour chaque signal auquel il est appliqué et pour chaque valeur de paramètre un signal implicite. Ils sont créés lors de la phase d'élaboration.

Par exemple, pour un signal explicite déclaré S, il y a lors de l'élaboration, création d'un signal implicite pour chaque combinaison d'attribut de signal et de valeur du paramètre.
Aussi, pour les écritures suivantes :

S'Delayed(4 ns),
S'Stable(4 ns),
S'Delayed(5 ns), il y aura création de trois signaux implicites

Les signaux implicites créés par les conditions de garde

Ce sont les signaux GUARD.

Par exemple, considérons un bloc gardé défini de la manière suivante :

Bloc1 : **block** (clock ='1' **and not** Clock'stable)
begin
S1<= **guarded** DATA1 **after** 10 ns;
S2<= **guarded** DATA2 **after** 15 ns;
end block BLOC1;

En fait au cours de l'élaboration un signal implicite GUARD de type booléen est défini par la condition booléenne (clock ='1' **and** not Clock'stable). Ce signal est local au bloc et il vaut 'true' si la condition est vrai et 'false' sinon. Les affectations de S1 et de S2 ne s'effectueront que si la condition est vraie.

3– Les pilotes

Un pilote est créé par signal et par processus. Il sert à mémoriser la valeur courante du signal et les valeurs futures prévues, stockées dans des transactions. Les transactions sont ordonnées suivant leur composante temporelle. La valeur initiale d'un pilote est déterminée par la valeur par défaut du signal auquel il est associé. Une affectation ne modifie jamais la valeur courante mais seulement les valeurs futures.

Chaque fois que le signal acquiert une nouvelle valeur au cours de l'avancement du temps, une transaction est effacée. Le pilote est alors dit actif. Illustrons cette définition par l'exemple suivant :

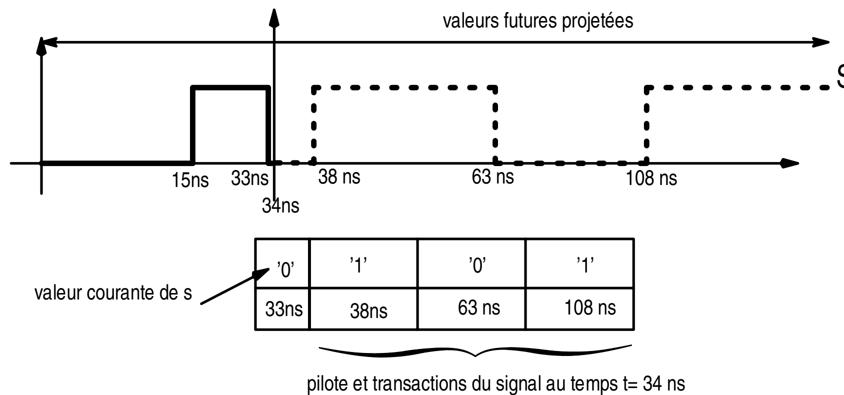


Figure 109 : Le pilote d'un signal S au temps de simulation 34ns

La figure1 nous montre l'état du signal S et l'état de son pilote au temps t = 34 ns. Puis la simulation avance et nous montrons dans la figure 2 le nouveau pilote au temps t =38 ns : une transaction a été effacée.

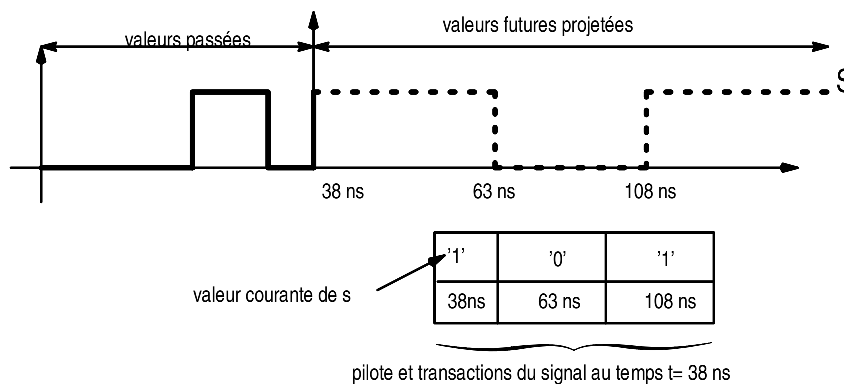


Figure 110 : L'état du signal S au temps t=38ns

4– Valeurs courantes, effective et pilotées d'un signal, signal actif

Il est nécessaire de préciser les termes valeur courante (current value) d'un pilote, valeur effective (effective value) et pilotée(s) (driving values) d'un signal.

Pour tout pilote, il existe une transaction dont la composante temporelle est inférieure ou égale au temps de simulation courant. La *valeur courante* d'un pilote est la composante valeur de cette transaction.

La *valeur pilotée* d'un signal S est la valeur que fournit le signal comme source aux autres signaux. Elle est définie de la manière suivante :

Si S n'a pas de source, sa valeur pilotée est la valeur par défaut de S.

Si S a une source qui est un pilote, et si S n'est pas résolu, alors la valeur pilotée de S est la valeur de ce pilote.

Si S a une source qui est un port et si S n'est pas résolu, alors la valeur pilotée de S est celle associée à ce port. Le calcul de cette valeur associée est donnée dans la partie 4.3.2.2 du Manuel de Référence du Langage.

Si S est résolu, alors les valeurs pilotées des sources de S sont examinées, et si il n 'y a pas de cas d'erreur alors la valeur pilotée de S est obtenue par l'exécution de la fonction de résolution associée. Les paramètres d'entrée de cette fonction de résolution sont constitués de toutes les valeurs pilotées des sources du signal S à l'exception de celles de transactions nulles.

Une transaction nulle est une transaction particulière dont la syntaxe est **null** [time–expression] et qui signifie que le pilote concerné est clos de telle manière qu'il stoppe sa contribution (au moins temporairement) pour le calcul de la valeur du signal cible.

La *valeur effective* d'un signal est en fait identique à la valeur pilotée sauf pour les ports connectés en mode in ou inout. Dans ce cas, il peut y avoir des fonctions de conversion. La valeur pilotée est transformée par ce type de fonction pour obtenir la valeur effective.

Activité d'un signal

En début de cycle de simulation, lorsque le temps courant est déterminé, si le temps courant est égal à la composante temporelle de la deuxième transaction du pilote, le signal est dit *actif* et la première transaction est effacée. La deuxième transaction devient la première et la composante valeur devient la nouvelle valeur courante du pilote La valeur effective du signal doit être recalculée.

II.2.2 – la synchronisation

1– la notion de processus

Un processus est une description comportementale constituée d'une partie déclaration et d'un programme d'instructions séquentielles. Il réagit à certains signaux devenus actifs par l'occurrence d'événements sur ces signaux. Il existe deux sortes de processus en VHDL : ceux qui sont définis à l'aide d'une liste de sensibilité et ceux qui contiennent des instructions WAIT. Une liste de sensibilité est une liste de signaux définie juste après le mot clé Process. Un événement sur un signal correspond au fait que celui–ci a changé de valeur. Dès qu'un événement survient sur l'un des signaux de la liste, le processus est réveillé. Tout processus avec liste de sensibilité est équivalent à un processus contenant une instruction WAIT en dernière instruction.

Un processus est constitué d'une suite d'instructions séquentielles dont certaines (les instructions de type wait) jouent un rôle particulier. Un processus s'exécute jusqu'à rencontrer une instruction wait. Il est alors suspendu jusqu'à son réveil (déterminé par le simulateur) puis reprend son exécution jusqu'à une nouvelle suspension (la rencontre d'une nouvelle instruction wait). Un processus s'exécute de manière cyclique : lorsqu'il a parcouru toutes les exécutions séquentielles, il reprend son exécution à partir du mot clé begin.

```
{label}:Process
        ...Déclarations
        ...
begin
        ...Instructions séquentielles
        wait {on liste de signaux} {until cond–bool} {for temps}
        ...Instructions séquentielles
end process {label};
```

Figure 111 : Processus VHDL

Processus actif– processus passif

Un processus peut être réveillé si un événement se produit sur au moins un des signaux de la liste de signaux sur laquelle il était en attente. Au cours de son exécution il peut modifier des signaux, créant ainsi des événements sur des signaux et donc des conditions de réveil pour d'autres processus. Un tel processus qui possède des moyens d'action sur d'autres processus est dit actif.
Contrairement aux processus actifs, ceux dont l'exécution n'entraîne pas l'exécution d'autres processus sont dits passifs. Ils ne modifient pas de signaux et sont en général des processus d'observation.

Processus Post–synchronisés

Un processus post–synchronisé est nécessairement passif ; il représente en fait un processus d'observation après stabilisation. Il n'est exécuté qu'une seule fois dans chaque unité de temps, et uniquement s'il a été réveillé après le calcul de tous les signaux.

2 – L'instruction Wait : élément de synchronisation

Tous les processus s'exécutent parallèlement durant la simulation du système complet de manière indépendante les uns des autres (sans échange d'information). La synchronisation est alors assurée par l'instruction séquentielle Wait. C'est lorsque tous les processus ont atteint une instruction Wait qu'ont lieu l'échange d'informations (concernant l'évolution des signaux).

Nous rappelons ici les différents types d'instruction Wait.

Wait on (liste_signaux) : un processus, suspendu sur cette instruction, est réveillé dès qu'un des signaux de la liste liste_signaux change de valeur. En fait, dès qu'il y a un événement sur l'un des signaux de liste–signaux.

Wait until (condition_booléenne) : le processus est suspendu jusqu'à ce que la condition spécifiée (condition_booléenne) soit satisfaite. Cette condition est évaluée chaque fois qu'un signal impliqué dans l'expression de la condition est modifié.

Wait for (time_expression) : dans ce cas un processus est suspendu pendant un temps de durée time_expression.

Cependant, une instruction wait peut être constituée de la combinaison d'une ou plusieurs des instructions précédentes. Dans ce cas chaque fois qu'un événement arrive sur un des signaux de la liste *liste_signaux*, la condition booléenne *condition–booléenne* est évaluée. Si la condition est vérifiée alors le processus reprend son exécution, sinon il reste suspendu jusqu'à ce qu'un nouvel événement sur un signal provoque une nouvelle évaluation de la condition ou bien jusqu'à ce que le temps défini par *time_expression* se soit écoulé. Une telle instruction s'écrit de manière générale :

Wait on (liste_signaux) until (condition_booléenne) for (time_expression)

II.2.3 – La notion de blocs et de composants

Blocs et composants sont deux moyens de hiérarchiser une description structurelle en VHDL. Un bloc peut être vu comme un composant instancié. Le composant, bien plus lourd à mettre en oeuvre que le bloc présente cependant l'avantage d'être réutilisable. Le choix d'utiliser des blocs ou des composants, dépendra des besoins du concepteur. Si ce dernier a besoin de concision ou si sa conception s'oriente vers une modélisation alors l'utilisation des blocs est préférable. Si une description d'ordre plus général est souhaitée alors le composant est adéquat.

composant

Le composant sert à définir une vue externe d'un objet. Il servira à créer ultérieurement un composant réel (au moyen d'une instanciation). Un composant détermine le nom et les paramètres génériques de l'objet ainsi que les ports formels qui le connectent à son environnement.

```
component nom_du_composant
        {generic (description des paramètres génériques);}
        {port (description_des_ports);}
end component;
```

Figure 112 : structure composant

La correspondance entre le composant et son modèle effectif est réalisée dans la déclaration d'une unité de conception de type configuration ou dans la spécification de configuration.

blocs

Outre le fait qu'un bloc soit un support de hiérarchie et qu'il permette de factoriser des déclarations locales visibles depuis les instructions concurrentes qu'il contient, le bloc a la possibilité de conditionner certaines affectations de signaux : un tel bloc est un bloc gardé.

```
label: block {(condition de garde)}
        {valeur_importées_de_l'environnement}
        ...
        déclaration locales
        ...
begin
        ...
instructions concurrentes
        ...
end block {label};
```

Figure 113 : structure bloc

Un bloc gardé est un moyen utile pour décrire les parties synchrones d'un circuit. Toutes les affectations de signaux sont soumises à une même condition (condition d'horloge). Chaque affectation devrait s'écrire sous forme d'affectation concurrente conditionnelle de signaux telle que :

A<= expr WHEN (clock= '1' and not clock'stable).

Pour alléger l'écriture la condition est factorisée : c'est la garde du bloc.

```
block (clock='1' and not clock'stable)
        begin
                S1 <= guarded data1 after 10 ns;
                S2 <= guarded data2 after 30 ns;
                ...
        end block
```

Cependant, blocs et composants peuvent être transformés en leur processus équivalent : c'est d'ailleurs ce que fait la phase d'élaboration.

II.2.4 – le temps VHDL

Un système VHDL évolue en fonction des synchronisations de processus et notamment en fonction des instructions wait. La sémantique de communication est une sémantique dirigée par événement. C'est l'arrivée d'un événement sur le système (une nouvelle affectation de signal, un réveil de processus) qui le fait évoluer. VHDL possède néanmoins une notion de temps VHDL. Selon deux dimensions ,

- Le temps physique se compte en unité de type TIME. C'est le temps vu par le concepteur;
- Le temps "delta" qui se compte en cycle de simulation, est géré par le simulateur. Ce temps reflète alors la causalité. D'un cycle à l'autre, le temps physique évolue ou non.

Définition : le délai "delta" (delta delay)

En l'absence de retard explicite dans une affectation de signal, la transmission de la valeur de l'expression source au signal cible prend un "delta". Le retard "delta" correspond en fait à un cycle de simulation du modèle. A chaque unité de temps physique simulée, le simulateur peut exécuter un nombre variables de "delta" cycles pour stabiliser le modèle. Le retard "delta" peut être vu comme une tranche de temps infinitésimale (mais de durée non nulle), telle que le cumul de délais "delta"n'atteint jamais ne serait ce qu'une femtoseconde (la plus petite unité de temps).

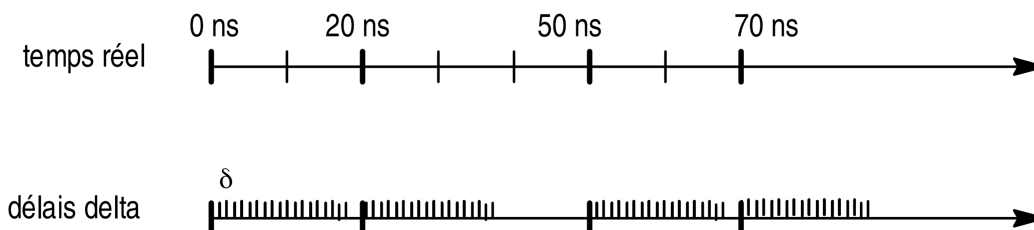


Figure 114 : Le temps VHDL

Définition de TIME'HIGH

La valeur TIME'HIGH est une valeur qui dépend de l'implantation et qui est utilisée pour arrêter la simulation. L'attribut HIGH est un attribut sur type. Le résultat de TIME'HIGH est une valeur de type TIME. C'est la plus grande valeur de ce type implantée. Le type TIME est un type prédéfini dans le paquetage STANDARD. L'intervalle de ses valeurs autorisées dépend de l'implantation; il doit correspondre au minimum à un codage de 32 bits. L'unité de base du type TIME est la femtoseconde.

II.3 – La sémantique de simulation

Le concepteur a la possibilité de modéliser un circuit de manière concurrente et distribuée [CCB, 91].

Le traitement d'une description VHDL est constitué de trois phases : la phase de compilation, la phase d'élaboration et la phase d'exécution. Cette dernière pouvant être une phase de simulation, de synthèse, de preuve.

Dans la suite, nous supposons que la description est syntaxiquement et sémantiquement correcte, et que la compilation est réalisée. Notre propos concerne les deux phases d'élaboration et de simulation. L'élaboration consiste à transformer une description concurrente et hiérarchique en un ensemble de processus interconnectés par des treillis (donc qui communiquent entre eux). Le résultat obtenu peut alors être exécuté à des fins de simulation du modèle initial.

II.3.1 – La phase d'élaboration

Cette phase consiste à transformer une description d'architecture en un ensemble de processus interconnectés en treillis.
Un treillis est un ensemble de pilotes, de signaux, (incluant les ports et les signaux implicites), de fonctions de conversions et de résolutions ; les valeurs effectives et courantes de chaque signal du treillis sont ainsi déterminées.

Un processus supplémentaire est construit : **le noyau**. Le noyau a accès à l'ensemble des signaux déclarés et implicites. Il sert à coordonner l'activité des processus définis par l'utilisateur. C'est un agent qui propage les valeurs des signaux et met à jour les valeurs des signaux implicites. Il détecte les événements produits et réveille les processus adéquats en réponse à ces événements.

♦ Pour tout signal donné S, qui est explicitement déclaré dans le modèle, le processus noyau contient une variable qui représente la valeur courante du signal. Toute évaluation d'un nom étiqueté par le signal S tient compte de la variable courante du signal S (celle contenue dans le processus noyau). Au cours de la simulation, le noyau met à jour cette variable en se basant sur les valeurs courantes des sources du signal.

♦ Le processus noyau contient également une variable qui représente la valeur courante de tout signal gardé, déclaré implicitement par la définition d'une condition de garde sur un bloc.

♦ En outre, pour tout signal S'Stable(T), défini pour tout signal S et pour tout temps T référencés dans le modèle, le processus noyau contient un pilote et une variable.
Il en est de même pour les signaux de type S'Quiet(T) et S'Transaction(T) et S'Delayed(T).

La phase d'élaboration est une phase statique (sauf en ce qui concerne les fonctions et procédures). A chaque élaboration, les instructions concurrentes (processus, assertions concurrentes, affectations concurrentes de signaux, bloc) sont transformées en processus.

Cette phase préliminaire permet la construction d'un modèle exécutable qui est simulable dans la sémantique de simulation de VHDL. En effet, dans la sémantique de simulation VHDL (cf chap9 LRM) les seuls objets manipulés sont les signaux, les variables, les processus et les sous–programmes (fonction et procédure).

II.3.2 – La phase de simulation

La simulation comprend l'exécution des processus définis par l'utilisateur. Ils interagissent les uns avec les autres mais aussi avec l'environnement. Lors d'une première étape, le système est initialisé. Ensuite, le cycle de simulation est exécuté de manière répétitive.

Définition : un cycle de simulation

Un cycle de simulation correspond à la mise à jour des signaux et du treillis. Il se déroule en trois étapes principales :

♦ Le temps avance jusqu'à ce qu'un pilote devienne actif ou qu'un processus endormi se réveille ;

♦ Les signaux sont mis à jour. Aussi, tous les processus qui étaient en attente d'événements sur ces signaux sont réveillés ;

♦ Chaque processus réveillé est exécuté jusqu'à sa suspension.

Au cours d'un cycle de simulation, les valeurs des signaux actifs sont calculées. En conséquence, un événement peut se produire sur un signal et un processus, sensible à ce signal, est réveillé et exécuté.

L'initialisation

Au début de l'initialisation, le temps courant Tc est supposé être 0 ns.

I1– Les valeurs pilotées et effectives de chaque signal déclaré explicitement sont calculées. Et la valeur effective est affectée à la valeur courante du pilote. Cette valeur est supposée avoir été la valeur du signal durant le temps (de longueur infinie) précédent le début de la simulation.
I2– La valeur de chaque signal implicite de la forme S'stable et S'quiet est positionnée à "true". La valeur initiale de chaque signal implicite de la forme S'delayed est positionnée à la valeur initiale de son préfixe S.
I3– La valeur de chaque signal implicite GUARD est initialisée par le résultat de l'évaluation de l'expression de garde.
I4– Chaque processus non post–synchronisé du modèle est exécuté jusqu'à ce qu'il soit suspendu.
I5– Chaque processus post–synchronisé du modèle est exécuté jusqu'à sa suspension.
I6– Le temps du cycle de simulation suivant (qui est dans ce cas le premier cycle de simulation) Tn est calculé suivant les règles de l'étape 6 du cycle de simulation décrit ci_après.

Un cycle de simulation

SIM1– Tc :=Tn le temps courant prend la valeur Tn. La simulation est terminée quand Tn = TIME'HIGH ou qu'il n'y a plus de pilotes actifs ni de réveil de processus au temps Tn.
SIM2– Chaque signal actif explicite du modèle est mis à jour (en conséquence, des événements peuvent arriver sur des signaux). Cela signifie la mise à jour de tous les pilotes actifs, et le calcul de la nouvelle valeur pilotée des signaux actifs. Un événement se produit sur un signal si sa valeur pilotée est modifiée.
SIM3– Chaque signal implicite du modèle est mis à jour (comme exprimé ci dessus).
SIM4– Pour tout processus P, si P est en attente sur le signal S et si un événement est survenu sur S au cours de ce cycle de simulation alors P est réveillé. L'exécution d'un processus P a pour effet de modifier les pilotes (associés à P) des signaux affectés dans P.
SIM5– Tout processus non post–synchronisé qui a repris la main au cours du cycle courant de simulation est exécuté jusqu'à sa suspension.
SIM6– Le temps du cycle de simulation suivant Tn est déterminé par la plus petite valeur de :

—TIME'HIGH
— La prochaine date à laquelle un pilote devient actif
— La prochaine date à laquelle un processus est réveillé

Si Tn = Tc alors le prochain cycle de simulation (s'il y en a un) sera un DELTA CYCLE.
SIM7– Si le pas de simulation suivant est un delta cycle alors l'étape 7 s'arrête là. Sinon, chaque processus post–synchronisé qui avait été réveillé, mais qui n'a pas été exécuté depuis son dernier réveil, est exécuté jusqu'à sa suspension. Puis Tn est recalculé suivant les règles de l'étape 6. Il y a erreur si l'exécution d'un quelconque processus post synchronisé entraîne l'occurrence d'un delta cycle immédiatement après le cycle de simulation courant.

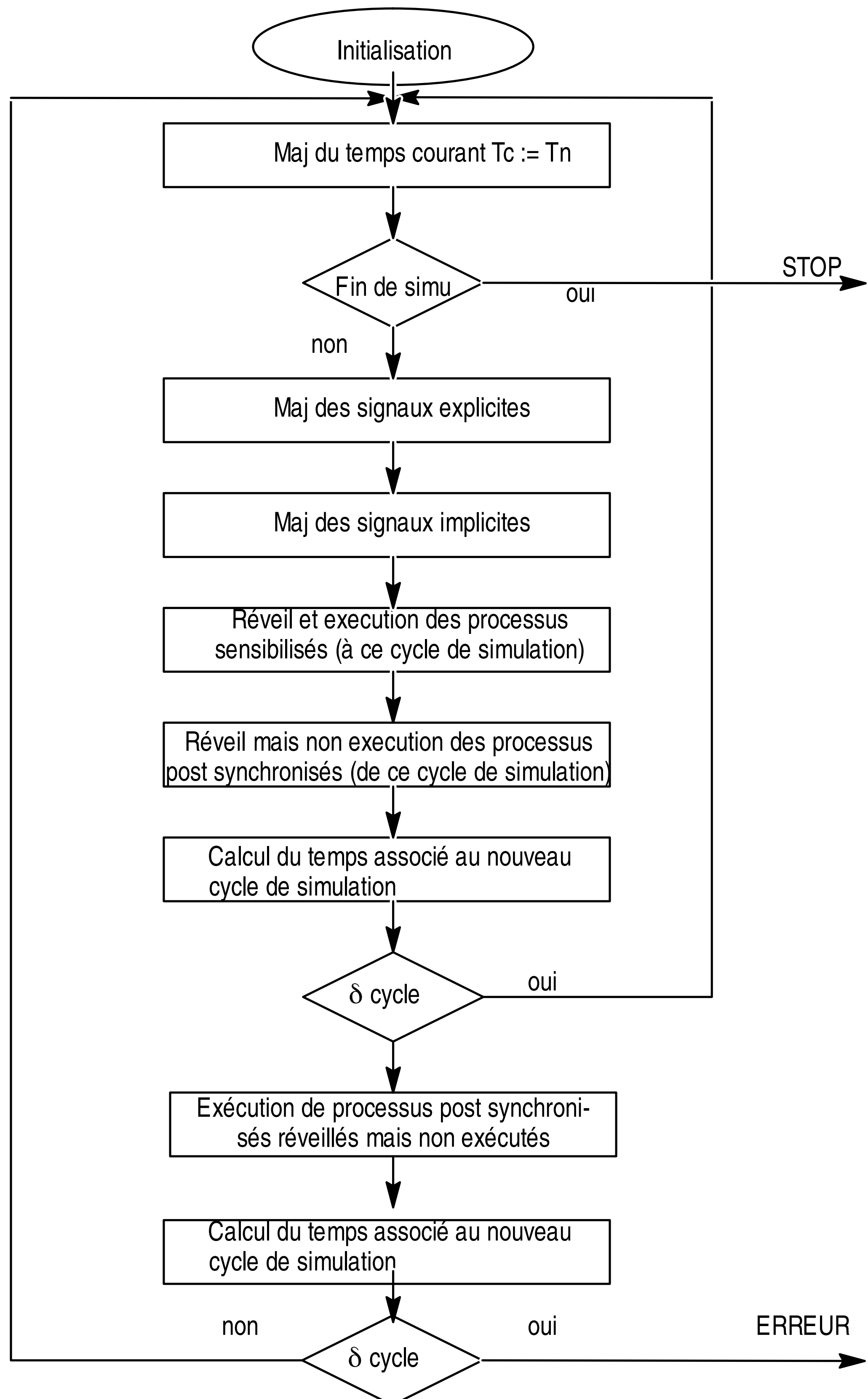


Figure 115 : Les étapes d'un cycle de simulation

III – LES PRINCIPES DE MODÉLISATION

III.1 – Définitions et principes

Considérons une description VHDL après la phase d'élaboration. Elle comprend un ensemble de processus représentatifs des processus définis par l'utilisateur et le processus supplémentaire créé lors de l'élaboration : le processus noyau.

Le point de départ de nos travaux sera la description obtenue après élaboration. Bien que la sémantique de VHDL prenne en compte les niveaux de description hiérarchique en termes de blocs et de processus, la synchronisation des blocs n'est définie qu'au travers de la phase d'élaboration et donc de la mise à plat de la description. Du point de vue systémique, la même sémantique de synchronisation au niveau blocs que celle définie au niveau processus aurait été souhaitable.

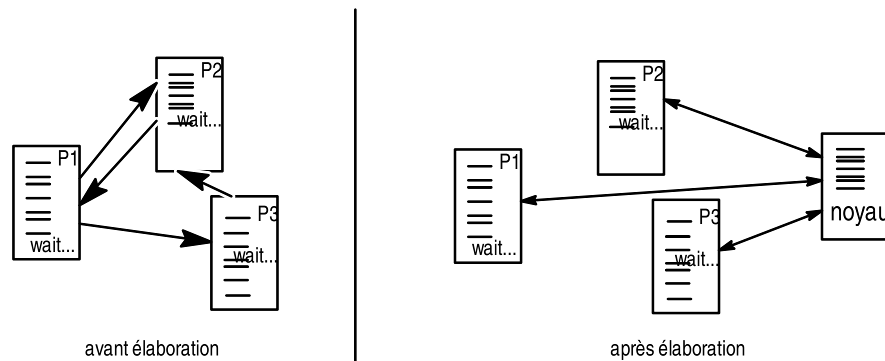


Figure 116 : Echanges entre processus VHDL

Une description VHDL évolue jusqu'à atteindre une situation de stabilité. A partir de cette situation stable, certains calculs sont effectués pour permettre une mise à jour du système. Puis l'évolution du modèle est relancée.

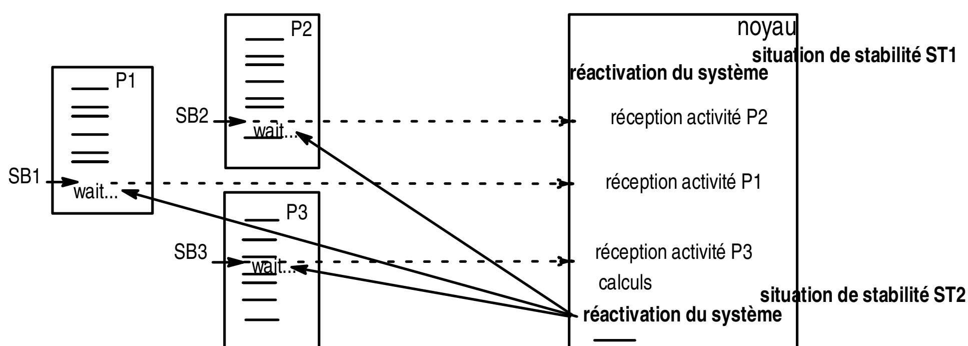


Figure 117 : De ST1 à ST2 : un cycle de simulation

Tous les processus activés évoluent simultanément jusqu'à atteindre un état wait. Quand tous les processus ont atteint un état wait, le système VHDL est dit stable.

Définition : La condition de stabilité d'un système VHDL correspond au fait que :

1) il n'y a plus de pilotes de signaux actifs au cours du pas de simulation courant,
2) tous les processus qui devaient être réveillés au cours de ce cycle se sont exécutés.

Chacune des situations de stabilités correspond à l'étape **Sim4** décrite dans le paragraphe précédent.

Définition : un processus s'exécute jusqu'à ce qu'il soit suspendu. Cela ne se produit que sur une instruction WAIT (ou l'équivalent d'un WAIT car tout processus peut être transformé en un processus équivalent avec uniquement des affectations de signaux et des instructions WAIT). L'état qui précède immédiatement toute instruction WAIT peut être défini comme un état de blocage d'un processus.

III.2 – Préservation du caractère synchrone

Tous les processus qui doivent être réveillés évoluent en parallèle jusqu'à leur situation de blocage. Lorsque la stabilité du système est atteinte, le noyau calcule le nouveau temps de simulation, affecte les valeurs courantes des signaux, détermine les nouveaux processus à réveiller. Puis le noyau réactive le système VHDL qui évolue jusqu'à une nouvelle stabilité. Les processus évoluent en parallèle donc le blocage arrive "n'importe quand" entre deux situations de stabilité. C'est le noyau qui synchronise l'évolution du système en attendant que tous les processus se soient exécutés avant de relancer "au même moment", ceux qui se sont réveillés au cours de ce nouveau cycle de simulation.

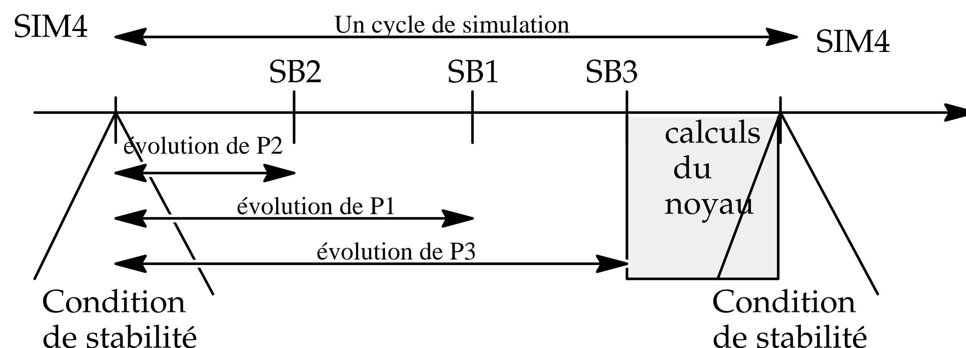


Figure 118 : Evolution dans le temps entre deux situations de stabilité

Notre objectif est de modéliser ce mécanisme selon une méthodologie décrite au Chapitre2. Nous présentons les principes de modélisation de ce mécanisme de communication sous forme STE (Système à Transitions Etiquetées). Notre intention est de modéliser chaque processus sous forme d'un STE (y compris le processus noyau). Ce sera le principe de fusion des transitions (d'une part celle des processus, d'autre part celles du noyau) qui permettra au système d'évoluer en préservant la sémantique exécutive synchrone.

Nous montrons (Figure 119 et Figure 120) le flot d'informations entre les processus et le noyau.

Tous les processus envoient leur future condition de réveil NCR et modifient les pilotes des signaux. puis le noyau met à jour les pilotes et calcule les nouvelles valeurs courantes des signaux. Nous notons DRIVi, la liste des pilotes gérée par le processus Pi, et ListS(Pi) la liste des valeurs courantes des signaux nécessaires aux processus Pi.

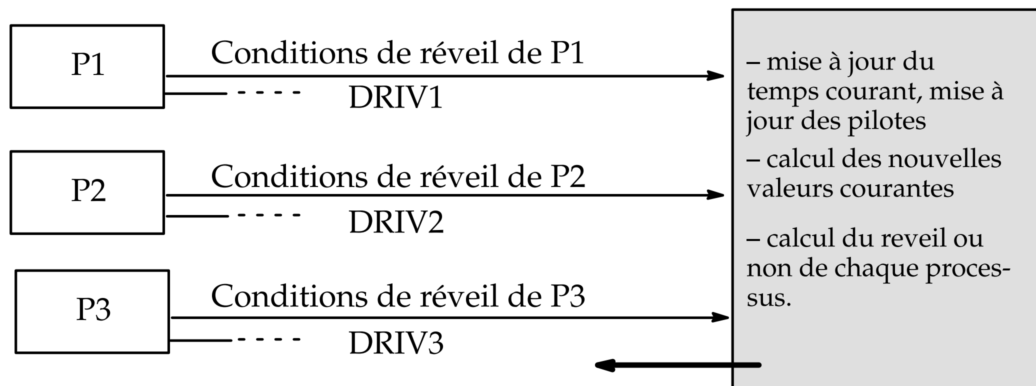


Figure 119: Flot d’information des processus vers le noyau

Chaque processus reçoit de manière synchrone l’information de se réveiller ou non. Puis, chaque processus réveillé lit les valeurs courantes nécessaires à l’exécution de la transition suivante, conduisant à de nouvelles conditions de réveil et à de nouveaux pilotes.

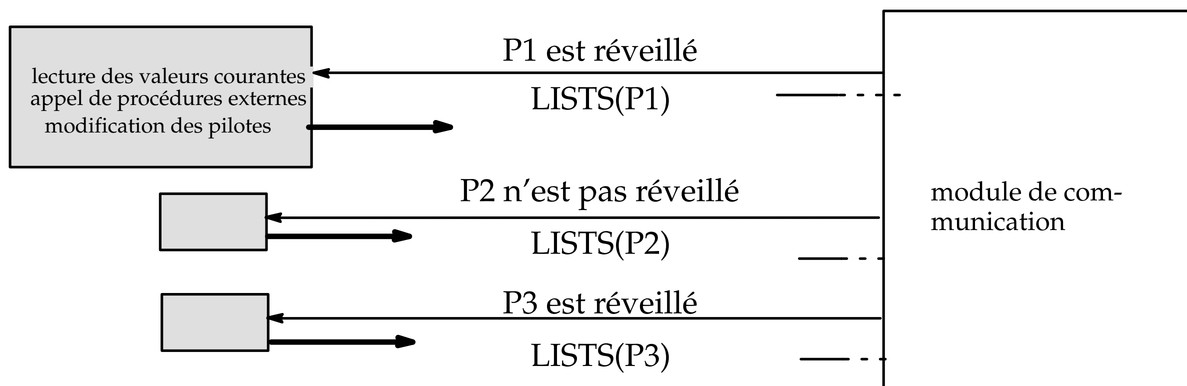


Figure 120: Flot d’informations du noyau vers les processus

Le cycle de simulation doit être considéré de manière atomique. En particulier, la communication depuis les processus vers le noyau, les opérations du noyau, et les communications du noyau vers les processus doivent être vus comme une tâche indivisible.

III.3 – Principes de la formalisation STE de la simulation de VHDL

Les grandes caractéristiques de la communication de VHDL peuvent se formaliser de manière tout à fait analogue à ce qui a été fait en VOVHDL. La correspondance des principales primitives est la suivante :

VOVHDL	VHDL
processus	processus après élaboration
evt de communication (SEND RECEIVE)	WAIT
ensemble des canaux de communication	Processus noyau
Un canal pour un send pour un receive	ensemble des signaux modifiés liste de sensibilité du wait
plusieurs protocoles	evt sur signal ou temporisation du wait
message émis message reçu	pilote des signaux modifiés valeur effective des signaux référencés

Figure 121 : La correspondance entre la communication de VHDL et de VOVHDL

Dans la suite, nous détaillons pour VHDL les points énumérés ci–dessus en termes de STE. Le lecteur ne sera pas étonné de retrouver de grandes similitudes avec le modèle du chapitre 2.

III.4 – Un processus VHDL

III.4.1 – Son rôle

Un processus VHDL s'exécute jusqu'à ce qu'il soit suspendu (i.e. jusqu'à sa situation de blocage). Entre deux situations de blocage, il gère son comportement, mais également sa communication. La gestion du comportement se fait immédiatement après une instruction WAIT, dès que le processus a repris son exécution. Sa communication consiste à envoyer au noyau son activité présente.

Il passe donc d'une phase de calcul (où il calcule les nouvelles valeurs des signaux) à une phase de communication (nouvelles affectations de signaux).

III.4.2 – Interprétation STE de son comportement

De même que nous l'avons fait pour un processus VOVHDL (cf chap2) nous pouvons interpréter un processus VHDL sous forme de STE qui communique par la sémantique de communication CCS. Pour cela, nous considérons le comportement synchrone d'un processus. Un processus est constitué d'un ensemble de transitions chacune correspondant à un événement de communication avec les autres processus. Il s'agit de définir proprement les états et les conditions de changements d'états. Un processus est dans une situation de blocage. C'est à dire que son exécution a été stoppée juste avant une instruction WAIT. Dès que le processus reçoit la condition de réveil adéquate, il s'exécute jusqu'à la nouvelle situation de blocage, et ainsi de suite (Figure 122).

Nous interprétons l'exécution d'un processus de manière STE avec les principes suivants :

- ♦ Un état correspond à l'instant immédiatement avant une instruction WAIT
- ♦ Une transition correspond à l'exécution entre deux WAIT consécutifs

- ♦ Les événements de communication (les étiquettes du STE) correspondent aux flots de données tels que définis précédemment.

Un état du STE correspond à l'état de blocage et la transition correspond au fait qu'une communication est effective entre le noyau et le processus.

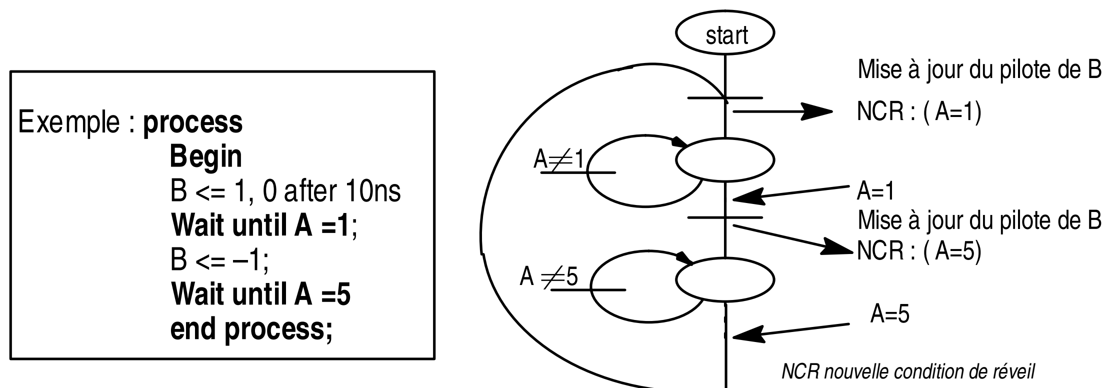


Figure 122 : Evolution d'un processus VHDL

III.4.3 – Modélisation sous forme canonique

Comme pour VOVHDL et pour les mêmes raisons, le comportement d'un processus VHDL est mis sous forme générique et particularisée (cf chap2 II –2.2).

La partie générique

C'est la partie qui gère l'évolution d'un processus depuis un état WAIT jusqu'au suivant. Quand un processus a atteint un état WAIT, il communique au noyau son activité constituée de ses nouvelles conditions de réveil et des nouvelles valeurs de ses pilotes.

Considérant les activités données par tous les processus, le noyau peut vérifier si les conditions de réveil des processus sont satisfaites ou non. C'est pourquoi deux transitions sont possibles pour caractériser un processus sous forme STE.

- ♦ La première (étiquetée par *nok*) correspond à une situation de blocage du processus qui reste dans le même état.
- ♦ La seconde (étiquetée par *ok*) où le processus peut évoluer. C'est à ce moment là que la partie particularisée est appelée

Chaque boucle sur l'état générique WAIT, franchie par l'une ou l'autre des deux transitions, correspond à un pas de simulation VHDL.

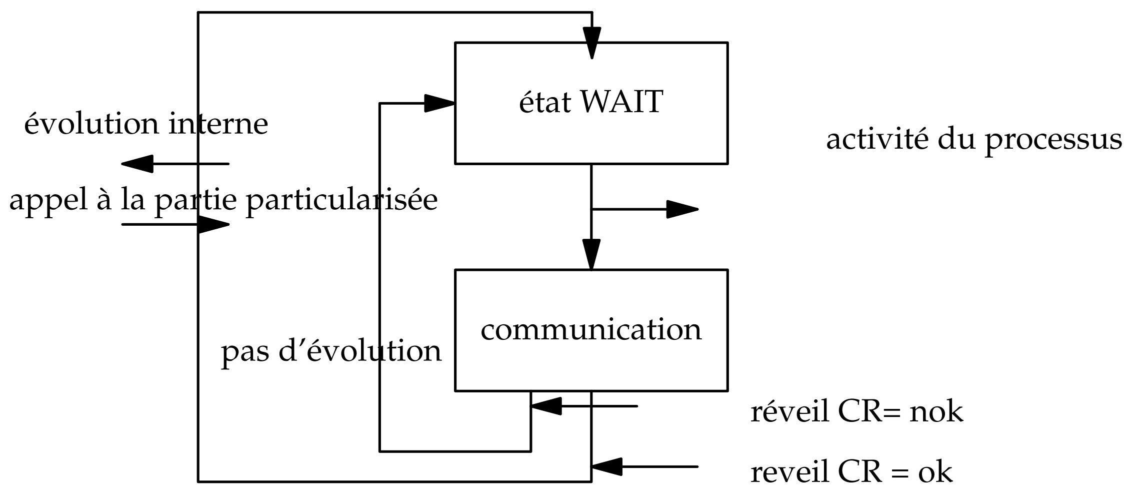


Figure 123: Le comportement du module générique

De même que pour VOVHDL, nous pouvons modéliser la partie générique sous forme d'un STE avec un seul état. Ainsi le STE de la Figure 122 sera représentée par un STE constitué des deux transitions suivantes :

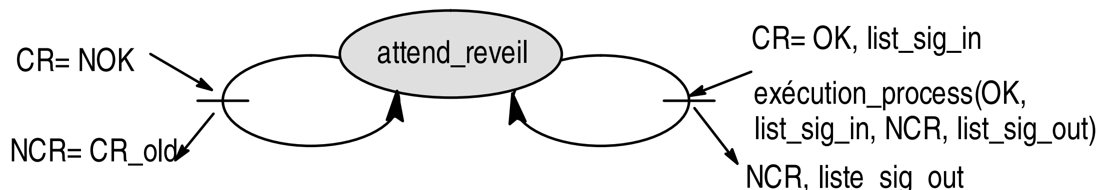


Figure 124 : Comportement STE d'un processus VHDL

Nous donnons ci_dessous la description d'une de ces transitions dans la syntaxe ASA+.

```
TRANS reveilOK
WHEN message_entree( OK, list_sig_in)
FROM attend_réveil
TO attend_réveil
BEGIN
        execution_process(Ok,list_sig_in, NCR,list_sig_out)
        OUTPUT message_sortie(NCR, list_sig_out)
END;
```

La transition de nom reveil_OK est tirée lorsque le processus reçoit en entrée le message (OK list_sig_in). Le processus passe de l'état attend_réveil dans lui_même après avoir

1) calculé la nouvelle condition de réveil du processus(NCR) et les nouvelles valeurs des signaux qui ont été modifiés au cours de son exécution (par l'appel de la fonction "exécution_process"),

2) renvoyé en message de sortie cette condition et ses valeurs (NCR, list_sig_out).

La partie particularisée

La sémantique d'exécution de chaque processus VHDL est modélisée de manière générique mais chaque processus aura son propre comportement. Ce comportement est interprété sous forme d'une machine à états finis dont les états sont définis comme les instants juste avant une instruction WAIT et sont caractérisés par les valeurs courantes des signaux à ces instants. Une transition correspond ainsi à l'exécution d'instructions comportementales depuis un état donné jusqu'au suivant.

L'objectif est double :

- stopper l'exécution lorsqu'un état est atteint et communiquer à la partie générique l'activité du processus (pilotes et conditions de réveil du processus).
- lorsque la partie générique le demande, reprendre l'exécution correspondant à une nouvelle transition.

D'après la définition des états et des transitions données ci–dessus, la machine à états finis du processus est en fait son graphe des états atteignables. Or, le calcul formel de ce graphe à partir d'une description VHDL se heurte au problème du développement des instructions de branchements conditionnels.

C'est pourquoi la machine d'état d'un processus VHDL est calculée par une simulation exhaustive du processus.

III.5 – Le noyau de communication

III.5.1 – Son rôle

Le noyau doit cadencer la simulation : il doit donc être capable de réveiller les processus "au bon moment" et donner aux signaux "leur bonne valeur".
Le rôle du noyau est donc de déterminer les conditions de réveil des processus et de gérer les pilotes et les valeurs de chaque signal. Le noyau doit donc avoir la connaissance de tous les signaux (pilotes et valeurs effectives).

Le noyau doit connaître la condition de réveil de chaque processus à chaque pas de simulation. Les conditions de réveil peuvent être de deux types :

- Celles qui correspondent au fait qu'un signal a été modifié. Par exemple : si un processus était bloqué sur l'état "Wait on S1" et que S1 a changé de valeur par rapport au pas de simulation précédent alors il est réactivé.
- Celles qui correspondent à l'expiration d'une date. Par exemple, si un processus était bloqué sur la condition "Wait for 30 ns" et que ce délai est écoulé au cours de ce pas de simulation alors le processus doit reprendre son exécution.

Le noyau de communication peut donc déterminer la liste des processus à réveiller. Il doit envoyer à chaque processus, l'information "réveille toi" ou "reste endormi" ainsi que la liste des valeurs effectives des signaux nécessaires à son exécution en cas de réveil.

III.5.2 – Son comportement

Comme expliqué ci–dessus, le noyau gère l'évolution du système pour un pas de simulation avant de réveiller les processus. Entre ces deux événements de communication, le module noyau doit calculer successivement différents résultats à partir des nouvelles valeurs des pilotes modifiés par les processus. Ces résultats peuvent être résumés comme suit : le nouveau temps courant, la mise à jour des pilotes et des valeurs courantes de chaque signal, le résultat de la condition de réveil de chaque processus.

De même que ce que nous avons fait pour les processus, le module de communication représente l'exécution du noyau avec une vision abstraite et atomique : un seul état et une seule transition.

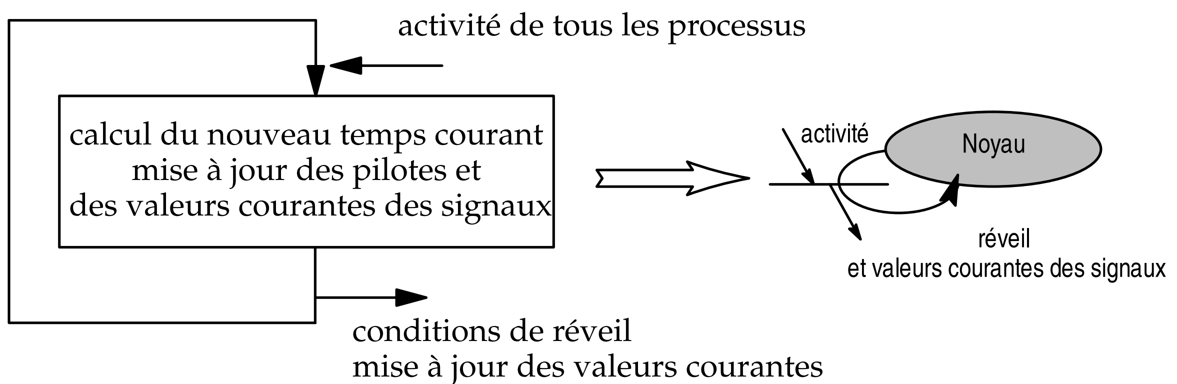


Figure 125: Le comportement du module de communication

Nous modélisons le comportement décrit ci–dessus sous forme de LTS. Celui–ci est constitué d'une seule transition qui boucle sur un seul état.

III.5.3 – Son interprétation STE

Le noyau reçoit en entrée pour chaque processus (de 1 à Max_proc) la condition de réveil (NCR) et la liste de valeurs de signaux que renvoie le processus (liste_sig_out). En fonction de ces données d'entrée, il exécute un cycle de simulation. Puis il renvoie à chaque processus son information de réveil. Il renvoie "Ok" si le processus est réveillé avec les valeurs de signaux dont il a besoin pour poursuivre son exécution (liste_sig–in). Il renverra "Nok" si il n'est pas réveillé et dans ce cas le processus restera bloqué et la condition de réveil restera la même pour le pas de simulation suivant.

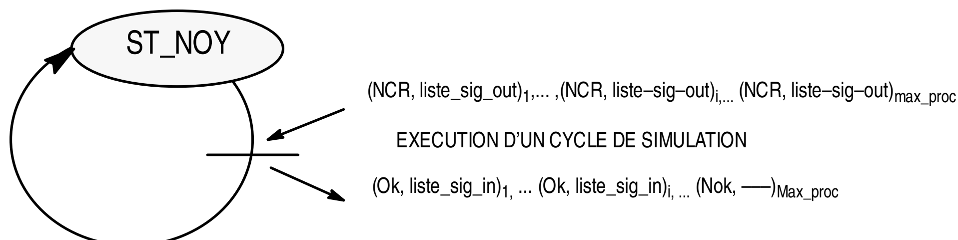


Figure 126 : Représentation STE du noyau

La description de l'unique transition associée au noyau sera la suivante :

TRANS execution_cycle
WHEN message_entree(NCR, list_sig_out)$_{1,...}$(NCR, list_sig_out)$_{i..}$(NCR, list_sig_out)$_{max_proc}$)
FROM ST_NOY
TO ST_NOY
BEGIN
  cycle_simulation(((NCR, list_sig_out)$_{1,...}$(NCR, list_sig_out)$_{i..}$(NCR, list_sig_out)$_{max_proc}$),
    ((Ok, liste_sig_in)$_{1,}$ ... (Ok, liste_sig_in)$_{i, ...}$ (Nok, —)$_{Max_proc}$)
  OUTPUT message_sortie((Ok, liste_sig_in)$_{1,}$ ... (Ok, liste_sig_in)$_{i, ...}$ (Nok, —)$_{Max_proc}$)
END;

De même que pour la modélisation VOVHDL, la modélisation STE est très simple et est constituée de peu de transitions. Une grande partie du comportement du noyau réside dans la fonction *cycle_simulation*, qui réalise les opérations du noyau de simulation VHDL.

III.5.4 – Les opérations du noyau

Le noyau gère les valeurs effectives de chaque signal. Ces valeurs sont mises à jour de temps à autre suivant les valeurs courantes des sources de ce signal (les valeurs fournies par la première transaction des pilotes). Le noyau gère également la valeur courante de chaque signal GUARD implicite, et pour chaque signal implicite une variable et un pilote.

Toutes ces opérations sont séquencées par le cycle du processus noyau qui reproduit le cycle de la simulation de VHDLexposé page 89:

III.6 – La synchronisation de l'ensemble au moyen du Rendez–Vous

III.6.1 – L'aspect synchrone d'un processus

D'un point de vue abstrait, la sémantique de l'algèbre CCS autorise la spécification d'un canal de communication qui supporte un transfert de données bi–directionnel. De cette manière, l'activité est envoyée au module noyau et le résultat est reçu au cours d'une unique transition. La signification d'une telle représentation est que le traitement effectué par le processus est considéré comme atomique (indivisible).

Comme pour VOVHDL, une conséquence de la modélisation abstraite est que le modèle semble être non déterministe (deux transitions d'un processus soient tirables en même temps).

Ainsi la propriété d'atomicité (un seul état et une seule transition) de notre modèle garantit son adéquation au principe d'exécution synchrone, avec un même pas de simulation.

III.6.2 – L'évolution concurrente de l'ensemble

De manière à représenter l'évolution concurrente de tous les processus, notre modèle repose sur les deux points :

- ♦ La modélisation du noyau comme un module de communication (medium),
- ♦ La forte synchronisation de tous les processus avec ce module de communication.

En ce qui concerne l'évolution de tous les processus qui composent un système, l'exécution synchrone du langage VHDL repose sur le fait que tous les processus atteignent une nouvelle affectation WAIT au même moment, et tous les processus non–postsynchronisés réveillés sont réexécutés au même moment. C'est par cette synchronisation de l'ensemble de processus que le système évolue.

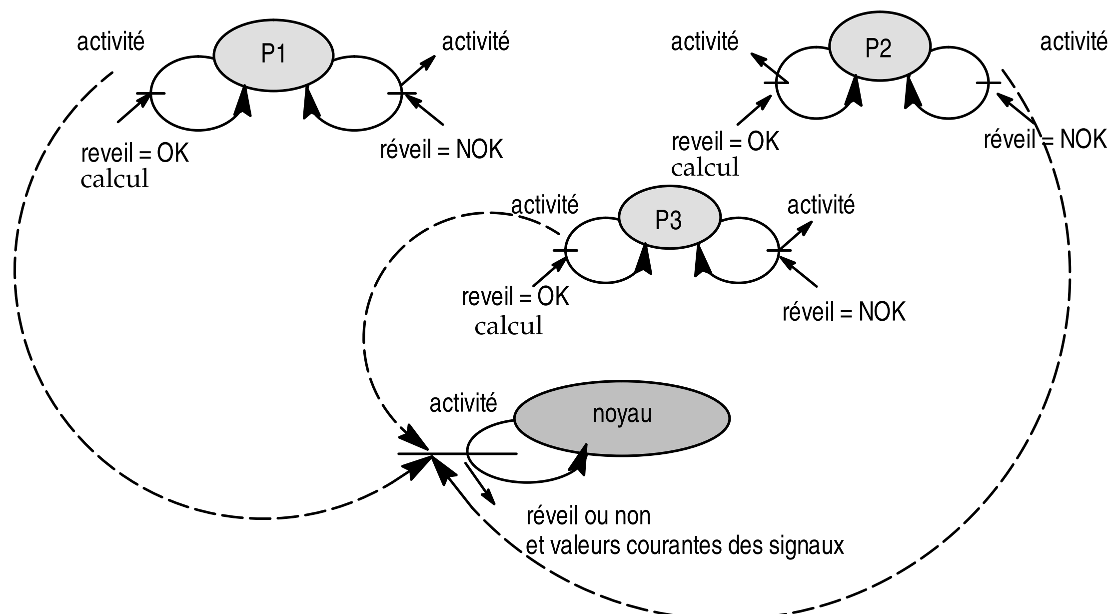


Figure 127: Les processus donnent leur activité de manière synchrone.

Grâce à notre modélisation en deux partie (générique et particularisée) d'un processus, et grâce au module de communication, la synchronisation est naturellement supportée par des règles de composition simples : cela consiste à faire communiquer tous les processus avec le module de communication au moyen du Rendez_Vous multiple. Ainsi, pour chaque processus, une des deux transitions définies à la Figure 124 fusionne avec une transition du noyau.

IV – LA MODÉLISATION

IV.1 – La modélisation des processus

Chaque processus VHDL peut être implantée par un module ASA+. L'interprétation en deux parties d'un processus (d'une part la partie générique et la partie particularisée) permet de modéliser de manière générique le comportement d'un processus.

```
TRANS
        FROM wait_state
        TO wait_state
        BEGIN
                IF awakening_glob is ok THEN
                call_C_procedure( prog_counter, system_glob, awakening_cond, data–rec)
                END
                OUTPUT to_kernel (wait_cond)
        END
```

où le suffixe glob signifie que la variable est globale et, de ce fait, visible de tous les modules.
prog_counter est le moyen de reprendre l'exécution de la procédure C au bon endroit.

Figure 128: Une transition ASA+ représentative du comportement d'un processus

La particularisée de chaque module est représentée au moyen d'une procédure ré–entrante C obtenue à partir de la description VHDL et en utilisant les instructions WAIT. L'objectif est de pouvoir reprendre l'exécution du dernier point d'arrêt jusqu'au point d'arrêt suivant. Le flot de donnée de cette procédure sera :

- ♦ En entrée, le pointeur du dernier point d'arrêt (prog_counter) et les nouvelles valeurs courantes des signaux et des pilotes mis à jour (system_glob),
- ♦ En sortie, le pointeur du point d'arrêt atteint(prog_counter), les conditions de réveil des processus et les pilotes modifiés.

Nous donnons une rapide description de la procédure C :

<table>
<tr>
<td>

```
Process
begin
   B<=1, 0 after 10 ns;
   wait until A =1
   B <= –1
   wait until A =5
end;
```

</td>
<td>

```
C_procedure( prog_counter, system_glob, awakening_cond,  data_rec)
            (* déclarations internes à la procédure C *)
            switch  (prog_counter)
            case 0    : GOTO label0
            case 1    : GOTO label1
            case 2    : GOTO label2
     label0 :
            maj_pilote( B, B<=1, 0 after 10 ns);
            maj_vecteur_d_etat( data_rec)
            awakening_cond = (A=1)
            prog_counter:=1
            exit
     label1 : (* correspond à la cond de réveil A =1 *)
            maj_pilote( B, B<=–1);
            maj_vecteur_d_etat( data_rec)
            awakening_cond = (A=5)
            prog_counter :=2
            exit
     label2 : (* correspond à la cond de réveil A =5 *)
            maj_vecteur_d_etat( data_rec)
            awakening_cond = True
            GOTO label0
            exit
```

</td>
</tr>
</table>

Figure 129: D’un processus VHDL vers une procédure C

L ’information donnée au noyau, et correspondant à l’évolution des pilotes, est transmise au moyen de la variable globale *system–glob.* La variable globale est utilisée pour implanter le transfert bi–directionnel de données. La variable *data–rec* qui mémorise l’état courant des variables internes de la procédure, peut être vu du module ASA+. Cela permet la construction du graphe des états atteignables d’un processus VHDL.

IV.1.1 – Le noyau

Le noyau sera représenté par un module ASA+ et modélisé en une seule transition. Toutes les opérations du noyau sont décrites dans le bloc action de l’unique transition.

```
TRANS
        WHEN  from_all_process( wait_condition)
        FROM kernel_state
        TO kernel_state
        BEGIN
                update  system_glob and awakening_glob
        END
```

Figure 130: La modélisation du noyau par une transition

CONCLUSION

L'objectif initial des travaux était de modéliser la description informelle d'un composant matériel afin de valider cette description vis à vis d'une norme de réseau industriel. Cette modélisation a motivé la définition du langage VOVHDL. En effet, VOVHDL a été conçu pour permettre la description de mécanismes de communication au niveau système. Les choix retenus dans VOVHDL présentent un double avantage. Tout d'abord, le langage possède une sémantique essentiellement orientée vers l'expression de la synchronisation ; par conséquent, il se prête particulièrement bien à une formalisation en termes de processus communicants. Puis, il peut être traduit en VHDL ; ce qui assure un lien, d'une part vers la simulation et d'autre part, vers les outils conventionnels de conception de matériel.

Ma contribution à la définition de VOVHDL a concerné la formalisation des protocoles de communication et la modélisation de la sémantique du langage sous forme de Systèmes à Transitions Etiquetées.

La réalisation informatique de cette modélisation a été programmée dans le langage de l'outil ASA+.

L'ensemble a permis de réaliser les objectifs initiaux du projet en produisant un modèle du circuit FICOMP analysable au moyen d'outils de validation. A l'heure actuelle, l'architecture matérielle du circuit a été validée.

Le travail réalisé à partir de VOVHDL nous a conduit naturellement à nous intéresser à VHDL. Ainsi, en appliquant la même formalisation basée sur la séparation de la partie algorithmique de la partie communication des processus, il a été possible d'aboutir à la définition d'une sémantique synchrone pour l'algorithme de simulation de VHDL.

Nos travaux nous laissent envisager une ouverture vers d'autres projets du même type. En effet, cette double définition sémantique de VOVHDL et VHDL en terme de sémantique d'exécution synchrone peut servir de base méthodologique dans le cadre de la conception mixte du matériel et du logiciel (co–design). Cela concerne en particulier la modélisation et la validation des aspects liés à la synchronisation du logiciel avec les parties matérielles.
De plus, notre approche met en évidence deux aspects dans la formalisation de langages tels que VHDL ou VOVHDL. Dans nos travaux, nous nous sommes attachés à l'aspect communication et synchronisation des éléments du langage au moyen d'une approche opérationnelle. Cependant, cette approche doit être complétée par la formalisation du comportement de chaque processus élémentaire ; celle–ci permettrait d'établir la vérification de la conformité de notre interprétation opérationnelle. Cependant, ce deuxième aspect de formalisation relève d'autres outils de vérification ; dans la pratique, il n'est pas réaliste de fusionner ces deux approches : la taille du modèle obtenu serait trop grande.

Il reste à réaliser des outils de traduction automatique permettant de passer directement d'un modèle VHDL à un modèle ASA+ complet selon le même schéma que celui adopté pour VOVHDL. Lorsqu'un tel outil sera disponible, il sera possible de valider la mise en oeuvre VHDL de spécifications VOVHDL à l'aide de techniques de comparaison de traces.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[AI, 90] ***R. AIRIAU, J.M. BERGE, V. OLIVE, J. ROUILLARD***
" *VHDL: du langage à la modélisation*"
Presses Polytechniques et Universitaires Romandes. 1990

[ACJV, 93] ***Y. ATAMNA, R. CARMO, G. JUANOLE, F. VASQUES***
" *Le modèle Réseau de Petri Temporisé Stochastique (RdPTS) pour l'analyse qualitative et /ou quantitative des systèmes distribués temps réel.*"
The real time system conference. Paris France, janvier 1993.

[At, 94] ***Y. ATAMNA***
" *Réseaux de Petri temporisés stochastiques classiques et bien formés: définition, analyse et application aux systèmes distribués temps réel*"
Thèse de Docteur de l'Université Paul Sabatier de Toulouse, 1994.

[Be, 94] ***M. BELADJ***
" *Conception d'architecture en utilisant Signal et VHDL* "
Thèse de Docteur de l'Université de Renne I, 16 dec 1994.

[BCG, 93] ***M. BELHADJ, R. MC CONNELL, P. Le GUERNIC***
" *A framework for Macro– and micro–time to model VHDL attributes*"
Proceeding de EURO–VHDL/EURO_DAC '93 Hambourg; 20–24 sept 93, IEEE Computer Societed Press

[BCDM, 86] ***J. BURCH, E.M. CLARKE, K.L. Mc MILLAN, D. DILL, L. HWANG,***
" *Symbolic model checking : 10^{20} states and beyond* "
LICS'90 : 5th annual symposium on logic in Computer Science Juin 1990.

[BFMR, 92] ***J.M. BERGE, A. FONKOUA, S. MAGINOT, J. ROUILLARD***
" *VHDL designer's reference* "
Kluwer Academic Publishers 1992.

[BFK, 93] ***P. BREUER, L.S. FERNANDEZ, C.D. KLOOS***
" *Clean formal semantics for VHDL*"
Proceeding de EURO–VHDL/EURO_DAC '93 Hambourg; 20–24 sept 93, IEEE Computer Societed Press .

[BFK, 95] ***P. BREUER, L.S. FERNANDEZ, C.D. KLOOS***
" *A Simple Denotational Semantics, Proof Theory and a Validation Condition Generator for Unit–Delay VHDL*"
Formal Method in System Design, vol 7, n 1–2, p 26–52, aout 1995, Kluwer.

[BGM, 94] ***E. BÖRGER, U. GLÄSER, W. MÜLLER***
" *The Semantics of VHDL'93 Description*"
Proceeding de EURO–VHDL/EURO_DAC '93 Hambourg; 20–24 sept 93, IEEE Computer Societed Press.

[BK, 85] ***J.A. BERGSTRA, J.W. KLOP***
" *Process Algebra : specification and verification in bisimulation semantics* "
Proceeding de CWI symposium mathematics and computer science II, North Holland, 1986.

[Br, 88] ***E. BRINKSMA***
" *On the design of extended LOTOS*"
PhD Thesis, Twente University, Holland 1988.

[BS, 93] ***J. BOWEN, V. STAVRIDOU***
" *Systèmes à sureté de fonctionnement critique: Méthodes formelles et normes* "
Génie logiciel et systèmes experts, Numéro30, mars 93.

[BS, 95] ***D. BORRIONE, A. SALEM***
" *Dénotational sémantics of a synchronous VHDL subset*"
Formal Method in System Design, vol 7, n 1–2, p 53–72, aout 1995, Kluwer.

[BSBCP, 94] ***C. BAYOL, B. SOULAS, D. BORRIONE, F. CORNO, P. PRINETTO***,
"*A process algebra interpretation of a Verification oriented Overlanguage of VHDL*"
Proceeding de EURO–VHDL/EURO_DAC '93 Hambourg; 20–24 sept 93, IEEE Computer Societed Press.

[CCPBS, 93a] ***P. CAMURATI, F. CORNO, P. PRINETTO, C. BAYOL, B. SOULAS***
" *VOVHDL : A verification–oriented dialect of VHDL*"
VHDL–FORUM for CAD in Europe HAMBURG sept 1993.

[CCPBS, 93b] ***P. CAMURATI, F. CORNO, P. PRINETTO, C. BAYOL, B. SOULAS***
" *Inter–process Communication for System–Level Design*"
CODES'93 IEEE Int. Workshop on Hardware–Software Co–design, Cambridge oct 1993.

[CCPBS, 93c] ***P. CAMURATI, F. CORNO, P. PRINETTO, C. BAYOL, B. SOULAS***
" *A verifiable methodology at system level* "
ICVC'93 IEE 3rd Int. Conference on VLSI and CAD, Taejon (Korea) Novembre 1993.

[CLSI, 90] ***CLSI***
" *IEEE Standard 1076 VHDL Tutorial*"
Rockville, MD, USA, 1990.

[Da, 93] ***K. DAVIS***
" *A denotational definition of the VHDL simulation Kernel*"
CHDL 93 Ottawa, 26–28 Avril, North Holland.

[DB, 95a] ***D.DEHARBE, D. BORRIONE***
" *Symbolic model checking with past and future temporal modalities : fundamen tals and algorithms*"
In model generation in electronic design, édité par J.M.Bergé, O.Levia, J.Rouillard, Kluwer.

[DB, 95b] ***D.DEHARBE, D. BORRIONE***
" *A qualitative finite subset of VHDL semantics*"
Rapport de recherche ARTEMIS 943–I Grenoble, fev 1995

[DBDD, 93] ***D. DESBIEN, G. BOCHMANN, A. DAS, J. DARGHAM***
" *Modeling and formal Specification of the personal Communication service*"
Proceeding de INFOCOM' 93 VOL2 p756–765.

[DDH&, 93] ***W. DAMN, G. DÖHMEN, J. HELBIG, R. HERMANN, B. JOSKO, P. KELB, F. KORF, R. SCHLÖR***
" *Correct System Level design with VHDL*"
Formal semantics for VHDL, édité par K. Delgado Kloos et P. Breuer, Kluwer,Janv 1995.

[DHP, 95] ***G. DÖHMEN, R. HERMANN, H. PARGMANN***
" *Translating VHDL into Functional Symbolic Finite–State Models"*
Formal Method in System Design, vol 7, n 1–2, p 125–148, aout 1995, Kluwer.

[DJS, 93] ***W. DAMM, B. JOSKO, R. SCHLÖR***
"*A Net–Based Semantics for VHDL "*
Proceeding de EURO–VHDL/EURO_DAC '93 Hambourg; 20–24 sept 93, IEEE Computer Societed Press

[DJS, 95] ***W. DAMM, B. JOSKO, R. SCHLÖR***
" *Specification and verification of VHDL–based System–level Hardware design"*
Edited by E. Boerger in "Specification and Validation Methods for Programming Languages and Systems", Oxford University Press, p 331–410, 1995.

[DNHe, 84] ***R. De NICOLA, M. HENNESSY***
" *Testing equivalence for processes*"
Theorical computer Science 34 1984 p 83–133.

[DO, 93] ***A. DEBREIL, P. ODDO***
" *Synchronous design in VHDL*"
Proceeding de EURO–VHDL/EURO_DAC '93 Hambourg; 20–24 sept 93, IEEE Computer Societed Press

[DR,92] ***K. DRIRA***
" *Transformation et composition de graphes de refus : analyse de la testabilité*"
Thèse de l'université Paul Sabatier RAPPORT LAAS N 92435, Toulouse octobre 1992.

[DW, 92] ***C. DENDORFER, R. WEBER***
" *From service specification to protocol entity implementation – An exercise in formal protocol development"*
Protocol spécification, testing and verification, XII R. Linn et M. Üyar Elsevier Scien ces Publisher B.V. (North Holland), 1992.

[EMCO, 92] ***H. EHRIG, B. MAHR, I. CLASSEN, F. OREJAS***
" *Introduction to algebraic specification Part 1 : Formal Method For Software Developement* ".
The Computer Journal VOL. 35, Numéro 5,1992.

[En, 94] ***E. ENCRENAZ***
" *Une méthode de vérification de propriétés de programmes VHDL basée sur des*

techniques de validation de réseau de Petri"
Rapport de présoutenance de thèse de doctorat de l'universite de Paris VI , nov. 1994

[ES, 89] ***ISO***
" *ESTELLE, formal Description technique based on extended States Transition Model* "
Norme de L'ISO 9074, 1989.

[Fe, 88] ***J.C. FERNANDEZ***
" *Un système de vérification par réduction de processus communicants*"
Thèse de docteur de l'université Joseph Fourier de Grenoble, 1988.

[FPS, 91] ***C. LE FAOU, L. PIERRE, A. SALEM***
" *A User Oriented Presentation of PREVAIL: A Proof Environment for VHDL descriptions* ",
Technical report N 71, IMAG/ARTEMIS Grenoble, septembre 1991.

[FI, 92] ***FICOMP PARTNERS***
" *Ficomp Fieldbus component : technical annexe* "
project 6526 version 2.1 22 avril 1992.

[FI, 93a] ***C. BAYOL, B. SOULAS, P. CAMURATI , F. CORNO, P. PRINETTO***
" *System modelisation and verification Method Strategy*"
Deliverable D31 Ficomp project(6526) EDF/ Politecnico di Torino juillet 1993.

[FM, 95] ***M. FUCHS, M. MENDLER***,
" *Functional semantics for delta delay VHDL based on Focus* "
Edited by C. D. KLOOS in "The Kluwer International series in Engineering and Computer Science" Vol 307. 1995.

[Ho, 85] ***C. HOARE***
" *Communicating Sequential Processes*",
Prentice Hall, Englewood Cliffs, NY(USA), 1985.

[Ho, 93] ***G. HOLZMANN***
" *Design and validation of protocol: a tutorial* ",
Computer Networks and ISDN systems 25, p 981–1027, 1993.

[KB, 93] ***M. KLEYN , J. BROWNE***
" *A High Level Language for specifying Graph Based Langages and their Pro gramming Environments* ".
15 th International Conference on Software Engineering 1993.

[Ko, 78] ***Z. KOHAVI***
" *Switching and Finite Automata Theory* ".
Second edition, Computer Science McGrAW HILL New York, 1978.

[Koo, 91] ***J.C. KOOMEN***
" *The design of communicating systems: a system engineering approach.*"
Kluwer Academic Publishers, Boston, MA(USA), 1991.

[Le, 90] ***G. LEDUC***
" *On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS* "
Dissertation présentée pour l'obtention du grade d'Agrégé de l'Enseignement Supérieur, Uni versité de Liège juillet 1990.

[LlAV, 90] ***JC. LLORET, P. AZEMA, F. VERNADAT***
" *Compositional design and verification with Labelled Predicates/Transition Nets*"
Proceedings Computer Aided Verification 90, Rutgers University, New Jersey 1990.

[LFMM, 92] ***B.LEVY, I. FILIPENKO, L. MARAIS, T. MENAS***
" *Using the state Delta verification System (SVDS) for Harware Verification*"
Theorem Proovers in circuit Design édité par V.Stavridou, T.Melhou et R.Boute North Holland 1992.

[LLo, 90] ***J.C. LLORET***
" *Réseaux Prédicats Transitions Etiquetés pour la Modélisation et la Vérification de Systèmes Informatiques Répartis*"
Thèse de Docteur de l'université Paul Sabatier de Toulouse 1990.

[LMcA, 92] ***G. LUNDY, R. Mc ARTHUR***
" *Formal Model of a High Speed Transport Protocol*"
Protocol specification. testing and verification, XII R. LINN et M. ÜYAR North Holland, 1992.

[LRM, 88] ***IEEE STANDARD VHDL LANGAGE REFERENCE MANUEL***
IEEE Inc.,New_York NY, Mar 1988.

[LRM, 93] ***IEEE STANDARD VHDL LANGAGE REFERENCE MANUEL***
ANSI/IEEE std 1076–1993

[MA, 92b] ***F. MARTINOLE***
" *Fusion of VHDL processe*s"
CRCTR91, Stanford Université Stanford CA Dec 1990.

[Man 74] ***Z. MANNA***
" *Mathematical theorie of computation*"
Mc Graw Hill Computer Science Series 1974.

[Mi, 89] ***R. MILNER***
" *Communication and Concurrency*"
Prentice Hall, Englewood Cliffs, NY (USA), 1989.

[Mil, 85] ***G. MILNE***
" *Circal and the representation of communication, concurrency and time* "
ACM transaction on Programming Languages and Systems Vol 7 N° 2, 1985.

[Mil, 89] ***G. MILNE***
" *Design for verifiability*"
Workshop on "Hardware Specification Verification and Synthesis: mathematicals aspects", Cornell University, Ithaca, NY (USA), juillet 1989.

[NFS, 95] ***E. NAJM, J.B. STEPHANI, A. FEVRIER***
" *Toward a mobile LOTOS*"
Proceeding de 8th int. Conf. on FORTE'95, oct 17–20, Quebec CANADA, 1995.

[OC, 93a] ***S. OLCOZ, J.M. COLOM***
" *Toward a formal Semantics of IEEE Std. VHDL 1076* "
Proceeding de EURO–VHDL/EURO_DAC '93 Hambourg; 20–24 sept 93, IEEE Computer Societed Press .

[OC, 93b] ***S. OLCOZ, J.M. COLOM***
" *A Petri Net approach for the analysis of VHDL description*"
Proc. CHARME'93 édité par Milne et Pierre dans "Lecture notes in Computer sciences 683: Correct Hardware Design and Verification Method". Springer Verlag mai 1993.

[Ol, 95a] ***S. OLCOZ***,
" *A formal model of VHDL Using Coloured Petri Nets*"
Formal semantics for VHDL, édité par K. Delgado Kloos et P. Breuer, Kluwer, Janv 1995.

[Ol, 95b] ***S. OLCOZ***,
" *A formal model of VHDL Using Coloured Petri Nets*"
Formal Method in System Design, vol 7, n 1–2, p 101–120, aout 1995, Kluwer.

[Par, 81] ***D. PARK***
" *Concurrency and automata on infinite Sequences* "
5th GI conference LNCS 104 Springer Verlag, Berlin 1981.

[Re, 94] ***S.READ***
" *Formal method for VLSI Design*"
Thèse de l'université de Manchester Department of Computation, 1994.

[RK, 95] ***R. REETZ, T. KROPF***
"*A Flowgraph Semantics of VHDL: Toward a VHDL Verification Workbench in HOL*"
Formal Method in System Design, vol 7, n 1–2, p 73–100, aout 1995, Kluwer.

[Ru, 95] ***D.M.RUSSINOFF***
" *A Formalization of a subset of VHDL in the Boyer–Moore logic*"
Formal Method in System Design, vol 7, n 1–2, p 7–25, aout 1995, Kluwer.

[Sa, 92] ***A. SALEM***
" *Verification formelle de circuits digitaux décrits en VHDL*".
Thèse de docteur informatique de l'université Joseph Fourier, Grenoble 1992.

[SGT, 92] ***B. SOULAS, JC. GEOFFROY, Mme TALLEC***
" *Toward an integrated approach to qualification of numerical systems*"
Int. Symposium on Nuclear Power Plant, Instrumentation and Control, Tokyo, mai 1992.

[So, 95] ***B. SOULAS***,
" *ASA+ : outil pour l'analyse comportementale des systèmes temps réél* ",
document EdF 1995.

[St, 93] ***V. STAVRIDOU***,
" *Formal method and VLSI engineering Practice",*
The Computer Journal, Vol 37 n 2 1994.

[Ta, 93] ***J. TANKOANO***
" Spécification du comportement des systèmes réactifs distribuables à l'aide de réseaux de Petri".
APPI vol 27 n2 1993 HERMES AFCET Editeurs.

[VLC, 93] ***S.T. VUONG , A. LOUREIRO, S. T. CHANSON***
" *A framework for the design for testability of Communication Protocols*"
6 th International IFIP Workshop on Protocol, Test Systems, Pau, France, septembre 1993.

[VT, 91] ***A. VALMARI, M. TIENARI***
" *An improved failures equivalence for FS systems with a reduction algorithm*"
11th Symposium on Protocol Specification Testing Specification, Testing and Verification, Int. IFIP WG 6.1, Stockolm, juin 1991.

[VT, 95] ***J. VAN TASSEL***
" *An operational semantics for a subset of VHDL.*"
In formal semantics for VHDL, édité par K. Delgado Kloos et P. Breuer, Kluwer, Janv 1995.

[Ve, 89] ***F. VERNADAT***
" *Vérification formelle d'application réparties, caractérisation logique d'une équivalence de comportement* "
Thèse de docteur de l'université Paul sabatier de Toulouse,1989.

[Ve, 94] ***VERILOG***
" *Présentation technique du produit ASA+*"
Note technique Verilog, octobre 1994.

[Wi, 93] ***M. WIRSING***
" Développement de logiciel et spécification formelle "
Technique et Science Informatiques, vol 12, p. 413–431, 1993.

L’exemple du DMA

ANNEXE 1 : LE MODULE DMA_CONTROL

Cette annexe a pour but d'illustrer le travail de spécification, de description et de traduction dans notre modèle sur un exemple simple. Nous avons pris volontairement un module non hiérarchique pour préciser les transformations subies par la spécification initiale en vue de sa validation à l'aide de l'outil ASA+ (cf chap3, figure 54).

1 – SPECIFICATION INITIALE DU MODULE DMA_CONTROL

Ce bloc est un sous module du module DMA. Il implante une machine d'états qui gère tout le comportement du DMA : en particulier la priorité de recherche dichotomique ID, le basculement sur les canaux (par transmission de COM_CHANNEL vers le Channel scheduler), et l'initialisation d'une opération sur le canal.

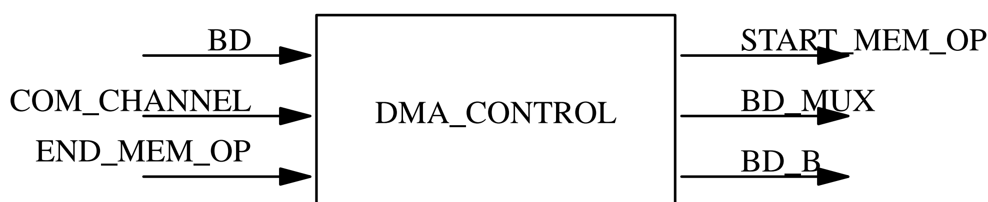


Figure 77 : Le diagramme d'interface du DMA_control

Les canaux sont identifiés par leur nom, leur protocole, le type de messages qu'ils véhiculent et leurs modules d'origine ou de destination sous forme de la table suivante :

NOM	Protocole	type	Source ou destination	divers
BD	RDV	Booléen	extérieur	flag de recherche dichotomique
END_MEM_OP	RDV	Booléen	MEMORY_INTERFACE	fin de transaction memoire
COM_CHANNEL	RDV	CHANNEL_OP _TYPE	channel_scheduler	opération de données sur un channel
START_MEM_OP	RDV	Booléen	MEMORY_INTERFACE	début de transaction mémoire
BD_MUX	VSIG_val	Booléen	MEMORY_INTERFACE	basculement pour le calcul d'adresses
BD_B	RDV	Booléen	BASE_INDEX_WORD	déclencheur du calcul d'adresse

Le comportement du module est fourni à un haut niveau de spécification, sous forme d'une description d'une machine d'états :

```
VAR     id_search, select, mem_op :  boolean;
        channel_com : CHANNEL_OP_TYPE;
        state, next_state : ( REPOS, GETRECID, MEM_ACCESS):

BEGIN   state=REPOS;
        loop
          id_search = false; channel_com = NOOP;
          case state of
             REPOS : multi–receive (BD, id_search; COM_CHANNEL, channel_com);
                     if id_search then
                       BD_MUX= true; next_state= GET_RECID;
                       send (BD_B, true);
                     elsif channel_com = NEW_NR then next_state = MEM_ACCESS;
                     else next_state = REPOS;
                     endif;
             GET_RECID : send (START_MEM_OP , true);
                         receive (END_MEM_OP, mem_op);
                         BD_MUX = false
                         next_state = REPOS;
              MEM_ACCESS : send (START_MEM_OP, true);
                         receive (END_MEM_OP, mem_op);
                         next_state =REPOS;
          end_case;
         state= next_state
        end_loop
END
```

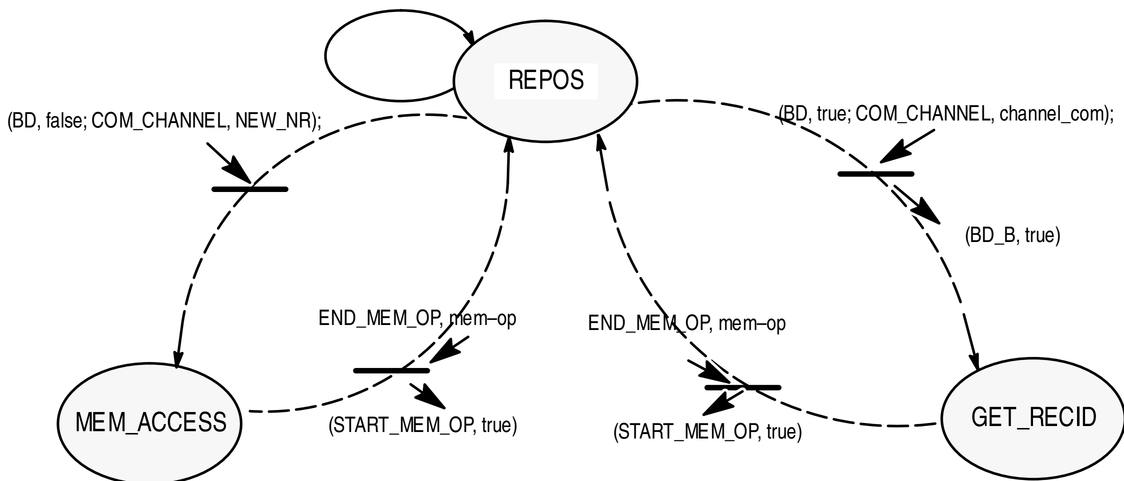


Figure 78 : Les états et les évènements de communications

2 – DESCRIPTION VOVHDL DU MODULE DMA_CONTROL

Ce fichier est constitué de la description VOVHDL de l'exemple du DMA

```
USE Work.Component_Pac.all ;

entity dma_control is
        port (       bd: in MRV boolean;
                     end_mem_op: in MRV boolean;
                     com_channel: in MRV CHANNEL_OP_TYPE;
                     start_meme_op: out MRV boolean;
                     bd_mux: out VSIG boolean;
                     bd_b: out MRV boolean);
end dma_control;


architecture behaviour of dma_control is

type state_ty is (REPOS, GET_RECID, MEM_ACCESS);

begin

  DMA_Control_process: process
        variable id_search, select, mem_op: boolean;
        variable channel_com: CHANNEL_OP_TYPE;
        variable state, next_state: state_ty;
        variable which: boolean_vector;

        begin
         state := REPOS;
         loop
          id_search := False;
          channel_com := CH_NOOP;
          case state is
                    when REPOS =>
                      multireceive(bd, id_search, com_channel, channel_com, which);
                       if id_search then
                              Send(bd_mux, True);
                              next_state := GET_RECID;
                              send(bd_b, True);
                       elsif channel_com=CH_NEW_AR then
                              next_state:=MEM_ACCESS;
                       else
                              next_state=REPOS;
                       endif;
                    when GET_RECID =>
                       send(start_mem_op, True);
                       receive(end_mem_op, mem_op);
                       send(bd_mux, False);
```

```
                        next_state=REPOS;
                      when MEM_ACCESS =>
                        send(start_mem_op, True);
                        receive(end_mem_op, mem_op);
                        next_state:=REPOS;
             end case;
             state := next_state;
            end loop;
  end process;
end dma_control;
```

3 – LE FICHIER C REALISANT LA PARTIE SPECIFIQUE DU MODULE DMA_CONTROL

Le fichier VOVHDL précédent a été traduit sous forme du fichier C suivant :

```
#include "com_define.h"
#include "component_pac.c"

#define ZZ_bd 0
#define ZZ_end_mem_op 1
#define ZZ_com_channel 2
#define ZZ_start_mem_op 3
#define ZZ_bd_mux 4
#define ZZ_bd_b 5

    typedef enum { repos, get_recid, mem_access }  state_ty;

    typedef struct {
  BOOLEAN  id_search;
  BOOLEAN  mem_op;
  channel_op_type  channel_com;
  state_ty  state;
  state_ty  next_state;
  struct { int val[ 2– 1+1]; }  rcv2;
} dma_control_behaviour_rec;

    typedef int data_in_out_type[NBR_PORT_MAX];

    void dma_control_behaviour_proc(prog_counter, data_rec, data_in, data_out)

     int  *prog_counter;
 data_in_out_type  *data_out, *data_in;
 dma_control_behaviour_rec  *data_rec;
{int ZZ_i;

    ZZ_i = 0;
goto label_start;
label0 :
  data_rec–>state = repos;
  (*data_out)[ZZ_bd_mux] = FALSE;
  Send( 1, label1 );
  while ( TRUE ) {
    switch ( data_rec–>state ) {
     case repos :
       (*data_out)[ZZ_bd] = 1;
       (*data_out)[ZZ_com_channel] = 1;
       MultiReceiveW( 2, label2, data_rec–>rcv2 );
       data_rec–>id_search = (*data_in)[ZZ_bd];
       data_rec–>channel_com = (*data_in)[ZZ_com_channel];
       if ( data_rec–>id_search == TRUE ) {
         (*data_out)[ZZ_bd_mux] = TRUE;
         Send( 3, label3 );
         (*data_out)[ZZ_bd_b] = TRUE;
```

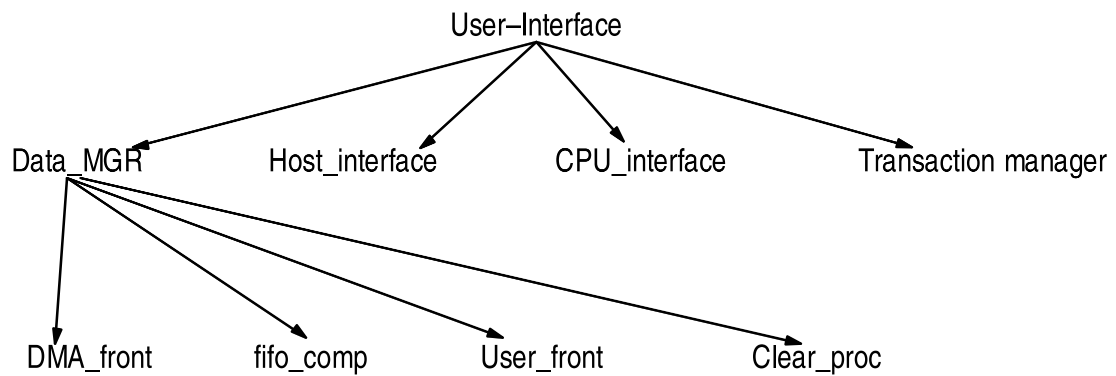
```
        Send( 4, label4 );
        data_rec–>next_state = get_recid; }
      else if ( data_rec–>channel_com == ch_new_nr )
       { data_rec–>next_state = mem_access; }
      else {data_rec–>next_state = repos; }
      break;
    case get_recid :
      (*data_out)[ZZ_start_mem_op] = TRUE;
      Send( 5, label5 );
      (*data_out)[ZZ_end_mem_op] = 1;
      Receive( 6, label6 );
      data_rec–>mem_op = (*data_in)[ZZ_end_mem_op];
      (*data_out)[ZZ_bd_mux] = FALSE;
      Send( 7, label7 );
              if ( data_rec–>channel_com == ch_new_nr ) {
        data_rec–>next_state = mem_access; }
     else {data_rec–>next_state = repos;}
     break;
    case mem_access :
      (*data_out)[ZZ_start_mem_op] = TRUE;
      Send( 8, label8 );
      (*data_out)[ZZ_end_mem_op] = 1;
      Receive( 9, label9 );
      data_rec–>mem_op = (*data_in)[ZZ_end_mem_op];
      data_rec–>next_state = repos;
      break;
    default : break; }
   data_rec–>state = data_rec–>next_state; }
  goto label0;
label_start :
  switch( *prog_counter )
  { case 0    : goto label0;
    case 1    : goto label1;
    case 2    : goto label2;
    case 3    : goto label3;
    case 4    : goto label4;
    case 5    : goto label5;
    case 6    : goto label6;
    case 7    : goto label7;
    case 8    : goto label8;
    case 9    : goto label9;
   default    : printf(”<<<Program Counter Out Of Sequence in dma_control_behaviour_proc>>>\n”);}}
```

Un exemple plus complexe et hiérarchique : le module ″Interface utilisateur″

L'objectif de cette annexe est d'illustrer notre travail de modélisation dans le cas d'un module complexe décomposé de manière structurelle en sous–modules

Nous montrons les différentes étapes permettant de passer des données initiales fournies par le constructeur jusqu'au modèle ASA+ vérifiable.

La structure du modèle est la suivante:



1– LA DESCRIPTION DU CONSTRUCTEUR

Cette partie est constituée des données telles que nous les fournissent les concepteurs du circuit.

Chaque module est défini de manière graphique et un module terminal est décrit de manière comportementale dans un langage algorithmique (les descriptions algorithmiques ne sont pas fournies ici). Les canaux de communication sont clairement identifiés.

2– LA DESCRIPTION VOVHDL ASSOCIEE

Chacun des modules définis par les concepteurs dans la première partie de cette annexe est décrit selon le langage VOVHDL. Nous fournissons ici les modèles VOVHDL du module data_MGR et de ses quatre sous modules : Clear_proc, User_front, Fifo_comp, DMA_front.

```
USE Work.User_Interface_pac.all ;

entity data_manager is
port (          dma_com: in MRV DOP_TYPE;
                dma_out: in VSIG GENERIC_DATA_TYPE;
                var_desc: in VSIG GENERIC_DATA_TYPE;
                fifo_ctrl: in VSIG FIFO_CTRL_TYPE;
                to_fifo: in VSIG GENERIC_DATA_TYPE;
                user_move: in MRV FIFO_ACCESS;
                clear_fifo: in MRV CLEAR_FIFO_TYPE;
                cpu_ctrl: in MRV CPU_CTRL_TYPE;
                fifo_ready: out VSIG boolean;
                dma_in: out VSIG GENERIC_DATA_TYPE;
                from_fifo: out VSIG GENERIC_DATA_TYPE;
                ustat: out VSIG USTAT_TYPE;
                block_in_progress: out MRV boolean;
                fr: out VSIG boolean );
end data_manager;

architecture structure of data_manager is

channel OUT_FIFO: GENERIC_DATA_TYPE, VSIG, one(fifo(out_fifo)),
one(dma_frt(out_fifo), user_frt(out_fifo));
channel IN_FIFO: GENERIC_DATA_TYPE, VSIG, one(dma_frt(in_fifo),
user_frt(in_fifo)), one(fifo(in_fifo));
channel FIFO_COM: FIFO_COM_TYPE, MRV, one(dma_frt(fifo_com),
user_frt(fifo_com), clr_proc(fifo_com)), one(fifo(fifo_com));

component clear_proc
port (          clear_fifo: in MRV CLEAR_FIFO_TYPE;
                fifo_com: out MRV FIFO_COM_TYPE );
end component;

component user_front
port (          to_fifo: in VSIG GENERIC_DATA_TYPE;
                out_fifo: in VSIG GENERIC_DATA_TYPE;
                user_move: in MRV FIFO_ACCESS;
                from_fifo: out VSIG GENERIC_DATA_TYPE;
                fifo_com: out MRV FIFO_COM_TYPE;
                in_fifo: out VSIG GENERIC_DATA_TYPE);
end component;

component dma_front
port (          dma_out: in VSIG GENERIC_DATA_TYPE;
                out_fifo: in VSIG GENERIC_DATA_TYPE;
                dma_com: in MRV DOP_TYPE;
                dma_in: out VSIG GENERIC_DATA_TYPE;
                fifo_com: out MRV FIFO_COM_TYPE;
                in_fifo: out VSIG GENERIC_DATA_TYPE);
end component;

component fifo_comp
port (          fifo_com: in MRV FIFO_COM_TYPE;
                in_fifo: in VSIG GENERIC_DATA_TYPE;
                var_desc: in VSIG GENERIC_DATA_TYPE;
                fifo_ctrl: in VSIG FIFO_CTRL_TYPE;
                cpu_ctrl: in MRV CPU_CTRL_TYPE;
                fr: out VSIG boolean;
                fifo_ready: out VSIG boolean;
                out_fifo: out VSIG integer;
                block_in_progress: out MRV boolean );
end component;

begin

clr_proc : clear_proc
port map( clear_fifo, FIFO_COM);

dma_frt : dma_front
port map( dma_out, OUT_FIFO, dma_com, dma_in, FIFO_COM, IN_FIFO );

fifo : fifo_comp
port map( FIFO_COM, IN_FIFO, var_desc, fifo_ctrl, cpu_ctrl, fr, fifo_ready,
OUT_FIFO, block_in_progress );

user_frt : user_front
port map( to_fifo, OUT_FIFO, user_move, from_fifo, FIFO_COM, IN_FIFO );

end structure;
```

```
USE Work.User_Interface_pac.all ;

entity clear_proc is
port (          clear_fifo: in MRV CLEAR_FIFO_TYPE;
                fifo_com: out MRV FIFO_COM_TYPE );
end clear_proc;

architecture behaviour of clear_proc is
begin
  CLEAR_process:
  process
  variable COM: CLEAR_FIFO_TYPE;
  begin
                loop
                  receive( clear_fifo, COM );
                  if COM = CL_FIFO_RAZ then send(fifo_com, FCM_RAZ_FIFO);
                end loop;
  end process;
end behaviour;

USE Work.User_Interface_pac.all ;

entity user_front is
port (          to_fifo: in VSIG GENERIC_DATA_TYPE;
                out_fifo: in VSIG GENERIC_DATA_TYPE;
                user_move: in MRV FIFO_ACCESS;
                from_fifo: out VSIG GENERIC_DATA_TYPE;
                fifo_com: out MRV FIFO_COM_TYPE;
                in_fifo: out VSIG GENERIC_DATA_TYPE);
end user_front;

architecture behaviour of user_front is
begin
  USER_process:
  process
  variable MOVE_COMMAND: FIFO_ACCESS;
  variable V_OUT_FIFO: GENERIC_DATA_TYPE;
  variable V_TO_FIFO: GENERIC_DATA_TYPE;
  begin
                loop
                  receive( user_move, MOVE_COMMAND);
                  case MOVE_COMMAND is
                              when F_READ_FIFO =>
                                send(fifo_com, FCM_READ_FIFO);
                                receive(out_fifo, V_OUT_FIFO);
                                send(from_fifo, V_OUT_FIFO);
                              when F_WRITE_FIFO =>
                                receive(to_fifo, V_TO_FIFO);
                                send(in_fifo, V_TO_FIFO);
                                send(fifo_com, FCM_WRITE_FIFO);
                              when others =>
                  end case;
                end loop;
  end process;
end behaviour;

USE Work.User_Interface_pac.all ;

entity dma_front is
port (          dma_out: in VSIG GENERIC_DATA_TYPE;
                out_fifo: in VSIG GENERIC_DATA_TYPE;
                dma_com: in MRV DOP_TYPE;
                dma_in: out VSIG GENERIC_DATA_TYPE;
                fifo_com: out MRV FIFO_COM_TYPE;
                in_fifo: out VSIG GENERIC_DATA_TYPE);
end dma_front;

architecture behaviour of dma_front is
begin
  DMA_process:
  process
  variable DMA_COMMAND: DOP_TYPE;
  variable V_OUT_FIFO: GENERIC_DATA_TYPE;
  variable V_DMA_OUT: GENERIC_DATA_TYPE;
  begin
                loop
                  receive( dma_com, DMA_COMMAND);
                  case DMA_COMMAND is
                              when D_READ_FIFO=>
                                send(fifo_com, FCM_READ_FIFO);
                                receive(out_fifo, V_OUT_FIFO);
                                send(dma_in, V_OUT_FIFO);
                              when D_WRITE_FIFO =>
```

```
                        receive(dma_out, V_DMA_OUT);
                        send(in_fifo, V_DMA_OUT);
                        send(fifo_com, FCM_WRITE_FIFO);
                  end case;
                end loop;
   end process;
end behaviour;

USE Work.User_Interface_pac.all ;

entity fifo_comp is
                port (  fifo_com: in MRV FIFO_COM_TYPE;
                        in_fifo: in VSIG GENERIC_DATA_TYPE;
                        var_desc: in VSIG GENERIC_DATA_TYPE;
                        fifo_ctrl: in VSIG FIFO_CTRL_TYPE;
                        cpu_ctrl: in MRV CPU_CTRL_TYPE;
                        fr: out VSIG boolean;
                        fifo_ready: out VSIG boolean;
                        out_fifo: out VSIG GENERIC_DATA_TYPE;
                        block_in_progress: out MRV boolean );
end fifo_comp;

architecture behaviour of fifo_comp is

   const FSIZE = 3;
   type TABEL: is array (1 to FSIZE) of GENERIC_DATA_TYPE;
   procedure DECAL_FIFO (variable X: TABEL);
    function func(variable P, SIZE, count: integer; variable X: FIFO_CTRL_TYPE; Y:
boolean) return boolean;

   procedure DECAL_FIFO (variable X: TABEL) is
                variable I: integer;
                begin
                  for I in 0 to FSIZE–1 loop
                        X(I) := X(i+1);;
                  end loop;
                  X(FSIZE) := G_NS;
   end DECAL_FIFO;

    function func(variable P, SIZE, count: integer; variable X: FIFO_CTRL_TYPE; Y:
boolean) return boolean is
```

```
      variable output: boolean;
                begin
                if Y then output = ((P>0) and (X = FCT_ECRIT_FIFO))
                   else output = (((P = (SIZE + 1)) and (X = FCT_ECRIT_FIFO)) or
((P>0) and (X = FCT_LIT_FIFO)));
                end if;
                return output;
   end func;

begin

   FIFO_process:
   process
   variable FIFO_TAB: TABEL;
   variable I,P,count: integer;
   variable progress: boolean;
   variable FIFO_COMMAND: FIFO_COM_TYPE;
   variable X: CPU_CTRL_TYPE;
   variable V_VAR_DESC: GENERIC_DATA_TYPE;
   variable V_FR: boolean;
   variable V_IN_FIFO: GENERIC_DATA_TYPE;
   variable V_FIFO_CTRL: FIFO_CTRL_TYPE;
   variable V_FIFO_READY: boolean;

   begin
                loop
                   P := 0;
                   for I in 0 to FSIZE loop
                        FIFO_TAB(I) := G_NS;
                   end loop;

                   repeat1: loop
                        receive(cpu_ctrl, X);
                        exit repeat1 when (X=C_VAR_DESC_SENT);
                   end loop repeat1;

                   receive(var_desc, V_VAR_DESC);
                   count := To_integer(V_VAR_DESC);

                   repeat2: loop
                        receive(fifo_com, FIFO_COMMAND);
                        case FIFO_COMMAND is
```

```
                    when FCM_RAZ_FIFO =>
                    when FCM_READ_FIFO =>
                              if ( P>0 ) then
                                send(out_fifo,FIFO_TAB(1));
                                DECAL_FIFO (FIFO_TAB);;
                                P := P – 1;
                                count := count – 1;
                              else
                                send(out_fifo,G_NS);
                              end if;
                              receive(fifo_ctrl, V_FIFO_CTRL);
                              V_FR    :=    (    (P>0)    and
(V_FIFO_CTRL=FCT_LIT_FIFO) );
                              send(fr, V_FR);
                              progress := (count /= 0);
                              send(block_in_progress, progress);
                    when FCM_WRITE_FIFO =>
                              if ( P < FSIZE ) then
                                P := P + 1;
                                receive(in_fifo, V_IN_FIFO);
                                FIFO_TAB(P) := V_IN_FIFO;
                              end if;
                              receive(fifo_ctrl, V_FIFO_CTRL);
                              V_FR   :=   (   (P=(FSIZE+1))   and
(V_FIFO_CTRL=FCT_ECRIT_FIFO) );
                              send(fr, V_FR);
                  end case;
                  receive(fifo_ctrl, V_FIFO_CTRL);
                  V_FIFO_READY   :=   funct(P,   FSIZE,   count,
V_FIFO_CTRL, count/=0);
                  exit repeat2 when (count=0 or FIFO_COMMAND =
FCM_RAZ_FIFO);
          end loop repeat2;
        end loop;
  end process;
end behaviour;
```

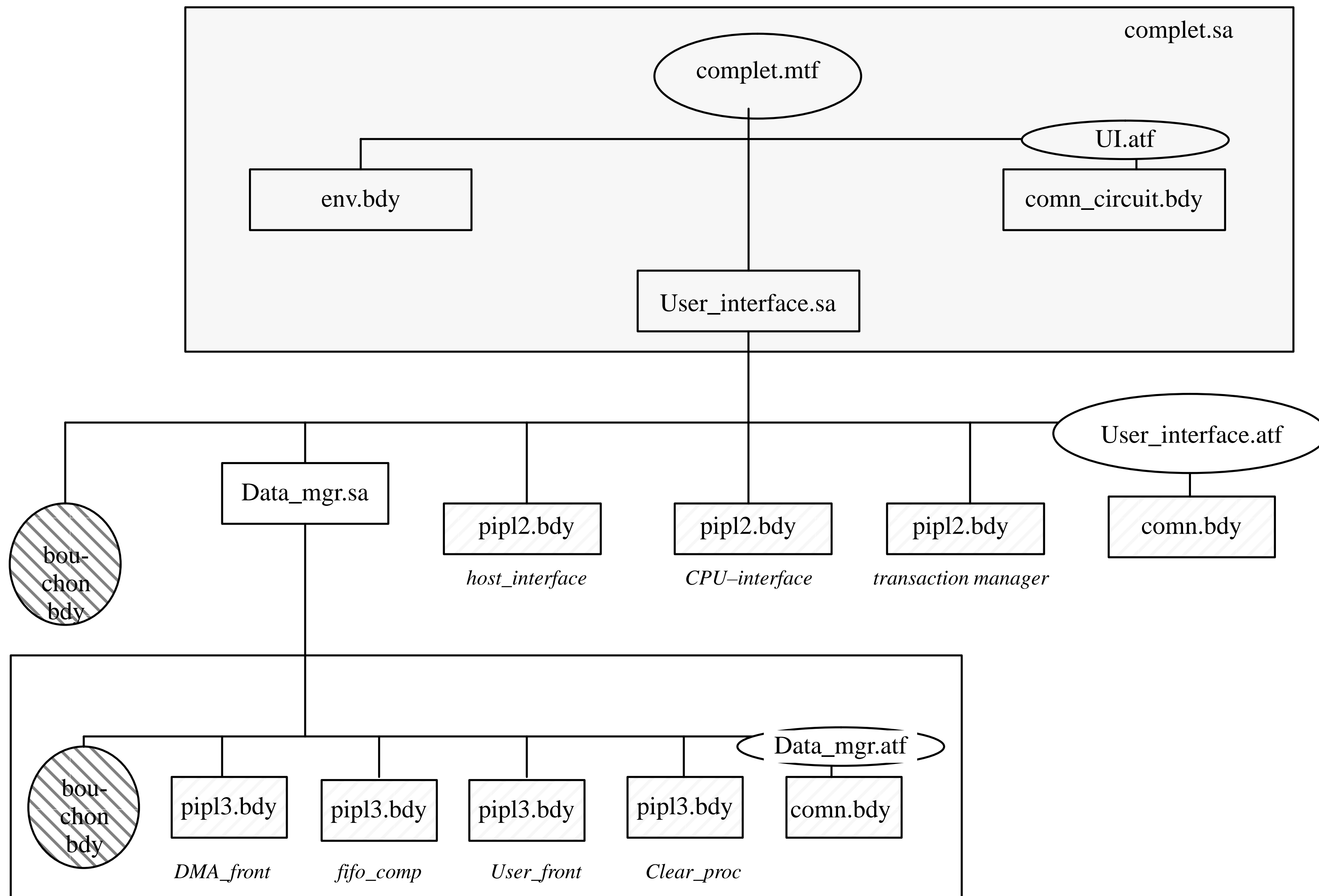
3– MODELISATION : LA STRUCTURE

La première page de cette partie donne une description structurelle de notre modèle. Les pages suivantes donnent la représentation graphique telle que le fournit l'outil ASA+. Les fichiers .sa donnent les mêmes descriptions mais sous forme du langage LSA+. Nous donnerons le fichier complet.sa.

Les fichiers suffixés par mtf contiennent la définition de la base de donnée du modèle complet.

Ceux suffixés par bdy contiennent la partie comportementale du module.

Les fichiers suffixés par atf contiennent la base de donnée propre au module.


complet.sa
complet.mtf
UI.atf
env.bdy
comn_circuit.bdy
User_interface.sa
User_interface.atf
bou-
chon
bdy
Data_mgr.sa
pipl2.bdy
host_interface
pipl2.bdy
CPU–interface
pipl2.bdy
transaction manager
comn.bdy
Data_mgr.atf
bou-
chon
bdy
pipl3.bdy
DMA_front
pipl3.bdy
fifo_comp
pipl3.bdy
User_front
pipl3.bdy
Clear_proc
comn.bdy

LE MODULE COMPLET.SA

```
DEFAULT RENDEZ_VOUS[*];

#include 'complet.mtf'

MODULE complet_mod;
IP
        INPUT user_input1 ;
        OUTPUT user_output1 ;
END; (* MODULE  *)

BODY complet_bod FOR complet_mod;

MODULE  (*$ ENVIRONMENT *) environment_mod;
IP
 INPUT entree_comp1 : (message(entree: type_entree));
 OUTPUT sortie_comp1 : (message(sortie: type_sortie));
  INPUT from_comn : (mess(tab_res: type_acti_res_tab));
  OUTPUT  to_pipe_circuit : (mess(tab_act: type_acti_res_tab));

END; (* MODULE  *)

BODY environment_bod FOR environment_mod;
        #include 'env.bdy'
END; (* body environment_bod *)

#include 'User_interface.sa'
MODULE  comn_circuit_mod (num_com : indice_mod);
 IP
   INPUT     int_com_in : (mess(tab_act: type_acti_res_tab)) ;
  OUTPUT     int_com_out: (mess(tab_res:type_acti_res_tab)) ;
END; (* MODULE  *)

BODY comn_circuit_bod FOR comn_circuit_mod;
        #include 'UI.atf'
        #include 'comn_circuit.bdy'
END; (* body comn_circuit_bod *)

MODVAR (* de complet *)
        environment : environment_mod;
        circuit : circuit_mod;
        comn_circuit : comn_circuit_mod;

INITIALIZE (* de circuit *)

  BEGIN
        INIT environment WITH environment_bod;
        INIT circuit WITH circuit_bod;
        INIT comn_circuit WITH comn_circuit_bod(0);
ATTACH user_input1 TO environment.entree_comp1;

ATTACH environment.sortie_comp1 TO user_output1;

CONNECT environment.to_pipe_circuit TO circuit.pipe;

CONNECT circuit.activity_out TO comn_circuit.int_com_in;
CONNECT comn_circuit.int_com_out TO environment.from_comn;
CONNECT comn_circuit.int_com_out TO circuit.result_in;

  END; (*  initialize  *)

END; (* body complet_bod *)

MODVAR
        complet : complet_mod;

INITIALIZE
BEGIN

  INIT complet WITH complet_bod;
END; (*  initialize  *)
END.
```

4– MODELISATION : LE COMPORTEMENT

Il s’agit ici de montrer sous quelle forme est décrit le comportement de notre modèle.

Le fichier .mtf contient la base de données.

Les fichiers .bdy contiennent la description comportementale des processus .pipl.bdy pour les processus définis par le concepteurs, comn.bdy pour le module de communication.

Nous donnons dans cette partie un fichier pipl.bdy. Celui_ci est constitué de fonctions pascal utilisées dans le corps des transitions qui représentent le modèle STE générique d’un module VOVHDL. Ces transitions sont définies en fin de fichier.

LE FICHIER PIPL2.BDY

1– Partie définition de fonctions et procédures

```
STATE e1, e2;
VAR
        tab_act_out: type_acti_res_tab;
        prog_counter : INTEGER;
        data_in , data_out: type_acti_res;
        data_rec : type_data_rec;

   PROCEDURE host_interface_behaviour_proc (VAR prog_counter: INTEGER;
VAR data_rec: type_data_rec; VAR data_in :type_acti_res; VAR data_out :
type_acti_res); C PRIMITIVE;

   PROCEDURE cpu_interface_behaviour_proc (VAR prog_counter: INTEGER;
VAR data_rec: type_data_rec; VAR data_in : type_acti_res; VAR data_out :
type_acti_res); C PRIMITIVE;

   PROCEDURE transaction_manager_behaviour_proc (VAR prog_counter:
INTEGER; VAR data_rec: type_data_rec; VAR data_in : type_acti_res; VAR data_out
: type_acti_res); C PRIMITIVE;

   (*————————————————————————————————*)
PROCEDURE appel_de_proc_C (VAR prog_counter: INTEGER; VAR data_rec:
type_data_rec; VAR data_in :type_acti_res; VAR data_out : type_acti_res) ;

   BEGIN
 IF num_mod = 2 THEN
 host_interface_behaviour_proc (prog_counter,data_rec, data_in, data_out);
 IF num_mod = 3 THEN
 cpu_interface_behaviour_proc (prog_counter,data_rec, data_in, data_out);
 IF num_mod = 4 THEN
 transaction_manager_behaviour_proc (prog_counter,data_rec, data_in, data_out);
END;

   (*————————————————————————————————*)
FUNCTION blocage(tab_res : type_acti_res_tab; num_mod : INTEGER) :
type_ok_nok;

    VAR bloc : type_ok_nok;
  j : INTEGER;
   BEGIN
        bloc := ok;
        FOR j := 1 TO NBR_PORT_MAX DO
                IF tab_res[num_mod][j]= 0 THEN
                bloc :=nok;*)
blocage := bloc;
END;
(* si l un au moins des ports est bloquant alors blocage *)
(*****************************************************************************)

INITIALIZE TO e1

   VAR J,K : INTEGER;
 BEGIN
        FOR J := 1 TO NBR_PROC_MAX DO
                FOR K := 1 TO NBR_PORT_MAX DO
                        tab_act_out[J,K] :=0 ;
        FOR K := 1 TO NBR_PORT_MAX DO data_in[K] :=0;
        FOR K := 1 TO NBR_PORT_MAX DO data_out[K] :=0;
        prog_counter :=0;
        appel_de_proc_C (prog_counter, data_rec, data_in, data_out);
 END;
```

2– la partie définition des transitions

```
   TRANS
        WHEN pipe.mess(tab_act)
        FROM e1
        TO e2
 BEGIN  tab_act_out := tab_act;
        tab_act_out[num_mod] := data_out;
        OUTPUT activity_out.mess(tab_act_out)
 END;
(* on transmet le pipe et l activity de la boite*)
   TRANS
        WHEN result_in.mess(tab_res)
        FROM e2
        PROVIDED blocage(tab_res,num_mod) = nok
```

```
                TO e2
BEGIN
END ;

    TRANS
                WHEN result_in.mess(tab_res)
                FROM e2
                PROVIDED  blocage(tab_res,num_mod) = ok
                TO e1

    BEGIN data_in := tab_res[num_mod];
    appel_de_proc_C(prog_counter,data_rec, data_in, data_out);
   END;
```

5– MODELISATION : LE MODULE DE COMMUNICATION

Ce module est un module comportemental donc suffixé par .bdy. Il est fixé quelquesoit le circuit. Il contient les définitions des règles de communication et celle des protocoles. C'est ce qui est défini dans une première partie du fichier. De même que pour le module générique d'un processus (presenté précedemment), les transitions sont définies en fin de fichier.

```
STATE SIM1, SIM2, SIM3;

VAR
tab_res_out, tab_act_out, tab_res_int : type_acti_res_tab;
data_out_sup : type_acti_res;
BUFF_IN_VECT : type_buff_in_vect;
TTAB_CH_PORT : type_tab_ch_port;
TTAB_EXT_INT : type_tab_ext_int;


val_choisie, valeur_emise: INTEGER;
liste_emetteur,liste_em_actif,liste_em_effect,liste_em_bloc: type_liste_em;
liste_recepteur,liste_rec_actif, liste_rec_effect, liste_rec_pred, liste_rec_bloc :
type_liste_rec;
liste_rec_pred_tab : type_liste_rec_tab;
CRule, DRule: type_rule;
CRule_bloc, DRule_bloc : type_ok_nok;
protocole : type_proto;

FUNCTION r_rand(X : INTEGER): INTEGER; C PRIMITIVE;

PROCEDURE srand(X:INTEGER); C PRIMITIVE;

(* declaration locale d une fonction C *)

FUNCTION set_alea (X : INTEGER): INTEGER;
BEGIN
  set_alea := r_rand(X);
END;

(************************************************************************)
PROCEDURE initialisation (VAR BUFF_IN : INTEGER; VAR tab_res_int
:type_acti_res_tab; VAR data_out_sup : type_acti_res);

VAR i,j :INTEGER;

BEGIN
        BUFF_IN := 0;
        FOR i := 1 TO NBR_PROC_MAX DO
                FOR j := 1 TO NBR_PORT_MAX DO
                        tab_res_int [i,j] := 0;
        FOR j := 1 TO NBR_PORT_MAX DO
                data_out_sup[j] :=0;

END;
```

Verification de CR

```
FUNCTION comparable_CR(L1,L2 : type_liste_em): BOOLEAN;

   VAR comparable_aux : BOOLEAN;
       i : INTEGER;

   BEGIN
       comparable_aux := TRUE;
       FOR i := 1 TO NBR_EM_MAX DO
        IF ((L1[i].num_proc <> L2[i].num_proc) OR
                (L1[i].num_port <> L2[i].num_port))
          THEN comparable_aux := FALSE;
       comparable_CR := comparable_aux;
END;

   (*==============================================*)
  PROCEDURE meme_val(tab_act : type_acti_res_tab; liste_em_actif:
type_liste_em; VAR valeur_emise : INTEGER; VAR CRule_bloc: type_ok_nok);

   VAR j, proc_val, port_val , val, val_pred : INTEGER;

       BEGIN
        proc_val := liste_em_actif[1].num_proc;
        port_val:= liste_em_actif[1].num_port;
        val_pred := tab_act[proc_val, port_val];
        CRule_bloc :=ok;
        j :=1;
        REPEAT
                j := j+1;
                proc_val := liste_em_actif[j].num_proc;
                port_val := liste_em_actif[j].num_port;
                val := tab_act[proc_val, port_val];
            IF (val <> val_pred) THEN CRule_bloc:=nok;
        UNTIL (( j = NBR_EM_MAX) OR (CRule_bloc =nok));
```

```
                IF (CRule_bloc =ok) THEN
                         valeur_emise := val_pred
                         ELSE valeur_emise :=0;
               END;
(*===============================================*)
```

PROCEDURE verifier_CR (CRule: type_rule; liste_emetteur, liste_em_actif: type_liste_em; tab_act: type_acti_res_tab; VAR valeur_emise : INTEGER; VAR CRule_bloc:type_ok_nok; VAR liste_em_effect : type_liste_em; VAR liste_em_bloc : type_liste_em);

```
   VAR j, port_aux_cr, proc_aux_cr,compt, compt2, Num_em : INTEGER;

   BEGIN
 IF CRule = tous THEN
               BEGIN
               liste_em_bloc := liste_em_actif;
               IF comparable_CR(liste_emetteur,liste_em_actif)=TRUE
                THEN
                 BEGIN
                  meme_val(tab_act, liste_em_actif, valeur_emise, CRule_bloc);
                  liste_em_effect := liste_em_actif;
                 END
                 ELSE
                 BEGIN
                  CRule_bloc := nok;
                  valeur_emise := 0;
                 END;
               END;

   IF CRule = one THEN
               BEGIN
                compt :=0;
                FOR j := 1 TO NBR_EM_MAX DO
                   IF (liste_em_actif[j].num_proc <> 0) THEN
                         compt := compt + 1;
                IF (compt = 0) THEN
                         BEGIN
                         CRule_bloc := nok;
                         valeur_emise :=0;
                         liste_em_bloc := liste_em_actif;
                         END
                         ELSE
                         BEGIN
                          CRule_bloc := ok;
                          Num_em := set_alea(compt);
                          compt2 :=0;
                          FOR j := 1 TO NBR_EM_MAX DO
                           BEGIN
                                 IF  (liste_em_actif[j].num_proc <>  0)
THEN
                          BEGIN
                          compt2 := compt2 +1;
                           IF (compt2 = Num_em) THEN
                   BEGIN
                   liste_em_bloc[j].num_proc :=0;
                   liste_em_bloc[j].num_port :=0;
                   proc_aux_cr := liste_em_actif[j].num_proc;
                   port_aux_cr := liste_em_actif[j].num_port;
                   valeur_emise := tab_act[proc_aux_cr, port_aux_cr];
                   liste_em_effect[j].num_proc := proc_aux_cr;
                   liste_em_effect[j].num_port := port_aux_cr;
                   END
                   ELSE
                   BEGIN
                   liste_em_bloc[j] := liste_em_actif[j];
                   liste_em_effect[j].num_proc := 0;
                   liste_em_effect[j].num_port := 0;
                   END
                     END
                           ELSE
                           BEGIN
                         liste_em_bloc[j] := liste_em_actif[j];
                         liste_em_effect[j].num_proc := 0;
                         liste_em_effect[j].num_port := 0;
                           END;
                     END;
                 END;
              END;
```

```
    IF CRule = some THEN
BEGIN
            compt :=0;
            FOR j := 1 TO NBR_EM_MAX DO
              IF (liste_em_actif[j].num_proc <> 0) THEN
                        compt := compt + 1;
            IF (compt = 0 )THEN
            BEGIN
            CRule_bloc := nok;
            valeur_emise := 0;
            liste_em_bloc := liste_em_actif;
            liste_em_effect := liste_em_actif;
            END
            ELSE
            BEGIN
             CRule_bloc := ok;
             Num_em := set_alea(compt);
            compt2 :=0;
             FOR j := 1 TO NBR_EM_MAX DO
              BEGIN
               IF (liste_em_actif[j].num_proc <> 0) THEN
                        BEGIN
                        compt2 := compt2 +1;
                        IF (compt2 = Num_em) THEN
            BEGIN
            proc_aux_cr := liste_em_actif[j].num_proc;
            port_aux_cr := liste_em_actif[j].num_port;
            valeur_emise := tab_act[proc_aux_cr, port_aux_cr];
            END;
                        END;
              END;
             FOR j := 1 TO NBR_EM_MAX DO
              BEGIN
               proc_aux_cr := liste_em_actif[j].num_proc;
               port_aux_cr := liste_em_actif[j].num_port;
               IF (proc_aux_cr = 0 ) THEN
                        BEGIN
                        liste_em_bloc[j] := liste_em_actif[j];
                        liste_em_effect[j] := liste_em_actif[j];
                        END;
               IF (tab_act[proc_aux_cr,port_aux_cr] = valeur_emise)
                        THEN
                        BEGIN
                         liste_em_effect[j] := liste_em_actif[j];
                         liste_em_bloc[j] := liste_em_actif[j];
                        END
                        ELSE
                        BEGIN
                         liste_em_effect[j].num_proc := 0;
                         liste_em_effect[j].num_port := 0;
                         liste_em_bloc[j] := liste_em_actif[j];
                        END;
              END;
             END;
    END;
END;(*procedure verifer_CR*)
```

VERIFICATION de DR

Procedure non incluse

FONCTION UTILES POUR LA DEFNITION DES PROTOCOLES

Fonctions non incluses

PROCEDURE SIMULATION PROPREMENT DITE

```
    PROCEDURE simulation (tab_act : type_acti_res_tab; VAR data_out_sup :
type_acti_res; VAR tab_res_int : type_acti_res_tab; VAR liste_rec_pred_tab :
type_liste_rec_tab; VAR BUFF_IN_VECT :type_buff_in_vect );

    VAR
i,k,l : INTEGER;
BUFF_IN, num_rec, num_em : INTEGER;
proc_aux, port_aux, proc_rec, port_rec, proc_em, port_em : INTEGER;

    BEGIN
(* initialisation des variables *)

    initialisation(BUFF_IN, tab_res_int, data_out_sup);
```

```
      (*=============================================*)
(* calcul du DATA_OUT_SUP *)
FOR i := 1 TO NBR_PORT_MAX DO
BEGIN
                proc_aux := TTAB_EXT_INT[i].num_proc;
                port_aux := TTAB_EXT_INT[i].num_port;
                IF (proc_aux = 0 )
                THEN data_out_sup[i] := 0
                ELSE
                data_out_sup[i] := tab_act[proc_aux, port_aux]
                END;

      (*=============================================*)
(* procedure simulation proprement dite *)

      FOR k := 1 TO NBR_CH_MAX DO
 BEGIN
  (* recuperer les proprietes du channel *)

                liste_emetteur := TTAB_CH_PORT[k].list_em;
                liste_recepteur := TTAB_CH_PORT[k].list_rec;
                CRule := TTAB_CH_PORT[k].CR;
                DRule := TTAB_CH_PORT[k].DR;
                protocole := TTAB_CH_PORT[k].proto;

     (* construire la liste des emetteurs actifs du channel *)

     FOR l :=1 TO NBR_EM_MAX DO
                BEGIN
                 proc_em :=liste_emetteur[l].num_proc;
                 port_em :=liste_emetteur[l].num_port;
                 IF ((proc_em <> 0) AND (port_em <> 0) AND
                   (tab_act[proc_em, port_em] <>0)) THEN
                                BEGIN
                                 liste_em_actif[l].num_proc := proc_em;
                                 liste_em_actif[l].num_port := port_em;
                                END
                                ELSE
                                BEGIN
                                 liste_em_actif[l].num_proc := 0;
                                 liste_em_actif[l].num_port := 0;
                                END
                END; (* sur liste_em_actif *)

      (* construire la liste des recepteurs actifs du channel *)
 FOR l :=1 TO NBR_REC_MAX DO
                BEGIN
                 proc_rec :=liste_recepteur[l].num_proc;
                 port_rec :=liste_recepteur[l].num_port;

                 IF ((proc_rec <> 0) AND (port_rec <> 0) AND
                                (tab_act[proc_rec, port_rec] <>0)) THEN
                                BEGIN
                                 liste_rec_actif[l].num_proc := proc_rec;
                                 liste_rec_actif[l].num_port := port_rec;
                                END
                                ELSE
                                BEGIN
                                 liste_rec_actif[l].num_proc := 0;
                                 liste_rec_actif[l].num_port := 0;
                                END
                END; (* sur liste_rec_actif *)

      (*=============================================*)
IF protocole = MRV THEN
  BEGIN
    verifier_CR (CRule,liste_emetteur, liste_em_actif, tab_act, valeur_emise,
CRule_bloc, liste_em_effect,liste_em_bloc);
  verifier_DR (DRule,liste_recepteur, liste_rec_actif, DRule_bloc, liste_rec_effect,
liste_rec_bloc);
 IF ( (CRule_bloc <> ok) OR (DRule_bloc <> ok)) THEN
locage_em_rec(tab_res_int, liste_em_bloc, liste_rec_bloc)
ELSE
  emettre (tab_res_int, liste_em_effect,liste_rec_effect, valeur_emise);
END;
(*=============================================*)
IF protocole = VSIG_val THEN
                BEGIN
                 BUFF_IN := BUFF_IN_VECT[k];
                 emettre (tab_res_int, liste_em_actif,liste_rec_actif, BUFF_IN);
                 choix_val_in ( tab_act, liste_em_actif, val_choisie);
                 IF (val_choisie <> 0) THEN  BUFF_IN_VECT[k] := val_choisie ;
                END;
```

```
(*==================================================*)
IF protocole = VSIG_evt THEN
            BEGIN
             BUFF_IN := BUFF_IN_VECT[k];
             num_rec := compter_rec(liste_rec_actif);
             IF (num_rec = 0) THEN
                liste_rec_pred := liste_rec_actif
              ELSE
              BEGIN
            calcul(liste_rec_effect, liste_rec_actif, liste_rec_pred);
            lecture (BUFF_IN,liste_rec_effect,liste_rec_actif, tab_res_int);
              END;

             num_em :=compter_em (liste_em_actif);
             IF (num_em <> 0) THEN
             BEGIN
              choix_val_in ( tab_act, liste_em_actif, val_choisie);
                          BUFF_IN := val_choisie;
                         END;
             BUFF_IN_VECT[k] := BUFF_IN;
             deblocage (liste_em_actif, tab_res_int);
            END; (* VSIg_evt *)

    IF (protocole = Vsig_evt) THEN liste_rec_pred_tab[k] := liste_rec_pred ELSE
liste_rec_pred_tab[k] := liste_rec_actif;
(*==================================================*)
 END;  (* sur la boucle channel *)

    END; (* procedure *)
```

PROCEDURE CONSTRUIRE

```
    PROCEDURE construire (VAR data_out_sup : type_acti_res; num_com :
INTEGER; tab_act : type_acti_res_tab;  VAR tab_act_out : type_acti_res_tab);

    BEGIN
            tab_act_out := tab_act;
            tab_act_out[num_com] := data_out_sup;
END;
```

PROCEDURE MERGER

```
    PROCEDURE merger (tab_res, tab_res_int :type_acti_res_tab; num_com
:INTEGER; VAR tab_res_out: type_acti_res_tab);

    VAR
            vect : type_acti_res;
            j, proc, port : INTEGER;

    BEGIN
            tab_res_out := tab_res_int;
            vect := tab_res[num_com];
            FOR j := 1 TO NBR_PORT_MAX DO
                        BEGIN
                         proc := TTAB_EXT_INT[j].num_proc;
                         port := TTAB_EXT_INT[j].num_port;
                         tab_res_out[proc,port] := vect[j];
                        END;
END;
```

PROCEDURE DE SIMULATION

```
    INITIALIZE TO SIM1

    VAR   J,K,L,M: INTEGER;

     BEGIN
                        FOR  J := 1 TO NBR_PROC_MAX DO
                         FOR K := 1 TO NBR_PORT_MAX DO
                          BEGIN
                                    tab_act_out[J,K] :=0 ;
                                    tab_res_out[J,K] := 0;
                                    tab_res_int[J,K] := 0;
                          END;
                        FOR K := 1 TO NBR_PORT_MAX DO
                         data_out_sup[K] :=0;
            num_com :=2;
            TTAB_CH_PORT := const_tab_ch_port_init;
            TTAB_EXT_INT := const_tab_ext_int_init;
            srand(1); (* initialisation du germe de tirage aleatoire *)
            FOR L := 1 TO NBR_CH_MAX DO
             BEGIN
                        BUFF_IN_VECT[L] := 0;
                        FOR M :=1 TO NBR_REC_MAX DO
                        BEGIN
```

```
                liste_rec_pred_tab[L,M].num_proc :=1;
                liste_rec_pred_tab[L,M].num_port :=3;
                END;
        END;
 END;

  (*————————————————————————————————*)
TRANS
        WHEN int_com_in.mess(tab_act)
        FROM SIM1
        TO SIM2
  BEGIN
simulation  (tab_act,  data_out_sup  ,  tab_res_int,  liste_rec_pred_tab,
BUFF_IN_VECT);
END;(* transition*)

  (*———————————————————————————————*)
TRANS
        WHEN pipe_in.mess(tab_act)
        FROM SIM2
        TO SIM3
        BEGIN
        construire(data_out_sup, num_com, tab_act, tab_act_out) ;
        OUTPUT ext_com_out.mess(tab_act_out);
 END;

TRANS
        WHEN ext_com_in.mess(tab_res)
        FROM SIM3
        TO SIM1
        BEGIN
        merger(tab_res_out, tab_res, num_com, tab_res_int);
        OUTPUT int_com_out.mess(tab_res_out);
END;
```

MOTS–CLES

modélisation, validation, vérification, composant micro_programmé, VHDL, communication, sémantique synchrone.

RESUME

Le mémoire décrit une méthode de modélisation et de validation de composants micro–programmés pour l'implantation de protocole de communication de réseaux. Cette méthode a été développée dans le cadre de la conception du composant FICOMP qui met en oeuvre la norme de bus de terrain FIELDBUS. Le premier chapitre décrit le contexte industriel du projet FICOMP, les différents niveaux de spécification du composant et les outils de simulation et de vérification utilisés. Le chapitre deux présente le langage VOVHDL, une extension de VHDL pour la spécification des communications et des synchronisations entre processus concurrents, et en donne une sémantique synchrone en termes de systèmes à transitions étiquetées. Le chapitre trois présente une approche de modélisation pour les descriptions VOVHDL hiérarchiques, et en illustre l'application au composant FICOMP: les modules internes sont reliés à un module de communication pour former un module de niveau supérieur ; ce module est alors traduisible dans le format d'entrée de l'outil de vérification ASA+. Le chapitre quatre rappelle les primitives essentielles du langage VHDL, et formalise la sémantique de simulation de ce langage en termes de systèmes à transitions étiquetées. Les annexes détaillent l'application de la méthode, par la spécification et la traduction dans le modèle proposé de deux modules du projet FICOMP.

KEY WORD

modeling, validation, verification, microprogrammed circuits, VHDL, communication, synchronous semantics

SUMMARY

The manuscript describes a modeling and validation methodology for the design of microprogrammed circuits that implement network communication protocols. The method was developped in the framework of the FICOMP project, which implements the FIELDBUS protocol standard. The first chapter describes the industrial context of the project, the various specification levels for the circuit, and the simulation and verification tools at hand. Chapter two presents the VOVHDL langage, an extension of VHDL for the specification of communication and synchronisations among concurrent processes; a synchronous semantics in terms of labeled transitions systems is given for VOVHDL. Chapter three presents a modeling approach for hierarchical VOVHDL descriptions, and shows its application to the FICOMP circuit: internal modules are interconnected with a communication module to build a higher level module; this model is then translated into the input format of the ASA+ verification tool. Chapter four recalls the essential features of VHDL and formalize its simulation semantics in terms of labeled transition systems. The application of the methodology to the specification of two modules of the FICOMP circuit, and their translation into the proposed model, are detailed in the appendices.