Vérification formelle des circuits digitaux décrits en VHDL

Ashrag Mohamed El-Farghly Salem

THESE

Présentée par

SALEM Ashraf Mohamed El-Farghly

pour obtenir le titre de

Docteur de l'Université Joseph Fourier - Grenoble I
(arrêté ministériel du 5 Juillet 1984 et du 23 Novembre 1988)

Spécialité : Informatique

Vérification formelle des circuits digitaux décrits en VHDL

Date de soutenance : 2 Octobre 1992

Composition du jury :

Mrs	Jacques VOIRON	Président
	François ANCEAU	Rapporteur
	Hans EVEKING	Rapporteur
Mme	Dominique BORRIONE	Directrice de thèse
Mr	Paul CASPI	Examinateur

Thèse préparée au sein du laboratoire ARTEMIS-IMAG

A Omnia, Nehal et Mahmoud

Je tiens à remercier

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

Madame le Professeur Dominique BORRIONE, Directrice du laboratoire ARTEMIS, qui a accepté de diriger cette thèse en m'accueillant au sein de son équipe et m'a permis d'effectuer mes recherches dans d'excellentes conditions; qu'elle reçoive l'expression de ma reconnaissance pour sa générosité, sa compréhension et sa gentillesse,

Monsieur le Professeur François ANCEAU, qui a eu la bienveillance de bien vouloir être le rapporteur de cette thèse, pour sa cordialité, ses commentaires judicieux et l'intérêt manifesté à l'égard de mon travail,

Monsieur le Professeur Hans EVEKING, qui a bien voulu être le rapporteur de cette thèse, je lui suis très reconnaissant pour les discussions fructueuses à l'origine de plusieurs idées présentées dans cet ouvrage,

Monsieur Paul CASPI, chargé de recherches au CNRS, qui a accepté de faire partie de ce jury, pour l'attention et le temps qu'il a bien voulu m'accorder,

Messieurs Claude Le FAOU et Gérard VITRY pour m'avoir aidé à surmonter des périodes très difficiles; qu'ils trouvent ici l'expression de toute ma reconnaissance,

Laurence PIERRE, Hélène COLLAVIZZA et Jean-Luc PAILLET pour leur collaboration efficace, leurs encouragements et leurs qualités de coeur,

tous mes collègues du laboratoire ARTEMIS qui m'ont apporté leur savoir et leur amitié tout au long de ce travail,

enfin et surtout, je dois beaucoup à ma femme Omnia, qui a accepté de vivre dans des conditions difficiles pendant de longues années. Sans ses sacrifices, ce travail n'aurait pu être mené à bien.

Chapitre 1

Introduction

1.1 Position du problème

Le progrès constant dans le domaine de la micro-électronique a permis de concevoir des circuits très hautement intégrés. La chaîne de conception de ces circuits est de plus en plus automatisée, notamment pour les niveaux les plus proches de la réalisation physique. Pour les niveaux plus abstraits, une intervention humaine est nécessaire pour s'assurer que la transition entre une spécification comportementale et une réalisation au niveau porte logique a été réalisée correctement. La simulation est la technique adoptée par l'industrie pour effectuer cette tâche. La simulation exhaustive d'un circuit de taille réelle est impossible, et la simulation non-exhaustive ne garantie pas l'absence totale d'erreur.

Les techniques de vérification formelle ont été proposées pour essayer d'offrir cette garantie. Un circuit est formellement vérifié, si nous prouvons mathématiquement que sa fonctionnalité réelle est équivalente à celle spécifiée par son concepteur. Pour ceci, quatre tâches doivent être accomplies.

1.Extraire la fonctionnalité du circuit.

2.Exprimer sa spécification.

3.Modéliser les deux fonctionnalités dans un modèle mathématique.

4.Réaliser la preuve de conformité dans ce modèle.

La recherche dans le domaine de la vérification formelle des circuits a été concentrée autour des deux derniers points:

Le calcul propositionnel, la logique de premier ordre, les machines d'état fini (FSM), la logique d'ordre supérieur et des calculs spécialement définis pour la preuve sont utilisés comme modèles mathématiques.

Les démonstrateurs généraux de théorèmes, Boyer-Moore [BM88], HOL [Go85] et OTTER[Mc89] sont utilisés pour réaliser la preuve. Les techniques basées sur les graphes de

décision binaire [Br86] sont utilisées pour prouver des circuits combinatoires industriels [MB88] et des machines d'état fini de complexité importante [CBM89].

En revanche, le domaine de la modélisation et de la spécification des circuits en vue de la vérification formelle a été moins étudié.

Trois approches peuvent être distingués dans ce domaine.

a) La modélisation directe des circuits dans le modèle formel utilisé pour la preuve.

b) L'emploi des langages de spécification formelle.

c) L'utilisation des langages de description de matériel.

Sémantiquement, la première approche est évidement la plus sûre car aucune transformation n'est demandée entre la description du circuit et la logique utilisée dans le processus de preuve. L'inconvénient de cette approche est syntaxique. Les logiques ont des syntaxes à la fois compactes et abstraites. Par conséquent, leur puissance d'expression est faible. Et, leur utilisation comme langage de description de matériels restreint le concepteur à un style de description très rigide.
La seconde approche exige une phase de traduction entre les langages de spécification et le modèle mathématique. Cette phase de traduction est souvent basée sur une sémantique formelle du langage de spécification.
La troisième approche demande, elle aussi, une phase de traduction. Cette phase est à la fois plus difficile et moins sûre par rapport à celle utilisée avec le langage de spécification formelle car les langages de description de matériels ont une syntaxe plus riche et une sémantique plus complexe. De plus, cette sémantique est souvent informelle.

L'intégration de la vérification formelle dans un système de CAO nécessite l'utilisation de langages de description de matériels comme interface pour les outils de vérification.
Cette intégration est la condition indispensable pour l'emploi de cette technique dans la chaîne industrielle de la conception de circuits, car la plupart des systèmes de CAO ont comme entrée un ou plusieurs langages de description de matériels. Le concepteur utilise ces langages pour simuler, tester et synthétiser ses circuits. L'adaptation de ces langages pour être utilisable comme entrée aux outils de preuve permettrait au concepteur de les utiliser de la même façon pour vérifier les circuits.
Le langage VHDL est le premier langage de description des systèmes digitaux qui ait fait l'objet d'une standardisation [Ie88]. En plus, la plupart des systèmes de CAO l'ont adopté comme leur langage d'entrée. Mais VHDL souffre, comme la majorité des langages de description, des trois problèmes précédemment cités: sa syntaxe est lourde, sa sémantique est complexe et cette

sémantique est informelle. Une contribution à la solution de ces trois problèmes dans le cadre de la vérification formelle est le sujet de cette thèse.

1.2 Etat de l'art

1.2.1 Les techniques de preuve

Une des premières techniques utilisées dans le domaine de la vérification formelle de circuits était issue du domaine de la preuve de programme. Darringer [Da79] a proposé d'appliquer la technique de l'exécution symbolique pour examiner le comportement du circuit décrit par un programme.
La limite des méthodes de preuve de programmes est imposée par la nature des langages de programmation. La plupart de ces langages sont procéduraux, ces méthodes ne sont donc applicable qu'à certains niveaux de conception, à savoir ceux dans lesquels le comportement est décrit par un algorithme [CP88].

Le calcul de prédicats a été utilisé pour la première fois dans le cadre de la vérification formelle de circuits par Wagner [Wa77], qui a prouvé un multiplieur 8-bit avec le démonstrateur FOL de l'université de Stanford.

W.Hunt [Hu86] a utilisé le calcul des prédicats pour vérifier automatiquement un microprocesseur conçu par lui, le FM6501. La spécification et la réalisation de ce microprocesseur ont été exprimés par des fonctions recursives dans la logique de Boyer et Moore [BM88]. La spécification a été considérée comme la succession des états du microprocesseur et la réalisation a été donnée par un réseau de portes logiques.

En se basant sur le travail de Hunt, le groupe de vérification formelle de l'IMEC a établi une méthode de vérification [VCM90] des modules paramétrés générés par l'environnement de synthèse Cathedral-II.

La logique d'ordre supérieur est une extension du calcul de prédicats qui admet l'utilisation des prédicats comme des variables. Cette logique est utilisée intensivement dans le cadre de la vérification formelle, surtout par des groupes anglais et canadiens. M.Gordon a modélisé les éléments de base des circuits séquentiels synchrones et des circuits combinatoires dans HOL [Go86]. Le démonstrateur de cette logique a été utilisé pour vérifier le microprocesseur VIPER [Co88]. Une méthodologie pour vérifier les modules paramétrés a été proposée dans[ACM91], et une comparaison entre cette méthode et celle employée par la même équipe avec le démonstrateur de Boyer et Moore est présentée dans [ACV92].

J .Herbert [He90] a modélisé le retard de propagation dans les portes logiques en HOL. Et il a utilisé le démonstrateur de cette logique pour raisonner formellement sur les caractéristiques temporelles des circuits synchrones et asynchrones.

Plusieurs chercheurs ont créé leur propre formalisme qui est plus ou moins inspiré par la logique de prédicat ou par la logique d'ordre supérieur.

Un système basé sur Prolog appelé Verify [Ba84] a été utilisé pour prouver un ordinateur simple. La spécification et la réalisation sont décrits en Prolog. Verify utilise, pour accomplir la preuve, les manipulations symboliques telles que la simplification, l'expansion et la mise dans une forme canonique. Quand la manipulation symbolique devient très lourde, il a recours à la simulation exhaustive.

Un modèle des processus concurrents appelé Circal[Mi85] a été développé par G. Milne. Circal est un modèle basé sur l'événement et il est destiné à représenter dynamiquement les comportements des systèmes concurrents. Chaque composant dans le système est modélisé par un processus Circal. Le comportement du système complet est établi par un opérateur de composition qui capture l'effet de l'interaction entre les processus actifs. La sémantique de Circal permet d'établir la cohérence d'un ensemble de règles de réécriture qui peut être utilisé ensuite pour prouver l'équivalence entre une spécification et une réalisation décrites en Circal.

J.L. Paillet de l'université de Provence a proposé un modèle fonctionnel pour la description et la spécification des circuits digitaux [Pa86]. Son modèle est basé sur l'opérateur passé **P** qui symbolise un retard d'une unité de temps. Il a défini l'opérateur P de la manière suivante.

$$(P(x))\ (t_i) = x(t_{i-1})$$

où x est la fonction représentant un signal x et t_{i-1} est la valeur de temps au top d'horloge précédente.

Paillet distingue deux types [Pa90] de circuits séquentiels, les circuits pseudo-combinatoires et les circuits séquentiels irréductibles. Pour les circuits pseudo-combinatoires l'opérateur temporel **P** disparaît grâce à la projection combinatoire, un processus qui remplace chaque puissance de **P** appliqué à une variable par une nouvelle variable. Quant aux circuits irréductibles, Paillet utilise le déroulement symbolique, une technique qui déplie les équations du circuit et élimine les variables irréductibles, pour un nombre fixé d'unités de temps.

A. Bronstein a défini une sémantique fonctionnelle pour les circuits synchrones [Br89] basée sur la notion de stream de Kahn [Ka74]. Bronstein a mécanisé sa sémantique dans le démonstrateur de Boyer et Moore. Et il a appliqué sa méthode sur les circuits prouvés par Paillet et sur des circuits de type "Pipeline".

L'emploi de graphes de décisions binaires [Ak78] par Bryant dans le développement des algorithmes performants pour la comparaison booléenne [Br86] et l'amélioration apportée sur ces graphes par Billon [Bi87][MB88] en proposant les graphes de décisions typés ont permis à la technologie de vérification formelle d'avoir une application industrielle. Des améliorations constantes sur les graphes de décisions binaires et les graphes de décisions types ont été publiées [DAC90][DAC91].

O.Coudert de BULL a présenté une technique de preuve basée sur les graphes de décisions typés [CBM89] pour les circuits séquentiels synchrones décrits au niveau booléen. Il a montré que les techniques d'énumération utilisées auparavant pour prouver ce type de circuits peuvent être remplacées par des manipulations symboliques.

La technique de "Model Checking" [QS82][BCD85] permet de vérifier qu'une réalisation satisfait des propriétés temporelles telles que la sûreté et la vivacité. Le système XESAR[Ro88], basé sur cette technique, à été développé au LGI-IMAG pour vérifier les protocoles de communication[RRSV87]. Il utilise le langage ESTELLE[Es85] pour décrire les protocoles et il les spécifie dans le Mu-Calcul. La technique de "Model Checking" était basée au départ sur les techniques d'énumération. Récemment, l'emploi des techniques symboliques [BCM90] a permis de prouver des automates de taille importante.

1.2.2 Les langages de description et la sémantique

La preuve des circuits décrits dans un langage de description de matériels a été présentée pour la première fois par Uehara et al [UMS81]. Ils ont proposé un système de preuve basé sur le langage DDL [DD68]. La réalisation du circuit est exprimée en DDL et sa spécification est donnée par une assertion en logique du premier ordre. La description DDL est traduite en des relations cause/effet qui montrent les conditions nécessaires et suffisantes pour le fonctionnement du circuit. Le système suppose que l'assertion est fausse et il trace les relations "cause/effet" pour trouver les erreurs de conception supposées. S'il montre qu'il n'existe aucune cause faisable, la supposition est niée et l'assertion est vérifiée.

L'équipe de H. Eveking a construit un système de vérification formelle appelé LOVERT [BEF90] qui est basé sur le langage de description de matériels SMAX [Ev85]. LOVERT accepte deux descriptions SMAX; l'une est considérée comme spécification et l'autre comme réalisation. Il décide alors si les deux descriptions sont équivalentes ou non. LOVERT vérifie:

a) L'équivalence entre une spécification booléenne et une réalisation décrite au niveau portes logiques.

b) L'agrégation correcte des expression scalaires à des expressions vectorielles.

c) L'équivalence entre une spécification au niveau transfert de registre et une réalisation structurelle.

L. Pierre de l'université de Provence a proposé un ensemble de techniques [Pi90] pour mécaniser la transition entre une description CASCADE[BL85] d'un circuit et sa représentation fonctionnelle en Boyer-Moore.

Le système PRIAM [Ma90] développé chez BULL est basé sur les graphes de décisions typés. Il compare le comportement extrait d'une réalisation physique d'un circuit combinatoire et sa spécification donnée par une description LDS [JM87]. Il permet aussi la localisation des erreurs de conception.

Les quatre langages traités par les travaux cités ci-dessus sont au niveau RTL. Ils n'ont pas, à l'exception de SMAX[Ev90], de sémantique formelle. Dans [Pi90], une sémantique formelle pour un sous-ensemble de CASCADE est définie dans le modèle fonctionnel de Paillet [Pa86]. Madre [Ma90] a donné une sémantique dénotationelle pour LDS en terme des graphes de décision typés.

D'autres chercheurs ont choisi de définir des langages de description basés sur des sémantiques formelles:

Sheeran a développé un langage de description des circuits appelé muFP [Sh84]. Ce langage est basé sur la sémantique de FP [Ba78] avec extensions pour manipuler le temps.

LUSTRE [HLP86] est un langage "flot de données" basé sur la notion de stream introduite par G.Kahn. Chaque variable en LUSTRE représente un couple. Le premier élément dans ce couple est une suite de valeurs et le deuxième est une horloge. Un programme LUSTRE [Pl88] est un réseau d'opérateurs connectés par des fils. Le programmeur peut définir ses propres opérateurs, appelés noeuds. Un noeud reçoit des flux en entrée, et produit des flux en sorties. Un circuit est décrit par deux noeuds représentant sa spécification et sa réalisation.

FUNDS [Bo88] est un langage fonctionnel de description de système qui supporte plusieurs sémantiques. Il est capable d'exprimer différents concepts de système dans le cadre de circuits digitaux et de circuits analogiques.

FUNNEL[SGE92] est un langage de description de matériels basé sur OBJ3[GW88]. Il permet d'exprimer la structure et le comportement de circuits. L'equivalence entre les deux peut être prouvé par 2OBJ, un démonstrateur pour le langage OBJ3.

Collavizza et Paillet ont proposé une sémantique fonctionnelle des microprocesseurs [Pa89][Co90]. Un outil de spécification et de preuve basé sur cette sémantique appelé μSPEED est en cours d'écriture [BCL91].

Un certain nombre de chercheurs ont travaillé sur la sémantique des langages de description industriels:

Borrione et Paillet [BP87] ont défini une sémantique pour les primitives **after** , **delayed** et **guarded** de VHDL dans le modèle fonctionnel de Paillet [Pa86]. Ils ont proposé une approche pour intégrer les outils de vérification formelle dans un système de CAO basé sur VHDL.

Une sémantique formelle basé sur la théorie de l'automate a été définie dans [BGL92] pour le langage ELLA [MPT85].

J.Tassel a défini [Ta90] une sémantique pour un sous-ensemble de VHDL dans VAL[Au89] et HOL. Et, il a défini, récemment, une sémantique formelle [Ta92] pour le cycle de simulation de VHDL en HOL.

1.3 Le plan de la thèse

Dans cette thèse, nous avons contribué à l'intégration des outils de vérification formelle dans les systèmes de CAO basés sur VHDL. Deux axes peuvent être distingués dans notre démarche: le premier est l'adaptation de VHDL pour les outils de preuve existants, le deuxième est le développement d'une sémantique formelle pour les primitives temporelles de ce langage. Les cinq chapitres suivant sont partagés entre ces deux axes. Les chapitres 2 à 4 détaillent le premier, quant au deuxième axe il est présenté dans les chapitres 5 et 6. Voici le contenu de chacun de ces chapitres.

Dans le chapitre 2, nous présentons les principes de la vérification formelle des circuits digitaux décrits en VHDL, en montrant l'incommodité de l'utilisation de la sémantique de VHDL pour décrire et prouver les circuits combinatoires et les circuits séquentiels synchrones.

Ensuite, nous définissons un sous-ensemble de VHDL, appelé P-VHDL, et un style de description dédié à la description de ces circuits. Nous associons une sémantique fonctionnelle basée sur le modèle de Paillet [Pa86] à ce sous-ensemble. L'équivalence entre cette sémantique et l'algorithme de simulation de VHDL est montrée. En nous basant sur cette sémantique, nous avons développé un compilateur orienté preuve pour ce sous ensemble. Ce compilateur constitue la base de notre environnement de preuve PREVAIL [BPS92a][BPS92b].

Dans le chapitre 3, nous prouvons JANUS un multiplieur en-ligne redondant[GHM89]. L'originalité de notre approche est double[Sa89][BS90]: (1) Pour spécifier JANUS, un circuit composé des éléments reconnus corrects est utilisé; (2) Le comportement séquentiel du multiplieur est déplié et interprété comme répétition dans l'espace, une technique qui ressemble à la méthode utilisée dans la conception des circuits systoliques.

Le chapitre 4 contient une définition dénotationnelle de P-VHDL. Nous commençons par la définition de la syntaxe abstraite de ce langage. Ensuite, nous définissions ses domaines sémantiques. Dans cette définition, nous distinguons trois domaines différents pour les variables, les signaux et les registres. Finalement, les fonctions d'évaluation pour les domaines syntaxiques sont définies.

Dans le chapitre 5, nous proposons une sémantique formelle pour les primitives temporelles de VHDL[SB91] . Notre sémantique est fonctionnelle, et elle est fortement inspirée de celle des langages flot de données [Ka74]. Dans notre formalisme, chaque processus VHDL est considéré comme une machine de calcul, et chaque signal est considéré comme une ligne de communication. A l'opposé de la sémantique opérationnelle de VHDL, la référence temporelle dans notre sémantique va du présent vers le passé. En conséquence, le calcul des valeurs des signaux est définitif, car toutes les informations nécessaires sont disponibles au moment du calcul.

Dans le chapitre 6, nous validons un certain nombre de lemmes utilisés dans les systèmes de vérification temporelle [EM90] [BJ83] dans notre sémantique. De plus, une équivalence partielle entre notre modèle formel et la sémantique informelle de VHDL est montrée. La validation de lemmes et la preuve d'équivalence ont été accomplies au moyen du démonstrateur automatique de Boyer-Moore après la modélisation de la sémantique dans la logique de celui-ci.

Chapitre 2

PREVAIL : Un Environnement de Preuve pour les descriptions VHDL

PREVAIL est un outil de preuve formelle conçu pour être intégré dans un système de CAO basé sur le langage VHDL. Ce chapitre est consacré à la présentation de cet outil. D'abord, notre approche de la vérification formelle des descriptions VHDL est présentée, en montrant l'incommodité de l'utilisation de la sémantique opérationnelle de VHDL dans le processus de preuve des circuits combinatoires et séquentiels synchrones. Puis, nous proposons un sous ensemble de VHDL pour décrire ces deux types de circuits et nous donnons une sémantique fonctionnelle de ce sous-ensemble. Enfin, l'équivalence entre cette sémantique et la sémantique opérationnelle de VHDL sera démontrée.

2.1. Principes de la vérification formelle de VHDL

Un circuit est décrit en VHDL par une entité de conception. L'entité est composée de deux parties: la déclaration de l'entité et une ou plusieurs architectures. La déclaration de l'entité décrit les paramètres du circuit et ses ports d'entrées sorties. Chaque entité peut avoir plusieurs architectures, décrivant des hypothèses de réalisation alternatives, ou des niveaux de description différents. Notre méthode pour vérifier formellement un circuit décrit en VHDL est de prouver l'équivalence entre deux architectures pour la même entité, où l'une est considérée comme une spécification et l'autre comme la réalisation.

entity Hardware_Unit **is**
    port (i_1: **in** T_{I1}; ... ; i_n : **in** T_{In}; o_1: **out** T_{O1}; ... ; o_m: **out** T_{Om});
end Hardware_Unit;

architecture Impl **of** Hardware_Unit **is**
    signal s_1 : T_{S1}; ... ; s_k : T_{Sk};
begin
    <architecture body>
end Impl;

architecture Spec **of** Hardware_Unit **is**
    signal $l_1 : T_{L1}; \ldots ; l_j : T_{Lj};$
begin
    <architecture body>
end Spec;

Pour raisonner formellement sur ce schéma VHDL, une sémantique formelle doit être définie pour les primitives du langage. Nous suivons un grand nombre d'auteurs [Go86][Sh84][Pa86] dans le choix d'une sémantique fonctionnelle pour décrire les circuits digitaux. Dans ce cadre, nous considérons l'architecture comme un ensemble de fonctions qui calcule les valeurs courantes des ports de sortie en utilisant les valeurs courantes et passées des ports d'entrée et les valeurs initiales des signaux internes.

Les sémantiques des architectures Impl et Spec peuvent être définies par les deux fonctions vectorielles suivantes:

$O =$ **Impl** (I, S) et $O =$ **Spec** (I, L)

où $O = < o_1, o_2, \ldots, o_m >$ le vecteur des ports de sortie
$I = < I_1, I_2, \ldots, I_n >$ le vecteur des historiques des ports d'entrée
$\forall\ j,\ 1 \leq j \leq n,\ I_j = < i_j(t), i_j(t-1), \ldots, i_j(0) >$ l'historique d'un port d'entrée
$S = < s_1(0), \ldots, s_k(0) >$ le vecteur des valeurs initiales des signaux internes pour la réalisation
$L = < l_1(0), \ldots, l_j(0) >$ le vecteur des valeurs initiales des signaux internes pour la spécification

Le problème de preuve consiste alors à décider si la relation de conformité
    Impl (I, S) R **Spec** (I, L)
prend la valeur vrai au faux, où R représente soit une implication, soit une équivalence.

2.2. La Structure du Système PREVAIL

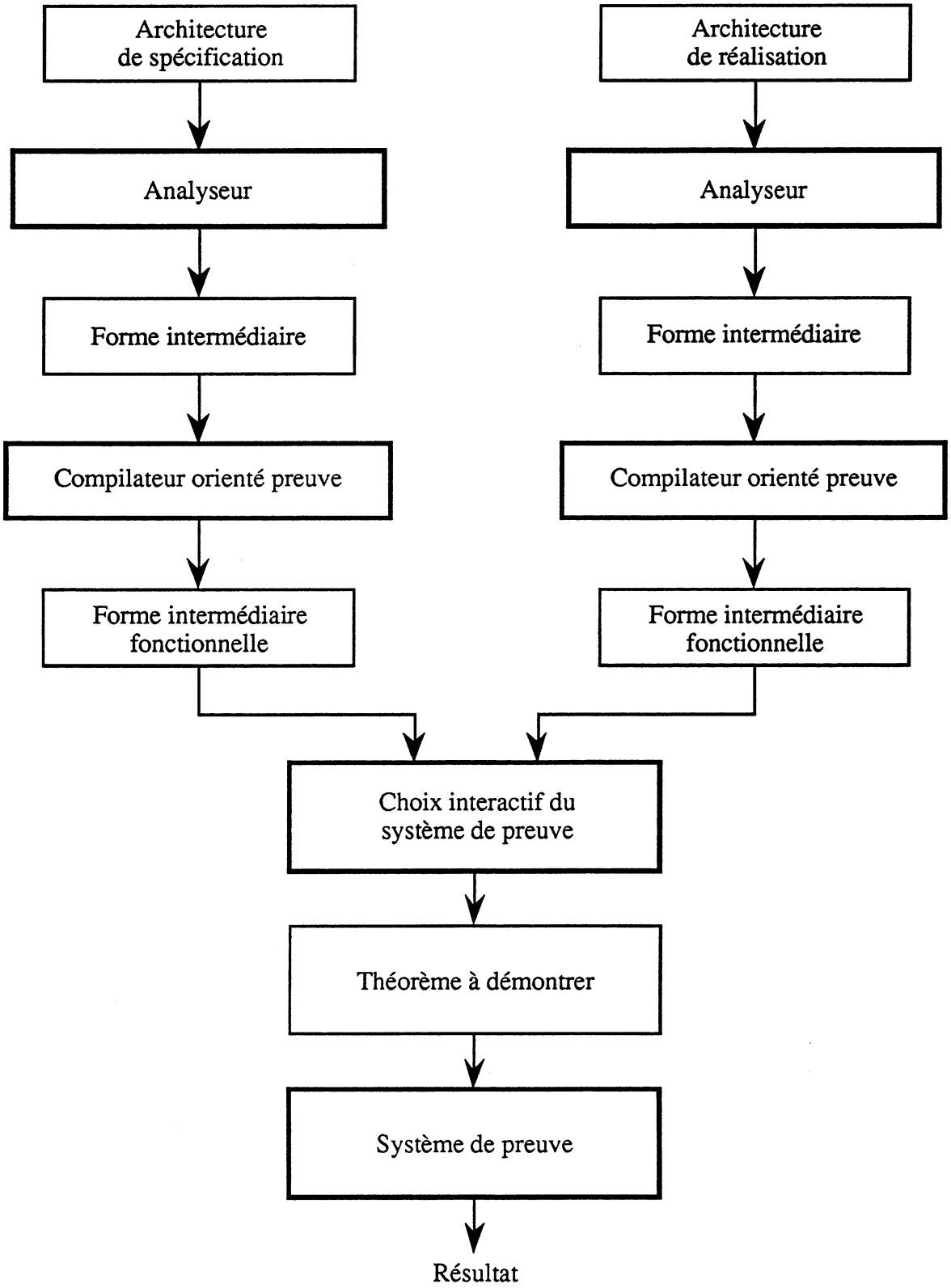
PREVAIL [BPS92a] [BPS92b] est un environnement de preuve pour les descriptions VHDL. Il accepte deux architectures, figure(2.1), de la même entité pour montrer si elles sont ou non équivalentes.
Un sous-ensemble de VHDL est utilisé par PREVAIL, ce sous-ensemble est appelé P-VHDL. Le source VHDL est compilé par un analyseur commercial [Cl90]. Le résultat de cette compilation est stocké dans une bibliothèque de conception sous une forme intermédiaire orientée objet. Cette forme intermédiaire est ensuite traduite par un compilateur spécialisé pour la preuve. Cette traduction est basée sur la sémantique fonctionnelle de P-VHDL. Le résultat de cette traduction est un ensemble de fonctions qui sera fourni à un des trois systèmes de preuve intégrés dans PREVAIL: TACHE[BP90], LOVERT [BFE90] et le démonstrateur général de Boyer et Moore [BM88].

1.TACHE est un démonstrateur de tautologies basé sur les graphes de décision binaires [Ak78] et les graphes de décision acyclique [Br85]. Utilisant la technique de déroulement symbolique [Pa86], ce démonstrateur est capable de prouver les circuits combinatoires et l'équivalence entre deux descriptions de circuits séquentiels à un point d'observation temporel donné.

2.LOVERT est un système destiné au préalable à prouver des circuits décrits au niveau transfert de registre (RTL). Il utilise trois techniques de preuve: la réécriture pour simplifier les expressions vectorielles, les graphes de décisions typés [MB88] pour les expressions booléennes et le processus d'exécution symbolique de la machine produit [CBM89] pour prouver l'équivalence des deux machines d'état fini.

3.Le démonstrateur inductif de Boyer et Moore est utilisé dans PREVAIL pour prouver la fonctionnalité des circuits combinatoires répétitifs [BPS92c][Pi90] où la spécification est donnée au niveau arithmétique.

Le processus du choix entre les trois démonstrateurs n'est pas automatique. L'usager doit choisir le système de preuve en fonction du type de circuit et de la nature de sa spécification.



Figure(2.1): La Structure du Système PREVAIL

2.3. La sémantique du langage et les modèles formels

La démonstration automatique de la relation de conformité entre **Impl** et **Spec** par un seul outil de preuve n'est pas réalisable pour trois raisons: la limitation de ces outils, le modèle temporel complexe de VHDL, et le nombre important de types de données prédéfinis et la possibilité de définir des types supplémentaires dans le langage.
De plus, cette démonstration n'est pas nécessaire pour une grande partie des circuits décrits en VHDL. En effet, VHDL complet est défini en vue de la simulation. Les applications autres que la simulation qui prennent VHDL comme langage d'entrée traitent en réalité un sous-ensemble du langage pour lequel on peut associer soit une sémantique en terme de circuit équivalent (synthèse[CST91]) soit un modèle formel manipulable (vérification[BP87][SB90][DBJ92]).

PREVAIL est destiné, en priori, à prouver deux types de circuits: les circuits combinatoires et les circuits séquentiels synchrones. Ces deux types de circuits peuvent être décrits en utilisant un sous-ensemble de VHDL. Leurs sémantiques, par contre, sont différentes par rapport à celle de VHDL. La sémantique des circuits combinatoires est exprimée, si on ignore les retards des portes, par la logique propositionnelle. Quant aux circuits séquentiels synchrones, leur sémantique est exprimée dans plusieurs modèle formels: la machine d'état fini, la logique de deuxième ordre[Go86], et la sémantique des chaînes[Pi90][Br90]. En revanche, la sémantique des primtives VHDL est définie par un algorithme de simulation. Cet algorithme traite, en principe, deux problèmes: le premier est l'exécution d'une description parallèle sur une machine séquentielle, le deuxième étant la simulation du temps.

Dans ce qui suit, nous montrons comment l'utilisation de la logique propositionnelle et la sémantique des machines d'état fini au lieu de celle des primitives de VHDL régle les trois problèmes rencontrés en prouvant la relation de conformité entre les deux architectures. Nous expliquons comment ce choix nous guidera dans le processus de définition du sous-ensemble de VHDL traité par PREVAIL.

Problème N° 1: la limite des outils de preuve

Les deux catégories de démonstrateurs automatiques utilisés pour prouver les circuits digitaux, c'est à dire, les démonstrateurs généraux et les outils conçus spécialement pour la preuve des circuits, expriment directement la fonctionnalité du circuit à prouver dans le système formel utilisé par le démonstrateur. Par conséquent, le choix de ce modèle formel comme sémantique pour le sous-ensemble rend possible l'utilisation de ces démonstrateurs pour prouver les circuits décrits dans ce sous-ensemble.

Problème N° 2: le modèle temporel de VHDL

Un circuit combinatoire peut-être décrit à un certain niveau d'abstraction sans avoir besoin d'utiliser des primitives temporelles, aussi son fonctionnement peut être calculé par une composition fonctionnelle dans un ordre décidable [FP84]. Donc, la sémantique de l'algorithme de simulation, i.e. le modèle temporel complexe de VHDL, peut être ignorée en traitant un sous-ensemble destiné à décrire les circuits combinatoires, et la logique propositionnelle peut être utilisé comme une sémantique de ce sous-ensemble.

Par contre, pour un circuit séquentiel, même si le calcul de l'ordre de la composition reste possible, l'utilisation des primitives temporelles (les primitives Stable et Wait pour la synchronisation, la primitive after pour décrire l'horloge) est inévitable. En conséquence, un sous-ensemble syntaxique de VHDL ne peut être suffisant pour ce type de circuits et un **style de description** sera indispensable pour pouvoir associer la sémantique des machines d'état fini à une description VHDL.

Problème N° 3: les types de données en VHDL

En limitant les circuits à prouver aux circuits combinatoires et aux circuits synchrones et en choisissant des spécifications aux niveaux bit et bit_vector, nous pouvons éliminer tout autre type de données du sous-ensemble utilisé par PREVAIL.

2.4. P-VHDL: Un sous ensemble pour les circuits combinatoires et les circuits séquentiels synchrones

P-VHDL est le sous-ensemble utilisé par PREVAIL pour décrire les circuits combinatoires et les circuits séquentiels synchrones. Il est associé avec un **style de description** et avec **un paquetage de preuve** pour identifier les éléments de mémoire dans un circuit synchrone.

Deux raisons ont rendu l'utilisation d'un style de description et d'un paquetage de preuve nécessaire pour le processus d'identification:

1- VHDL est un langage asynchrone sans horloge pré-définie,

2- VHDL n'offre pas de classe spéciale pour les signaux de mémoire synchrones.

L'avantage de s'appuyer sur un paquetage de preuve en plus du style de description est de réaliser le processus d'identification d'une manière syntaxique. En revanche, en se limitant seulement au style de description, ce processus ne peut être que sémantique.

Nous nous basons donc sur notre paquetage **proof** figure(2.2). Dans celui-ci, le type **clock** est utilisé pour déclarer qu'un signal est une horloge. La sémantique de ce type est implicite dans le modèle formel. Un signal de ce type peut faire partie d'un expression logique pour générer des horloges secondaires. Les types **reg, reg_vector** et **reg_matrix** sont utilisés pour déclarer les registres. Les affectations des signaux de ces types doivent être synchronisées par un signal de type **clock.**

```
package proof is
subtype clock is bit;
subtype reg is bit;
subtype reg_vector is bit_vector;
type bit_matrix is array(integer range <>, integer range <>) of bit;
subtype reg_matrix is bit_matrix;
. . . .
end package;
```

Figure(2.2): Les déclarations des sous-types dans le paquetage PROOF

P-VHDL supporte les trois styles de descriptions VHDL: le style flot de données, le style structurel, le style comportemental. Les restrictions suivantes sont commune aux trois styles:

- la primitive **after** n'est pas autorisée,
- seules les opérations sur les types **bit** et **bit_vector** sont autorisées,
- les attributs **stable** et **event** sont uniquement autorisés sur les signaux de type **clock.**
- les attributs **transaction, last_event, active, last_active** et **last_value** ne sont pas autorisés.
- l'instruction **wait** est uniquement autorisée soit sur un signal de type **clock** soit sur une expression contenant un signal de ce type.
- les instruction **while** et **repeat** ne sont pas autorisées.

Dans ce qui suit, les restrictions syntaxiques et sémantiques imposées par P-VHDL sur les primitives de chaque style sont présentées, et une sémantique fonctionnelle basée sur le modèle de Paillet [Pa86] pour ces primitives est définie.

2.4.1 Le style flot de données

Dans une description du style flot de données, l'architecture est composée d'un ensemble d'affectations concurrentes. Ce style de description est inspiré des langages de niveau transfert de registre. Nous suivons ce type de langage en distinguant l'affectation d'un élément de mémoire (le registre), où la cible doit avoir le type **reg**, et l'affectation qui modélise une partie combinatoire, où la cible a le type bit.

VHDL a trois types d'instruction d'affectation du signal : l'affectation non conditionnée, l'affectation conditionnée, l'affectation gardée. P-VHDL autorise ces trois types. Le bloc et l'affectation gardé en P-VHDL doivent respecter les deux restrictions suivantes:

- la cible de l'affectation gardée doit avoir le type **reg, reg_vector** ou **reg_matrix.**
- la condition de garde doit contenir un signal de type **clock**, et ce signal doit être attribué par l'attribut stable.

En P-VHDL, nous appelons un bloc gardé respectant ces deux restrictions, un **bloc synchrone.**

Les exemples suivants montrent l'utilisation de P-VHDL pour modéliser les quatre éléments principaux dans un langage de transfert de registres ainsi que la sémantique fonctionnelle associée à cette modélisation.

Elément 1 : Une porte logique

```
signal a,b,c : bit;
. . . . . .
c <= a and b;
```

La fonction suivante exprime la sémantique de cette instruction:

c = **et** a b

Elément 2 : Un Multiplexeur

```
signal a,b,c : bit_vector(0 to 3);
signal x : bit;

c <= a when x = '0' else b;
```

La fonction qui exprime la sémantique de cette instruction est:

c<0:3> = **si non** x **alors** a<0:3> **sinon** b<0:3>

Elément 3 : Une bascule

```
signal a : bit;
signal b : reg;
signal clk : clock;

Bascule: block(clk = '1' and not clk'stable)
begin
     b <= guarded a;
end block;
```

La sémantique de ce bloc synchrone est donnée par la fonction:

b =**P** a où (**P** a)(t_j) = a(t_{j-1}) et $t_j - t_{j-1}$ = la période d'horloge[Pa86]

Elément 4 : Une bascule avec une condition de chargement

```
signal a,x : bit;
signal b : reg;
signal clk : clock;
Latch: block(clk = '1' and not clk'stable and x = '1')
      b <= guarded a;
end block;
```

La sémantique de ce bloc synchrone est donnée par la fonction suivante:

b = **P** (**si** x **alors** a **sinon** b)

2.4.2 Les descriptions comportementales

La description comportementale décrit le circuit dans un style algorithmique. Les instructions séquentielles en VHDL peuvent être trouvées dans un processus, dans une fonction et dans une procédure. Les instructions séquentielles peuvent utiliser des signaux et des variables.

Le déclenchement d'un processus est contrôlé soit par la liste de sensibilité du processus, soit par l'instruction Wait.

Nous imposons dans P-VHDL des limitations sur les deux méthodes: la première méthode doit être utilisée dans les processus décrivants les circuits combinatoires, appelés dans la suite **processus combinatoires**, et la seconde, dans les processus décrivants les circuits séquentiels, appelés dans la suite **processus synchrones**.

Dans un **processus combinatoire**, la liste de sensibilité doit contenir tous les signaux lus mais non modifiés dans le processus, pour assurer que le processus est exécuté à chaque modification d'un signal en partie droite d'affectation, faute de quoi il y aurait mémorisation. Sous cette restriction, tous les signaux de la liste de sensibilité peuvent être considérés comme des entrées du processus, et tous les signaux modifiés peuvent être considérés comme des sorties du processus.

Dans les **processus synchrones**, nous avons choisi l'utilisation de l'instruction **Wait** pour synchroniser les affectations des signaux. Une autre possibilité consistait à mettre l'horloge dans la liste de sensibilité. Notre choix a deux avantages:

- permettre de décrire une succession d'états implicite, s'il y plusieurs Wait sur l'horloge dans le processus.
- Donner un critère syntaxique pour identifier un processus synchrone.

Dans la version actuelle de PREVAIL, on se limite à une seule instruction Wait par processus placée comme première instruction et sa forme est limitée aux trois modèles suivants:

- **wait on** clk **until** clk = '1';
- **wait on** clk = '1' **and not** clk'**stable**;
- **wait on** clk = '1' **and** clk'**event**;

où clk est un signal de type **clock**.

P-VHDL impose également des restrictions sur l'utilisation des variables et des signaux. Dans un processus synchrone, l'utilisation des variables comme éléments de mémorisation est interdite. Et les signaux utilisés comme des cibles dans les instructions d'affectation doivent avoir les types **reg** et **reg_vector**, qui permettent de les identifier comme éléments de mémorisation.

Dans un processus combinatoire, un signal affecté sous la portée d'une condition doit être affecté pour toutes les valeurs possibles de cette condition, i.e. tous les **if** doivent avoir un else, tous les **case** doivent prévoir toutes les valeurs possibles, ou avoir une partie **else.**
Les variables sont autorisées dans les processus avec les restrictions suivantes:

- la variable ne doit pas être référencée dans une expression avant qu'elle ne soit affectée.
- la variable doit être affectée par une expression contenant comme opérandes, soit des constantes, soit des signaux faisant partie de la liste de sensibilité, soit des variables respectant les restrictions précédentes.

Les quatres éléments du langage de transfert de registres modélisés par des descriptions flot de données dans la section 2.4.1, sont modélisés par les quatre processus P-VHDL:

Elément 1 : Une porte logique

```
et : process (a,b)
begin
      c<= a and b;
end  process;
```

Elément 2 : Un Multiplexeur

```
Multiplexeur : process (a,b,x)
begin
      if x= '0' then c <= a;
      else c<= b;
      end if;
end  process;
```

Elément 3 : Une bascule

```
Bascule : process
begin
wait on clk until clk = '1';
      b <= a;
end  process;
```

Elément 4 : Une bascule avec une condition de chargement

```
Bascule_conditionne: process
begin
wait on clk until clk = '1';
      if x = '1' then b <= a; end if;
end  process;
```

Les sémantiques fonctionnelles pour les quatre processus sont identiques à celles des deux affectations et des blocs utilisés pour les même éléments dans la section (2.4.1)

Les procédures et les fonctions sont autorisées dans P-VHDL avec la condition que leurs paramètres ne soient pas de classe signal . Cette restriction limite les objets accessibles dans une procédure ou dans une fonction à la classe **variable**. Cela est justifié par notre choix d'utiliser les procédures et les fonctions comme moyen de calcul et de ne les pas utiliser comme outils de modélisation.

Exemple: Une procédure pour calculer la somme de deux bit

```
procedure add_1 (a,b,c: in bit; sum,carry:out  bit) is
begin
      sum := (a xor b) xor c ;
      carry := (a nand b) nand (c nand (a xor b));
end;
```

Les deux fonctions suivantes expriment la sémantique de cette procédure:

add_1-sum = λ (a b c) . (**xou** c (**xor** a b))

add_1-carry = λ (a b c) . (**nand** (**nand** b a) (**nand** c (**xor** a b)))

Le nom de chaque fonction est composé de deux parties: la première est le nom de procédure et la deuxième est le nom de variable.

PREVAIL utilise un ensemble de fonctions prédéfinies[BEF90], les déclarations de ces fonctions se trouvent dans le paquetage **proof** figure (2.3). Leurs sémantiques sont intégrées dans le démonstrateur LOVERT.

```
package proof is
subtype clock is bit;
subtype bit_reg is bit;
subtype bit_vector_reg is bit_vector;
type bit_matrix is array(integer range <>,integer range <>) of bit;
subtype bit_matrix_reg is bit_matrix;
function rol (p:bit_vector) return bit_vector; -- rotation à gauche
function rot (p:bit_Vector) return bit_vector; -- rotation à droite
function inc (p:bit_vector) return bit_vector; -- incrémentation
function dcr (p:bit_vector) return bit_vector; -- decrémentation
function rsh (p:bit_vector) return bit_vector; -- décalage à droite
function lsh (p:bit_vector) return bit_vector; -- déclage à gauche
function fae (p:bit;n:integer) return bit_vector; -- fan out
function adc (p1:bit_vector;p2:bit_vector;p3:bit) return bit_vector; -- addition avec retenue
function sbb (p1:bit_vector;p2:bit_vector;p3:bit) return bit_vector; -- soustraction
function add (p1:bit_vector;p2:bit_vector) return bit_vector;  -- addition sans retenue
function mint (p1:bit_vector;n:integer) return bit; -- min terme
function maxt (p1:bit_vector;n:integer) return bit; -- max terme
function cat (p1:bit_matrix;p2:bit_vector) return bit_matrix; -- concaténation
function cat (p1:bit_vector;p2:bit_vector) return bit_matrix;
function mpx1 (p1:bit_vector;p2:bit_vector) return bit; -- Multiplexeur
function mpx2 (p1:bit_matrix;p2:integer) return bit_vector;
function mpx2 (p1:bit_matrix;p2:bit) return bit_vector;
function mpx2 (p1:bit_matrix;p2:bit_vector) return bit_vector;
end proof;
```

Figure(2.3): Le paquetage PROOF

2.4.3 Le style structurel

Dans une description structurelle, les circuits sont décrits par une interconnexion de composants. Ce type de description cache les détails de réalisation des composants et permet d'avoir une description hiérarchique du circuit. Le lien entre le composant et le couple (entité, architecture) est établi par une configuration. Cette configuration peut être statique, c'est-à-dire spécifiée dans le bloc où le composant est utilisé, ou dynamique, c'est-à-dire définie par une configuration globale pour l'entité déterminant les liens entre tous les composants utilisés dans cette entité et les couples (entité,architecture) correspondants. P-VHDL autorise seulement la configuration statique, car l'utilisation d'une configuration globale empêche l'apparition de l'architecture comme unité de preuve. Nous appelons le bloc utilisé pour instantier les composants un **bloc d'instantiation** .

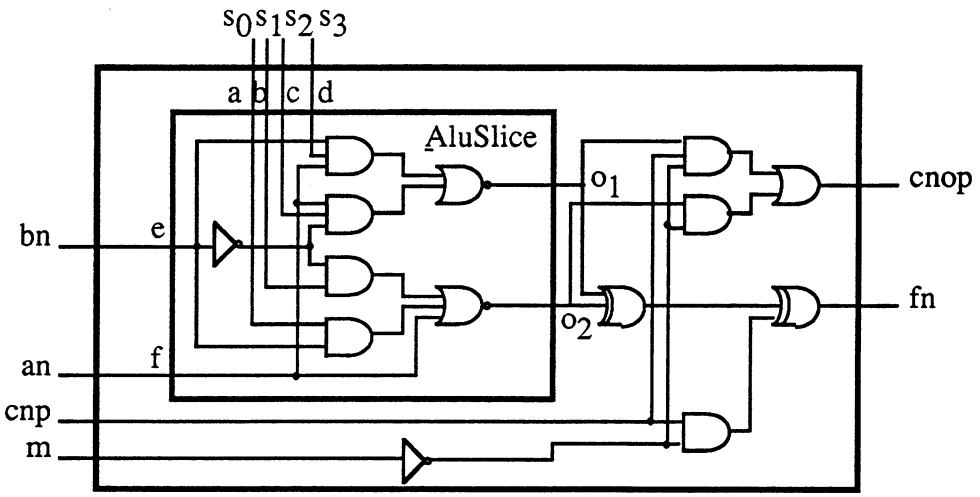
La sémantique d'un couple (entité, architecture) dans P-VHDL est donnée par un ensemble de fonctions, une par port de sortie et par signal de type **reg**. Les paramètres des fonctions sont les

ports d'entrée et les signaux de type **reg** déclarés dans l'architecture. Dans la définition sémantique le nom de fonction est composé de trois partie: le nom de l'entité, le nom de l'architecture et le nom de signal ou de port.

La sémantique de l'instruction d'instanciation est exprimée par un ensemble de fonctions, une par port formel de sortie et par signal de type registre déclaré dans l'architecture qui décrit le composant.

Exemple: Un circuit combinatoire

Considérons l'UAL 1-bit figure(2.4). Cette UAL est construite en utilisant un AluSlice et sept portes logiques.



Figure(2.4): UAL 1-bit

Le source VHDL suivant décrit L'AluSlice:

```
entity AluSlice is
      port(s:in bit_vector (0 to 3); e,f: in bit;
            o1,o2 : out bit);
end AluSlice ;

architecture dataflow of AluSlice is
begin
      o1 <= (s(3) and e and f) nor (f and s(2) and (not e)) ;
      o2 <= not(f or (e and s(0)) or (s(1) and (not e))) ;
end dataflow;
```

La sémantique de l'architecture *dataflow* de l'entité AluSlice est donnée par les deux fonctions suivantes:

AluSlice-dataflow-o1 = λ (s<0:3> e f) . **nor** (**et** s<3> (**et** e f)) (**et** f (**et** s<2> (**non** e))

AluSlice-dataflow-o2 = λ (s<0:3> e f) . **ou** (**non** (**ou** f (**et** e s<0>)))(**et** s<1> (**non** e))

L'entité *OneBitAlu* et l'architecture *structure* donnent pour L'UAL:

```
entity OneBitAlu is
      port(s:in bit_vector(0 to 3); bn,an,m,cnp:in bit ; fn,conp:out bit);
end OneBitAlu;

architecture structure of OneBitAlu is

component AluSlice
      port(s:in bit_vector(0 to 3); e,f:in bit; o1,o2:out bit);
end  component ;

signal so1,so2 : bit;

begin
      SliceBlock:block
      for sl1: AluSlice use entity work.aluslice(dataflow);
      begin
            sl1 : AluSlice port map(s,bn,an,so1,so2);
      end  block;
      fn <= (so1 xor so2) xor ((not m) nand cnp);
      conp <= (not m and so2) or (so1 and (not m) and cnp);
end structure;
```

La sémantique de l'architecture *structure* est donnée par les quatre fonctions suivantes:

OneBitAlu-structure-so1= λ (s<0:3> bn an m cnp) . AluSlice-dataflow-o1(s<0:3>, bn,an)

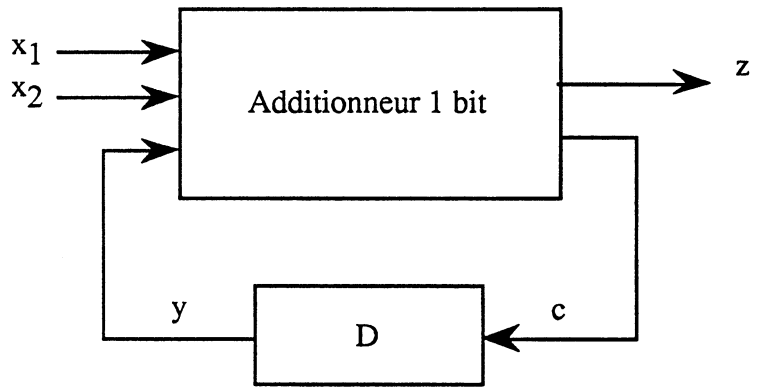
OneBitAlu-structure-so2= λ (s<0:3> bn an m cnp) . AluSlice-dataflow-o2(s<0:3>, bn,an)

```
OneBitAlu-structure-fn =
      λ (s<0:3> bn an m cnp) .
            xou  (xou OneBitAlu-structure-so1(s<0:3>, bn, an , m , cnp)
                        OneBitAlu-structure-so2(s<0:3> bn , an ,  m , cnp))
                 (nand (non m) cnp)
```

```
OneBitAlu-structure-conp =
      λ (s<0:3> bn an m cnp) .
            ou (et (non m) OneBitAlu-structure-so2 (s<0:3>, bn , an , m , cnp))
               (et OneBitAlu-structure-so1(s<0:3> , bn , an , m  , cnp) (et (non m) cnp)
```

Exemple: Un circuit séquentiel (Un additionneur en série[Ko78])

Nous modélisons dans cet exemple un additionneur en série [Ko78]. Cet additionneur figure (2.5) est construit en utilisant une bascule est un additionneur 1-bit. La bascule **D** est modélisée par l'entité FlipFlop.



Figure(2.5) : L'additionneur en série

```
entity FlipFlop is
      port ( d : in bit ;  clk : in clock; q : out reg)
end FlipFlop;

architecture Imp of FlipFlop is
begin
      b: block ( clk = '1' and not clk'stable)
      begin
            q<= guarded d;
      end block;
end imp;
```

La sémantique du couple (FlipFlop, Imp) est exprimée par la fonction suivante:

FlipFlop-Imp-q = λ (d) . **P** d

Et, l'additionneur 1 bit est modélisé par l'entité FullAdd.

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

architecture Imp of FullAdd is
begin
      sum <= (a xor b) xor c;
      carry <= (a nand b) nand (c nand (a xor b));
end;
```

La sémantique du couple (FullAdd, Imp) est exprimée par les deux fonctions suivantes:

FullAdd-Imp-sum = λ (a b c) . (**xou** c (**xor** a b))

FullAdd-Imp-carry = λ (a b c) . (**nand** (**nand** b a) (**nand** c (**xor** a b)))

L'entité SerialAdder modélisé l'additionneur en série par une interconnexion entre l'additionneur 1 bit et la bascule.

```
entity SerialAdder is
    port ( x1, x2 : in bit; clk : in clock; z : out bit);
end SerialAdder;

architecture Struc of SerialAdder is
component FullAdd
    port(a,b,c : in bit; sum,carry : out bit)
end component;
component  FlipFlop
    port ( d : in bit ; clk : in clock; q : out reg)
end component;
signal c:bit; signal y: reg;
begin
    b: block
    for A:Fulladd  use entity FullAdd(Imp);
    for D:FlipFlop use entity FlipFlop(Imp);
    begin
        A: Fulladd port map (x1,x2,y,z,c);
        D: FlipFlop port map (c,clk,y);
    end block;
end struc;
```

La sémantique du couple (SerialAdder , Struc) est exprimée par les trois fonctions suivantes:

SerialAdder-Struc-z = λ (x1 x2 SerialAdder-Struc-y)
Fulladd-imp-sum(x1,x2,SerialAdder-Struc-y)

SerialAdder-Struc-c = λ (x1 x2 SerialAdder-Struc-y)
Fulladd-imp-carry(x1,x2,SerialAdder-Struc-y)

SerialAdder-Struc-y = λ (x1 x2 SerialAdder-Struc-y)
FlipFlop-Imp-q (SerialAdder-Struc-c(x1,x2,SerialAdder-Struc-y))

Chaque fonction a comme paramètres les deux entrées **x1, x2** et la variable d'état **y.** La sémantique de d'instruction d'instanciation **A** est donnée par les deux fonction **SerialAdder-Struc-z** et **SerialAdder-Struc-c.**

2.5. Primitives pour le niveau arithmétique

Le sous-ensemble de section (2.4) permet l'utilisation des démonstrateurs booléens (Tache, Lovert) pour prouver les circuits combinatoires et séquentiels synchrones décrits au niveau booléen. Pour traiter les circuits répétitifs ayant des spécification au niveau arithmétique, PREVAIL utilise un démonstrateur général (Boyer et Moore) [BPS92c].

Les primitives suivantes ont été ajoutées au sous-ensemble pour permettre la description de ce type de circuit:

- Les signaux et les variables à valeur **integer.**
- La primitive **generate.**
- Les entités génériques avec paramètres **natural.**

Deux types sont ajoutés au paquetage **proof: int_reg** et **nat_reg.**

subtype int-reg **is** integer;
subtype nat-reg **is** natural;

Ainsi que deux fonctions de conversion **bv_to_nat** et **nat_to_bv.** Les sémantiques de deux fonctions sont intégrée dans le démonstrateur de Boyer et Moore [Hu86].

function bv_to_nat (p1:bit_vector) **return** natural;
function nat_to_bv (p1:natural) **return** bit_vector;

Exemple:

L'entité **Add_N** modélise un additionneur à retenue propagée N bit, l'architecture **Impl** décrit la réalisation de ce circuit en terme d'une interconnexion de N additionneurs 1 bit. L'architecture **Spec** donne une spécification arithmétique pour ce circuit.

```
entity Add_N is
    generic (n :natural);
    port  (a,b : in bit_vector (0 to  n-1 ); o : out bit_vector  (0 to  n-1 );  rco : out bit);
end Add_N;
```

```
architecture Impl of Add_N is

component FullAdd
      port(a,b,c : in bit; sum,carry : out bit)
end component;

signal c :bit_vector (0 to n);
begin
      B:block
      for all : FullAdd use entity FullAdd(Impl);
      begin
            b1: for j in 0 to n-1 generate
                  b2: FullAdd port map (a(j), b(j), c(j), o(j), c(j+1));
            end generate;
      end block;
      c(0) <= '0'; rco <= c(n)
end impl;
```

La sémantique du couple (Add_N, Impl) est exprimé par les fonctions suivantes:

Add_N-Impl-c<0> = λ (n a <0: n-1> b<0:n-1>) . **Faux**

∀ j, 0 ≤j≤n-1, Add_N-Impl-o<j> = λ (n a <0: n-1> b<0:n-1>) .
FullAdd-Impl-sum(a<j>, b<j>, Add_N-Impl-c<j>(n, a <0: n-1> , b<0:n-1>))

∀ j, 1 ≤j≤n , Add_N-Impl-c<j> = λ (n a <0: n-1> b<0:n-1>) .
FullAdd-Impl-carry(a<j-1>, b<j-1>, Add_N-Impl-c<j-1> (n, a <0: n-1> , b<0:n-1>))

Add_N-Impl-rco = λ (n a <0: n-1> b<0:n-1>) .
Add_N-Impl-c<n>(n, a <0: n-1> , b<0:n-1>)

```
architecture spec of Add_N is

signal sum : natural;
signal o_c : bit_vector(0 to n);

begin
      sum <=bv_to_nat (a) + bv_to_nat(b);
      o_c <= nat_to_bv (sum);
      o <= o_c (0 to n_1); rco <= o_c(n);
end spec;
```

La sémantique du couple (Add_N, Spec) est exprimé par les fonctions suivantes:

Add_N-Spec-sum = λ (n a <0: n-1> b<0:n-1>) . (bv_to_nat(a) + bv_to_nat(b))

Add_N-Spec-o_c = λ (n a <0: n-1> b<0:n-1>) . nat_to_bv(sum)

Add_N-Spec-o<0: n-1> = λ (n a <0: n-1> b<0:n-1>) .
Add_N-Spec-o_c<0:n-1>(n, a <0: n-1> , b<0:n-1>)

Add_N-Spec-rco = λ (n a <0: n-1> b<0:n-1>) .
Add_N-Spec-o_c<n>(n, a <0: n-1> , b<0:n-1>)

2.6. La relation entre le modèle formel et la sémantique de P-VHDL

Nous avons associé les sémantiques de circuits combinatoires et de circuits séquentiels synchrones à la syntaxe de P-VHDL. Nous justifions ce choix en montrant comment l'utilisation du sous-ensemble et du style de description rend la sémantique opérationnelle de VHDL équivalente à celles de ces deux types de circuit.

A. Les circuits combinatoire

Un circuit combinatoire à retard nul est caractérisé par le fait que ses sorties, pour tout t>0, peuvent être exprimées comme des fonctions de ses entrées au même instant t . Tous les signaux internes, qui expriment explicitement les interconnections des composants dans une description structurelle, ou implicitement dans une description flot de données, ou qui agissent comme des canaux de communication pour passer les valeurs dans un processus, peuvent être éliminés par une composition fonctionnelle en traçant la description du circuit de ses sorties à ses entrées. En particulier, ceci implique l'absence de boucle de rétroaction. En respectant cette condition, un circuit combinatoire décrit en P-VHDL peut être exprimé par un ensemble de formules de calcul propositionnel (une pour chaque sortie). Et l'équivalence entre deux architectures peut être prouvée en décidant si la formule suivante est une tautologie ou non.

Impl (I) ⇔ Spec (I) où $I = < i_1, ..., i_n >$

i_1, ... i_n sont des variables booléennes (une par entrée scalaire du circuit)

Impl, Spec sont des fonctions booléennes vectorielles (un élément par sortie du circuit)

Le simulateur VHDL utilise le retard unitaire delta pour sérier l'exécution des affectations de signaux concurrents. Sous l'hypothèse que le graphe de dépendance entre les affectations de signaux n'a pas de cycle, le résultat de la simulation d'un circuit combinatoire décrit en utilisant

les affectations concurrentes stabilisées est équivalent à la formule de la logique propositionnelle décrivant le circuit. Par conséquent, la logique propositionnelle peut définir la sémantique des instructions des affectation concurrentes. La justification de cette équivalence est la suivante [BPS92a]:

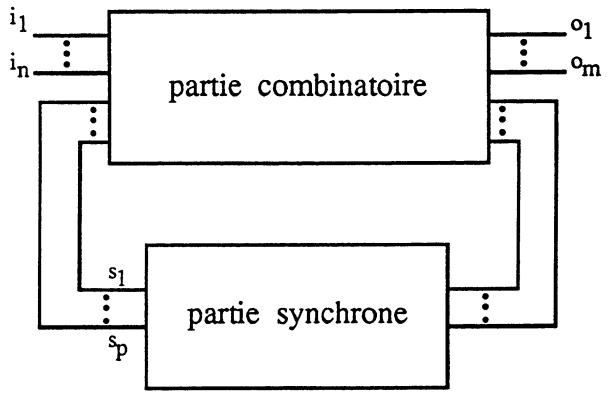
Soient X et Y deux signaux définis dans une architecture ou dans une liste de ports de l'entité correspondante. Nous dirons que X dépend de Y si Y apparait soit en partie droite de l'affectation de X, soit dans l'expression de contrôle de l'affectation conditionnée.
Si les ports sont limitées aux directions IN et OUT, la relation de dépendance définit un pré-ordre partiel pour lequel les ports d'entrée sont les plus grands éléments, et les ports de sorties les plus petits éléments. Plus précisément, dans la construction du graphe de la relation de dépendance, les arêtes correspondant aux affectations conditionnées sont étiquetées avec leurs conditions. Les fausses boucles sont éliminés en imposant que les seuls chemins qui peuvent être pris sont ceux pour lesquels la conjonction des étiquettes trouvées en parcourant les arêtes n'est pas égale à '0'. Cette méthode est utilisée dans le système CASCADE [FP84] pour trouver un ordre d'évaluation en une seule passe (parmi plusieurs ordres possibles), pour les descriptions flots de données, équivalent à l'algorithme de stabilisation à passes multiples de VHDL.

Dans une description comportementale P-VHDL, les variables locales peuvent être déclarées dans un processus combinatoire, les affectations de ces variables sont exécutées en séquence, dans l'ordre lexical des instructions d'affectations. Ces variables sont rémanentes. Elles sont créées à l'initialisation du processus et gardent leurs valeurs lorsqu'il devient inactif; lorsque celui-ci redevient actif, toutes les variables reprennent leurs dernières valeurs. Par conséquent, un processus permet de décrire une mémoire.

En P-VHDL, nous avons imposé que toutes les variables doivent être affectées avant qu'elles ne soient utilisées. De plus, seuls les signaux faisant partie de la liste de sensibilité sont autorisés à affecter les variables. Donc, à chaque cycle de simulation toutes les variables reçoivent des nouvelle valeurs qui effacent les anciennes. Par conséquent une variable sous ces condition ne peut pas représenter un élément de mémoire.

B. Les circuits séquentiels

Les circuits séquentiels synchrones sont représentés généralement par le schema de la figure (2.6)[Ko78].



Figure(2.6): Le circuit synchrone

Le circuit a un nombre fini d'entrées (i_1, ..., i_n) et un nombre fini de sorties (o_1,... , o_m) . Les variables d'état sont les sorties des éléments de mémorisation(s_1,....., sp). Les entrées du circuit et les variables d'état sont fournies à la partie combinatoire pour calculer les sorties, et les entrées de la partie synchrone à l'instant t. Les valeurs des variables d'état à l'instant t+1 sont calculées par la partie synchrone. La synchronisation est assurée par le signal d'horloge, et pour garantir le bon fonctionnement du circuit, il faut que le retard de propagation dans la partie combinatoire soit infèrieur à la période d'horloge.

En P-VHDL, le seul signal qui a un retard temporel associé à son changement est l'horloge principale; ce signal se distingue par un type spécial, le type **clock**. Par conséquent, le retard physique dans la partie combinatoire est égal à zéro, et une horloge peut avoir une période quelconque. Les conditions de synchronisation dans les processus et les blocs synchrones deviennent vrai une seule fois à chaque front montant de l'horloge, et ce changement a lieu pendant le premier delta. Donc les valeurs des signaux d'état (les registres) seront changées une fois seulement pour chaque cycle de horloge, et ces changements seront séparés par une période d'horloge. Les valeurs des autres signaux changent pendant la même unité physique de temps, grâce à l'absence de la primitive **after**. Un ou plusieurs cycles (deltas) sont demandés ensuite pour calculer les valeurs finales (stables) des ces signaux.

Par conséquent, la relation de conformité entre deux architectures décrivant un circuit synchrone est réduite à:

$$Impl(I(t),S(t)) \; \boldsymbol{R} \; Spec(I(t),L(t))$$

où

- $S(t+1) = F\text{-}Impl(I(t),S(t))$
- $L(t+1) = F\text{-}Spec(I(t), L(t))$
- $I(t) = \langle i_1(t), \ldots\ldots, i_n(t) \rangle$ les entrées du circuit à l'instant t
- $S(t) = \langle s_1(t), \ldots\ldots, s_n(t) \rangle$ les variables d'état de la réalisation
- $L(t) = \langle l_1(t), \ldots\ldots, l_n(t) \rangle$ les variables d'état de la spécification

Le paragraphe précèdent justifie l'usage de la sémantique des circuits synchrones comme modèle formel pour P-VHDL sous deux conditions:

1. Le signal d'horloge n'est pas utilisé dans la partie combinatoire.
2. Le circuit n'utilise pas des horloges dérivées.

Pour pouvoir décrire des circuits synchrones qui ne respectent pas ces deux conditions, P-VHDL devrait être étendu. La génération d'horloges secondaires se ferait dans la partie combinatoire, à l'aide d'affectations à retard physique non nul. En particulier, une horloge dérivée considérée comme synchrone avec l'horloge principale devrait en réalité être décalée d'un retard physique minimal (en VHDL une femto-seconde), afin de garantir le chargement des variables d'état pendant le premier delta[DJ92]. Et l'usage de l'horloge principale dans la partie combinatoire doit être faite par l'intermediare d'un signal équivalent à cette horloge mais décalé avec un retard minimal pour assurer que le changement dans la partie combinatoire a été stabilisé avant le changement d'horloge.

Chapitre 3

Exemple d'Application: La Vérification Formelle du Multiplieur en ligne JANUS

Dans ce chapitre, nous vérifions formellement le bon fonctionnement du multiplieur en ligne JANUS [GHM89]. Ce multiplieur est construit au laboratoire TIM3-IMAG pour effectuer des multiplications à haute précision. Le fonctionnement de JANUS est basé sur l'arithmétique en-ligne [EV77] où les nombres circulent des poids forts vers les poids faibles. L'arithmétique en-ligne permet le "pipelining" qui accélère le calcul. De plus, la circulation de donnée des poids forts vers les poids faibles est nécessaire pour réaliser quelques types de calcul tels que la division ou la comparaison de deux nombres. L'arithmétique en-ligne utilise des systèmes de numération redondants pour éviter la propagation de retenue associée avec le système classique de numération.

Nous avons choisi JANUS comme un exemple à prouver par PREVAIL pour trois raisons:

1- C'est un circuit réel, il n'a pas construit spécialement pour la preuve.

2- Le système de numération utilisé dans JANUS n'est pas simple. Par conséquent, le choix d'une spécification n'est pas une tâche facile.

3- JANUS est construit de manière modulaire. Un style mixte, flot de données et structurel, peut le décrire.

3.1 La représentation des nombres dans JANUS.

JANUS utilise une représentation des nombres avec des *chiffres signés*. Dans cette représentation chaque chiffre x est représenté par un couple de bits (a,b). La valeur de x est égale à b - a. Chaque chiffre peut représenter une des trois valeurs:

-1 = (1,0), 1 = (0,1) ou 0 qui a deux représentations (0,0) et (1,1).

Ce système de numération est redondant. En effet chaque nombre peut avoir plusieurs représentations.

Exemples:

3 = (0,0),(0,1),(0,1) ou (0,1),(0,0),(1,0) ou (0,1),(1,0),(0,0),(1,0) .....

-5 = (1,0),(0,0),(1,0) ou (1,0),(0,0),(0,1),(0,1) .....

3.2 La description de JANUS

Le multiplieur est construit en utilisant trois unités principales: l'additionneur parallèle à trois entrées (TIA), l'additionneur série à trois entrées (TISA) et le multiplieur SBDM. Les deux additionneurs sont basés sur un opérateur primaire appelé PPM (Plus Plus Moins).

3.2.1 L'opérateur PPM

L'opérateur PPM figure (3.1) a trois entrées binaires D, E et F et deux sorties Sout et Cout. La valeur de sortie de cet opérateur est calculée soit par l'expression (D + E - F), soit par l'expression (F - D - E). Le choix entre les deux expressions dépend du signe affecté aux entrées et aux sorties.

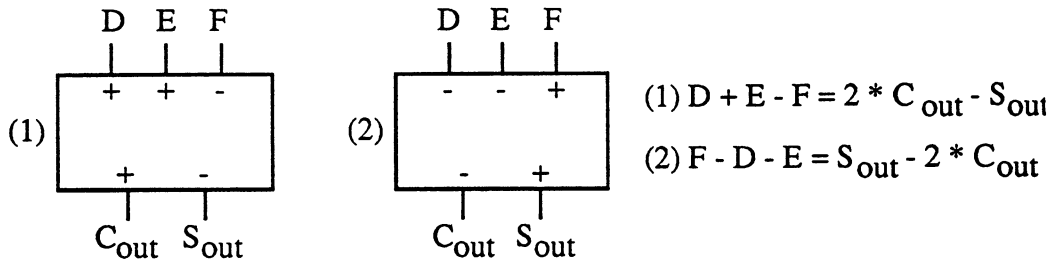


Figure (3.1): L'opérateur PPM

Les équations booléennes de cet opérateur sont:

Sout = D xou E xou F

Cout = (D et E) ou (D et non F) ou (E et non F)

L'entité PPM décrit l'interface de cet opérateur. Son modèle flot de données est décrit par l'architecture Dataflow.

```
entity PPM is
     port(D,E,F : in bit; Sout, Cout : out bit);
end PPM;

architecture Dataflow of PPM is
begin
      Sout <= D xor (F xor E);
      Cout <= (D and E) or ((not F) and D) or ((not F) and E);
end Dataflow;
```

Les deux fonctions suivantes expriment la sémantique de l'architecture PPM (Dataflow):

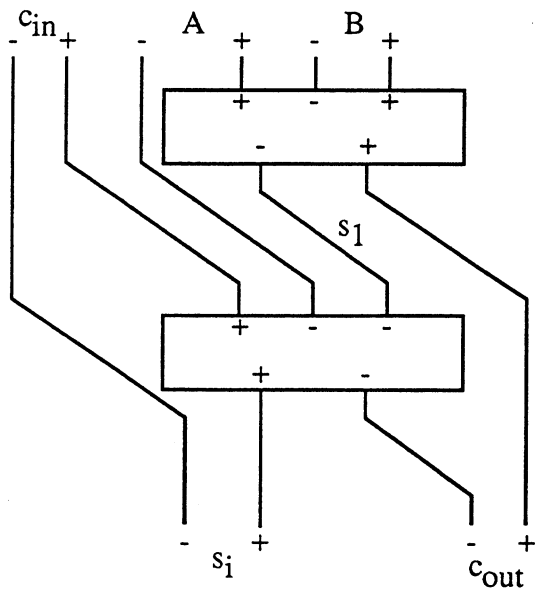
PPM-Dataflow-Sout = λ (D E F). (**xou** D (**xou** E F))

PPM-Dataflow-Cout = λ (D E F) . (**ou** (**ou** (**ou** (**et** D F) (**et** (**not** F)D)) (**et** (**non** F) E)))

3.2.2 L'additionneur parallèle à trois entrées (TIA)

L'additionneur TIA est construit en utilisant deux autres additionneurs: le premier est l'additionneur à retenue libre (CFA) et le second est l'additionneur à retenue stocké (CSA). Le CFA figure (3.2) utilise deux opérateurs PPM liés par la ligne S1. La sortie de cet additionneur est calculée par la formule:

$$2 * (Cout^{+} - Cout^{-}) + Si^{+} - Si^{-} = A^{+} - A^{-} + B^{+} - B^{-} + Cin^{+} - Cin^{-}$$



Figure(3.2): L'additionneur CFA

La description VHDL suivante décrit l'additionneur CFA. L'entité PPM est utilisée comme un composant et le signal S1 modélise la connexion entre les deux PPMs.

```
entity Cfa is
      port(Cinp,Cinm,Ap,Am,Bp,Bm : in bit;
            Sip,Sim,Coup,Coutm : out bit);
end Cfa;
architecture Structure of CFA is
component ppm_comp
      port(D,E,F : in bit;Sout,Cout : out bit);
end component;

signal S1 : bit;
begin
      Sim <= Cinm;
      CfaBlock : block for all : ppm_comp use entity PPM(Dataflow);
      begin
            ppm1: ppm_comp port map (Ap,Bp,Bm,S1,Coutp);
            ppm2: ppm_comp port map (Am,S1,Cinp,Sip,Coutm);
      end block;
end Structure;
```

Cinq fonctions modélisent la fonctionnalité de l'architecture CFA (Structure), ce sont:

Cfa-Structure-Sip = λ (Cinp Cinm Ap Am Bp Bm) .

PPM-Dataflow-Sout (Am, **CFA-Structure-S1** (Cinp,Cinm,Ap,Am,Bp,Bm), Cinp)

Cfa-Structure-Coutm = λ (Cinp Cinm Ap Am Bp Bm) .

PPM-Dataflow-Cout (am, **CFA-Structure-S1** (Cinp,Cinm,Ap,Am,Bp,Bm), Cinp)

Cfa-Structure-Coutp = λ (Cinp Cinm Ap Am Bp Bm) . **PPM-Dataflow-Cout**(Ap,Bp,Bm)

Cfa-Structure-S1 = λ (Cinp Cinm Ap Am Bp Bm) . **PPM-Dataflow-Sout**(Ap,Bp,Bm)

Cfa-Structure-Sim = λ (Cinp Cinm Ap Am Bp Bm) . Cinm

L'additionneur CSA figure (3.3) utilise deux opérateurs PPM non interconnectés. Les sorties de cet additionneur sont calculés selon une équation similaire à celle de l'additionneur CFA: $2 * (Cout^+ - Cout^-) + Si^+ - Si^- = A^+ - A^- + B^+ - B^- + C^+ - C^-$

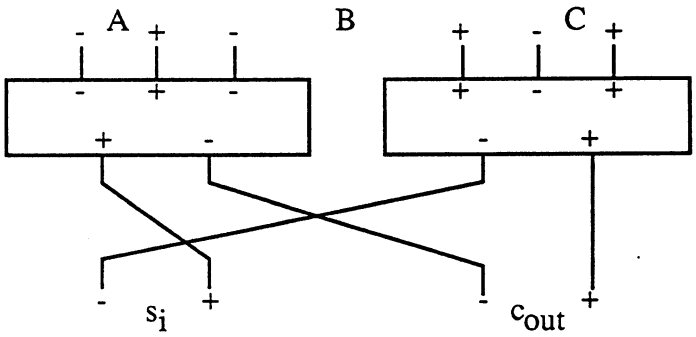


Figure (3.3): L'additionneur CSA

La description structurale suivante modélise l'additionneur CSA.

```
entity Csa is
      port(Ap,Am,Bp,Bm,Cp,Cm : in bit; Sip,Sim,Coutp,Coutm : out bit);
end Csa;

architecture Structure of Csa is

component ppm_comp
      port(D,E,F : in bit;Sout,Cout : out bit);
end component;

begin
      csablock : block for all : ppm_comp use entity PPM(Dataflow);
      begin
            ppm1 : ppm_comp port map(Bp, Cp, Cm, Sim, Coutp);
             ppm2 : ppm_comp port map(Am Bm,Ap,Sip,Coutm);
      end block;
end Structure;
```

Les quatres fonctions suivantes traduisent les deux instructions d'instanciation des composants ppm1 et ppm2 dans l'architecture Csa(Structure).

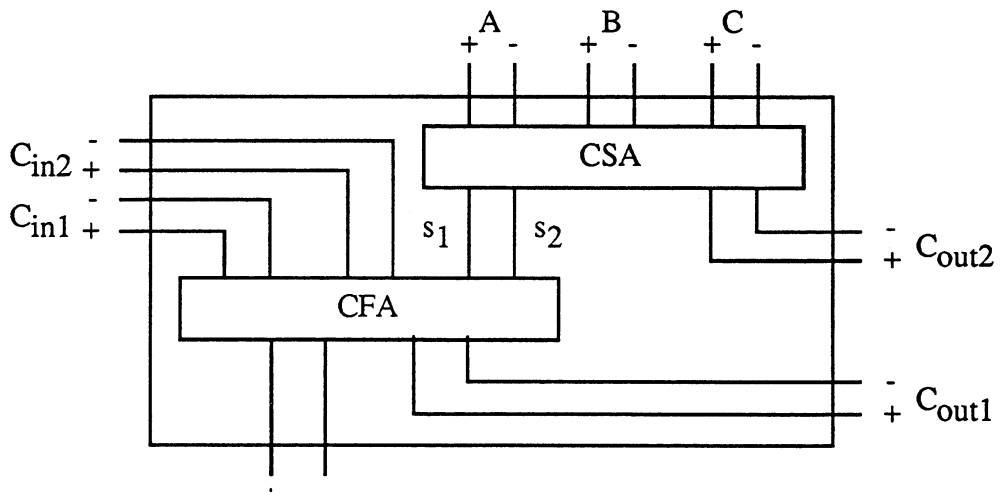
Csa-Structure-Sip = λ (Ap Am Bp Bm Cp Cm) . **PPM-Dataflow-Sout** (Am,Bm,Ap)
Csa-Structure-Coutm = λ (Ap Am Bp Bm Cp Cm) . **PPM-Dataflow-Cout** (Am,Bm,Ap)
Csa-Structure-Sim = λ (Ap Am Bp Bm Cp Cm) . **PPM-Dataflow-Sout** (Bp,Cp,Cm)
Csa-Structure-Coutp = λ (Ap Am Bp Bm Cp Cm) . **PPM-Dataflow-Cout** (Bp,Cp,Cm)

L'additionneur TIA figure(3.4) a en entrée cinq chiffres binaires signés, les trois chiffres et les deux retenues, et en sortie trois chiffres binaires signés, la somme et les deux retenues. Les sorties de cet additionneur sont calculées par la formule suivante:

$$S^+ - S^- + 2 * (Cout_1^+ - Cout_1^- + Cout_2^+ - Cout_2^-) =$$
$$A^+ - A^- + B^+ - B^- + C^+ - C^- + Cin_1^+ - Cin_1^- + Cin_2^+ - Cin_2^-$$

La description VHDL suivante décrit l'additionneur TIA. Les deux signaux S1 et S2 modélisent les deux connexions entre l'additionneur CSA et l'additionneur CFA.

```
entity Tia is
      port(Cin1p,Cin1m,,Cin2m,Ap,Am,Bp,Bm,Cp,Cm : in bit;
            Sip,Sim,Cout1p,Cout1m,Cout2p,Cout2m : out bit);
end Tia;
```



Figure(3.4): L'additionneur TIA

```
architecture Structure of Tia is

component csa-comp
      port (Ap,Am,Bp,Bm,Cp,Cm : in bit; Sip,Sim,Coutp,Coutm :out bit);
end component;

component cfa-comp
      port (Cinp,Cinm,Ap,Am,Bp,Bm : in bit; Sip,Sim,Coutp,Coutm : out bit);
end component;

signal S1,S2 : bit;
begin
      TiaBlock : block
            for all : csa-comp use entity Csa(Structure);
            for all : cfa-comp use entity Cfa(Structure);
      begin
            csa1: csa-comp port map (Ap,Am,Bp,Bm,Cp,Cm,S1,S2,Cout2p,Cout2m);
            cfa1: cfa-comp portmap (Cin1p,Cin1m,Cin2p,Cin2m,S1,S2,Sip,Sim,Cout1p,Cout1m);
      end block;
end Structure;
```

Les deux fonctions suivantes modélisent de manière formelle le port de sortie Sip et le signal interne S1.

Tia-Structure-Sip =

λ (Cin1p Cin1m Cin2p Cin2m Ap Am Bp Bm Cp Cm) .

Cfa-Structure-Sip (Cin1p, Cin1m, Cin2p, Cin2m,

Tia-Structure-S1(Cin1p,Cin1m,Cin2p,Cin2m,Ap,Am,Bp,Bm,Cp,Cm),

Tia-Structure-S2(Cin1p,Cin1m,Cin2p,Cin2m,Ap,Am,Bp,Bm,Cp,Cm))

Tia-Structure-S1 = λ (Cin1p Cin1m Cin2p Cin2m Ap Am Bp Bm Cp Cm) .

Csa-Structure-Sip(Ap,Am,Bp,Bm,Cp,Cm)

3.2.3 L'additionneur série à trois entrées(TISA)

L'additionneur série à trois entrées (le chiffre de poids fort d'abord) est construit avec quatre opérateurs PPM et cinq bascules de type D figure (3.5), où A, B et C sont fournis en série et le premier chiffre du résultat S est obtenu une période d'horloge après l'arrivée des premières entrées.

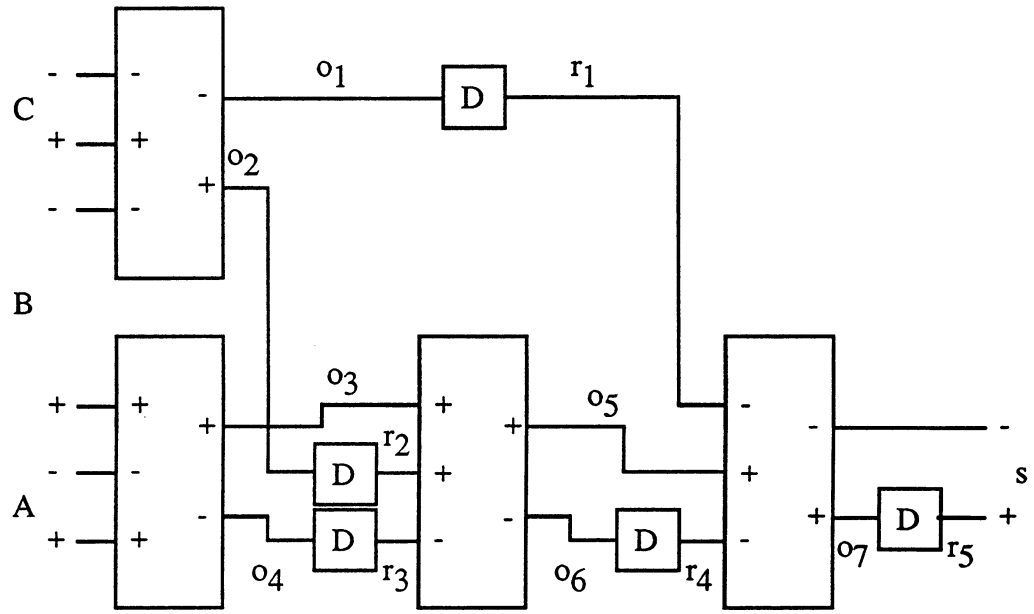


Figure (3.5): L'additionneur TISA

La spécification de cet additionneur est :

$$\Sigma^{n}_{m=0} 2^{n-m+2} (S^{+}(t+m+1) - S^{-}(t+m+1)) = \Sigma^{n}_{m=0} 2^{n-m} (A^{+}(t+m) - A^{-}(t+m)) +$$
$$\Sigma^{n}_{m=0} 2^{n-m} (B^{+}(t+m) - B^{-}(t+m)) + \Sigma^{n}_{m=0} 2^{n-m} (C^{+}(t+m) - C^{-}(t+m))$$

Le signal de l'horloge Clk a le type clock. Les bascules sont modélisés par des affectations gardées par l'expression : (Clk ='1' and not Clk'stable). L'entité Tisa et l'architecture Impl décrivent cet additionneur.

```
entity Tisa is
      port(Ap,Am,Bp,Bm,Cp,Cm:  in bit; Clk : in clock; Sp,Sm :out bit);
end Tisa;

architecture Impl of Tisa is
component ppm-comp
      port(D,E,F : in bit; Sout,Cout : out bit);
end  component;
signal o: bit_vector(1 to 7);
signal r : bit_reg(1 to 5);

begin
      ppm-block : block
      for all : ppm-comp use entity PPM(Dataflow);
      begin
            ppm1 : ppm-comp port map (Bm, Cm, Cp, o(2), o(1));
            ppm2 : ppm-comp port map (Ap, Bp, Am, o(4), o(3));
            ppm3 : ppm-comp port map (r(2), o(3), r(3), o(6), o(5));
            ppm4 : ppm-comp port map (r(4), r(1), o(5), o(7), Sm);
      end block;
```

```
        FlipFlopBlock : block (Clk = '1' and not Clk'stable)
        begin
                r(2) <= guarded o(2);
                r(3) <= guarded o(4);
                r(4) <= guarded o(6);
                r(1) <= guarded o(1);
                r(5) <= guarded o(7);
        end block;
        Sp <= r(5);
end Impl;
```

La fonction Tisa-Impl-r<5> modélise le signal interne r(5). Ce signal reçoit l'ancienne valeur, valeur à l'instant t - la période de l'horloge , du signal o(7).

Tisa-Impl-r<5> = λ (Ap Am Bp Bm Cp Cm Tisa-Impl-r<1:5>) .

P Tisa-Impl-o<7>(Ap,Am,Bp,Bm,Cp,Cm,Tisa-Impl-r<1:5>)

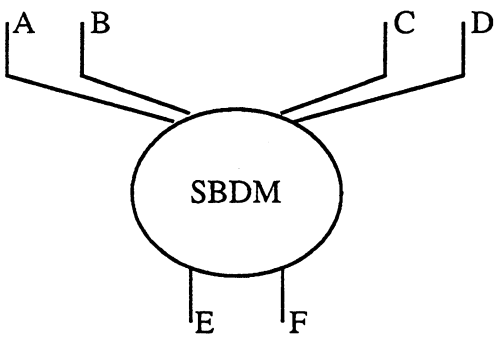
3.2.4 Le multiplieur SBDM

Ce multiplieur figure (3.6) a quatre entrées et deux sorties. Les quatre entrées correspondent aux deux chiffre signés qui vont être multipliés, les deux sorties sont le résultat de la multiplication.

Les équations logiques de ce multiplieur sont:

E = non((A et C) ou (B et D))

F = non((A et D) ou (B et C))



Figure(3.6): Le multiplieur d'un chiffre signé

La description VHDL correspondante est:

```
entity Sbdm is
        port (A,B,C,D: in bit; Rp,Rm : out bit);
end Sbdm;

architecture Dataflow of Sbdm is
begin
        Rm <= not((A and C) or (B and D));
        Rp <= not((A and D) or (B and C));
end Dataflow;
```

Les deux fonctions suivantes modélisent l'architecture SBDM(DataFlow):

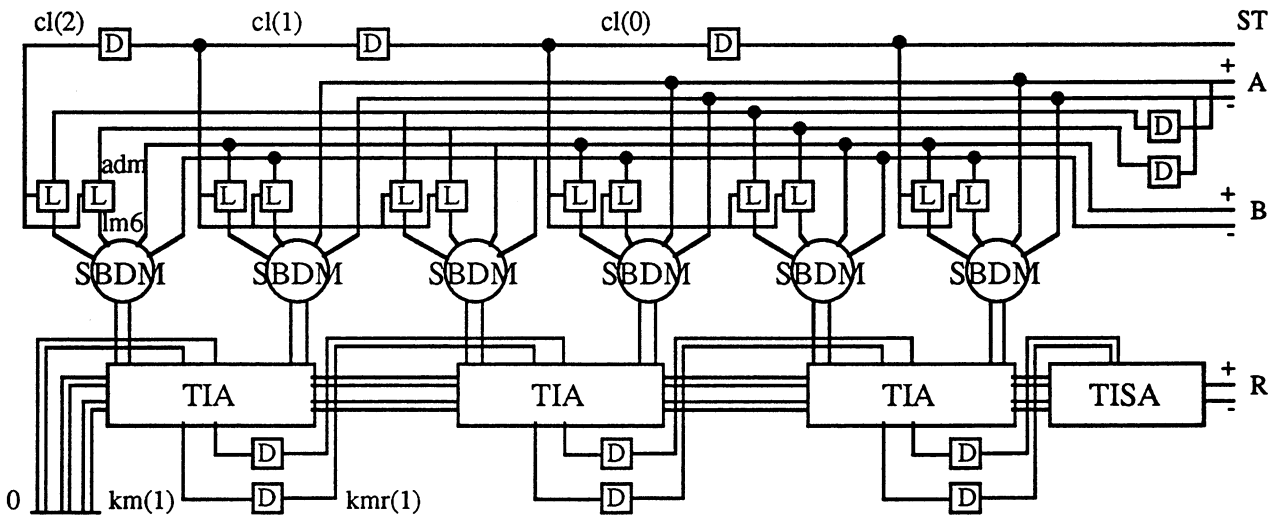
Sbdm-Dataflow-Rp = λ(A B C D) . **non(ou (et** A D)(**et** B C)))

Sbdm-Dataflow-Rm = λ(A B C D) . **non(ou (et** A C)(**et** B D)))

La spécification du multiplieur est : F - E = (B - A) * (D - C)

3.2.5 Le multiplieur en ligne JANUS

La figure (3.7) représente le multiplieur JANUS à trois chiffres. Le multiplieur est construit avec trois cellules; chacune se compose d'un additionneur TIA, de deux bascules de types D, de deux multiplieurs SBD et de quatre latchs. Les sorties de la dernière cellule sont connectées à l'additionneur série TISA. Le signal ST est mis à 1 au début du calcul puis remis à 0 tout de suite après. Le premier chiffre du résultat sera retardé de deux périodes d'horloge par rapport aux entrées. La longueur du résultat de la multiplication est de sept chiffres.



Figure(3.7): Le multiplieur JANUS

La spécification du JANUS est :

$$\sum^{2}_{m=0} 2^{2-m} (A^{+}(t+m) - A^{-}(t+m)) * \sum^{2}_{m=0} 2^{2-m} (B^{+}(t+m) - B^{-}(t+m)) = \sum^{6}_{m=0} 2^{6-m} (R^{+}(t+m) - R^{-}(t+m))$$

La description VHDL du multiplieur JANUS est donnée par la figure (3.8). Les 11 bascules sont modélisées par le bloc Bascules, la condition de garde dans ce bloc est:

clk ='1' and not clk'stable.

Les dix latchs sont représentés par quatre blocs: LatchSt, LatchCl, LatchCl1 et LatchCl2. La condition de garde de ces blocs est la conjonction de la condition de synchronisation et du signal de contrôle du latch. Le bloc LatchSt modélise les deux latchs contrôlés par le signal st. Les quatre latchs chargés sous le contrôle du signal cl(0) sont modélisés par le bloc LatchCl0. Les deux blocs LatchCl1 et LatchCl2 modélisent les latchs chargés sous le contrôle des signaux cl(1) et cl(2) respectivement. Les deux vecteurs de bit ip et im modélisent les sorties de six multiplieurs SBDM. Les sorties des trois additionneurs TIA sont modélisées par les deux vecteurs kp et km. Les sorties de latchs sont représentées par les vecteurs de registres lp et lm.

La sémantique de l'architecture Janus(Stru) est modélisée par un ensemble de fonctions qui ont comme paramètres:

- les ports d'entrée de l'entité Janus: ap, am, bp, bm et st;
- les 22 variables d'état de l'architecture Stru: les signaux adp , adm , les vecteurs kpr, kmr, lp, lm, cl et les cinq registres internes de l'additionneur série TISA.

Les trois fonctions suivantes font partie de cet ensemble: la fonction Janus-Stru-rp modélise le port de sortie rp; le corps de cette fonction est un appel à la fonction Tisa-Impl-sp qui représente l'architecture Tisa(Impl). Le latch lp<1> est modélisé par la fonction Janus-Stru-lp<1>. La bascule cl(0) est modélisée par la fonction Janus-Str-cl<0>; le corps de la fonction est une application de l'opérateur P sur st.

```
Janus-Stru-rp =
  λ (ap am bp bm st Janus-Stru-kpr<1:3> Janus-Stru-kmr<1:3> Janus-Stru-lp<1:6>
    Janus-Stru-lm<1:6> Janus-Stru-cl<0:2> Janus-Stru-adp Janus-Stru-adm Janus-Stru-Sad<1:5>) .
      Tisa-Impl-Sp (Janus-Stru-kp<8>, Janus-Stru-km<8>, Janus-Stru-kp<9>,
                    Janus-Stru-km<9>,Janus-Stru-kpr<3>, Janus-Stru-kmr<3>, Janus-Stru-Sad<1:5>)
```

```
entity Janus is
    port(ap,am,bp,bm,st : in bit; clk : in clock; rp,rm : out bit );
end Janus;

architecture Stru of Janus is

component Tisa port(ap,am,bp,bm,cp,cm : in bit;clk : in clock; sp,sm : out bit); end component;
component Tia port(cin1p,cin1m,cin2p,cin2m,ap,am,bp,bm,cp,cm : in bit;
                   sip,sim,cout1p,cout1m,cout2p,cout2m : out bit); end component;
component Sbdm port (a,b,c,d : in bit; rp,rm : out bit); end component;
signal cl:reg_vector(0 to 2); signal adp,adm: reg; signal ip,im, : bit_vector(1 to 6);
signal kpr,kmr:reg_vector(1 to 3);lp,lm :reg_vector(1 to 6); signal kp,km:reg_vector(1 to 9)

begin
LatchSt:block(clk = '1' and not clk'stable and st = '1')
    begin
    lp(1) <= guarded bp; lm(1) <= guarded bm;
    end block;
LatchCl0: block(clk = '1' and not clk'stable and cl(0) = '1')
    begin
    lp(3) <= guarded bp; lm(3) <= guarded bm; lp(2) <= guarded adp; lm(2) <= guarded adm;
    end block;
LatchCl1: block(clk = '1' and not clk'stable and cl(1) = '1')
    begin
    lp(4) <= guarded adp; lm(4) <= guarded adm; lp(5) <= guarded bp; lm(5) <= guarded bm;
    end block;
LatchCl2: block(clk = '1' and not clk'stable and cl(2) = '1')
    begin
    lp(6) <= guarded adp; lm(6) <= guarded adm;
    end block;
sbdblock: block for all : Sbdm use entity sbdm(dataflow);
    begin
    s1:Sbdm port map (am,ap,lm(1),lp(1),ip(1),im(1));
    s2:Sbdm port map (bm,bp,lm(2),lp(2),ip(2),im(2));
    s3:Sbdm port map (am,ap,lm(3),lp(3),ip(3),im(3));
    s4:Sbdm port map (bm,bp,lm(4),lp(4),ip(4),im(4));
    s5:Sbdm port map (am,ap,lm(5),lp(5),ip(5),im(5));
    s6:Sbdm port map (bm,bp,lm(6),lp(6),ip(6),im(6));
    end block;
tiablock: block for all : TIA use entity TIA(Stru);
    begin
    a1:Tia port map('0','0','0','0','0','0',ip(6),im(6),ip(5),im(5),kp(1),km(1),kp(2),km(2),kp(3),km(3));
    a2:Tia port map (kp(2), km(2), kp(3), km(3), kpr(1), kmr(1), ip(4),
                     im(4), ip(3), im(3),kp(4),km(4),kp(5),km(5),kp(6),km(6));
    a3:Tia port map(kp(5),km(5),kp(6),km(6),kpr(2), kmr(2),ip(2),im(2),
                    ip(1),im(1),kp(7),km(7),kp(8),km(8),kp(9),km(9));
    end block;
Bascules: block(clk = '1' and not clk'stable)
    begin
    cl(0) <= guarded st; cl(1) <= guarded cl(0); cl(2) <= guarded cl(1);adp <= guarded ap;
    adm <= guarded am;kpr(1) <= guarded kp(1); kmr(1) <= guarded km(1);kpr(2) <= guarded kp(4);
    kmr(2) <= guarded km(4); kpr(3) <= guarded kp(7); kmr(3) <= guarded km(7);
    end block;
SerialAdder: block for all : Tisa use entity Tisa(Impl);
    begin
    Sad:Tisa port map(kp(8),km(8),kp(9),km(9),kpr(3),kmr(3),clk,rp,rm);
    end block;
end Stru;
```

Figure(3.8): La description VHDL de JANUS

Janus-stru-lp<1>=

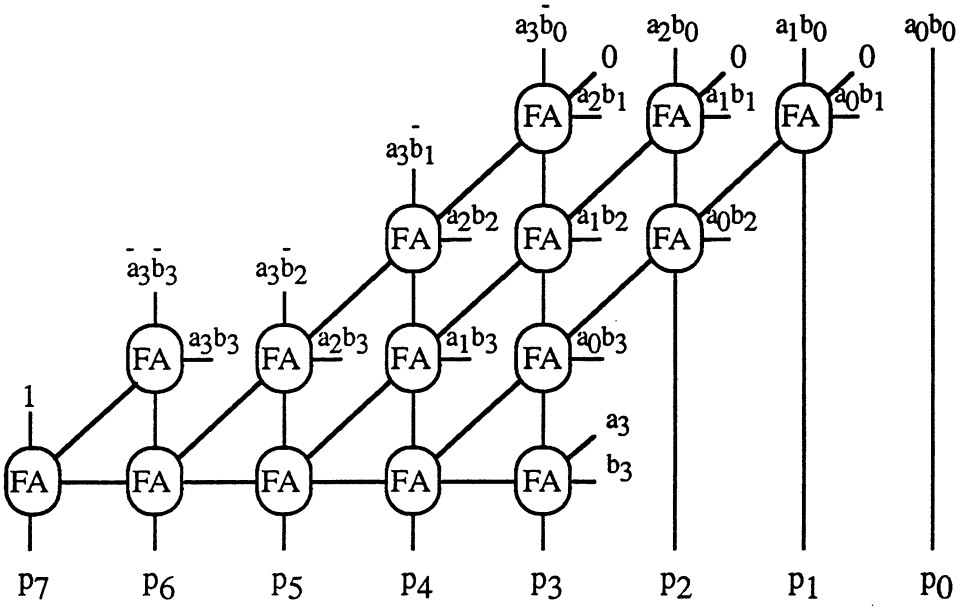
λ (ap am bp bm st Janus-Stru-kpr<1:3> Janus-Stru-kmr<1:3> Janus-Stru-lp<1:6>
Janus-Stru-lm<1:6> Janus-Stru-cl<0:2> Janus-Stru-adp Janus-Stru-adm Janus-Stru-Sad<1:5>) .
si P st alors P bp sinon P Janus.Stru.lp<1>

Janus-stru-cl<0>=

λ (ap am bp bm st Janus-Stru-kpr<1:3> Janus-Stru-kmr<1:3> Janus-Stru-lp<1:6>
Janus-Stru-lm<1:6> Janus-Stru-cl<0:2> Janus-Stru-adp Janus-Stru-adm Janus-Stru-Sad<1:5>) . P st

3.3 La spécification de JANUS

Le multiplieur de Baugh et Wooley alimenté par la transformation en complément à deux des entrées en série de JANUS, est considéré comme spécification. Le multiplieur de Baugh et Wooley est présenté dans la figure (3.9).



Figure(3.9): le multiplieur de Baugh et Wooley

Il utilise un additionneur 1-bit comme unité primaire où:

. a0, a1, a2, a3 sont les quatre bits du multiplicateur,

. bo, b1, b2, b3 sont les quatre bits du multiplicande,

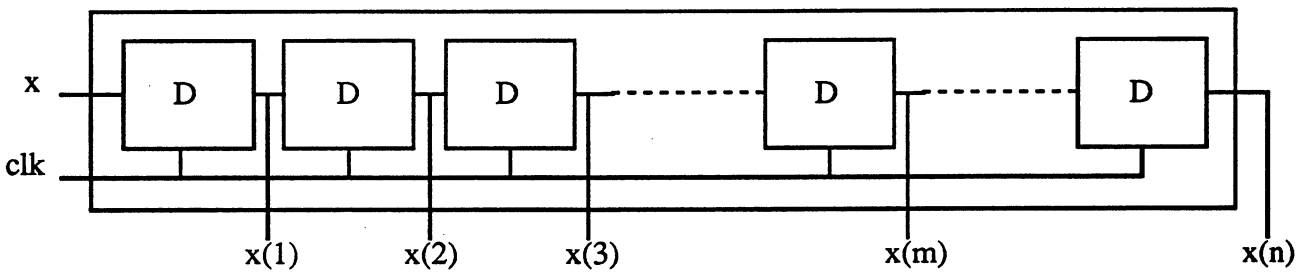
. p1, ..., p7 sont le résultat de la multiplication.

La description VHDL figure (3.10) décrit ce multiplieur. L'entité full_adder modélise l'unité primaire FA. L'architecture impl de l'entité baugh_wooley utilise cette entité comme composant.

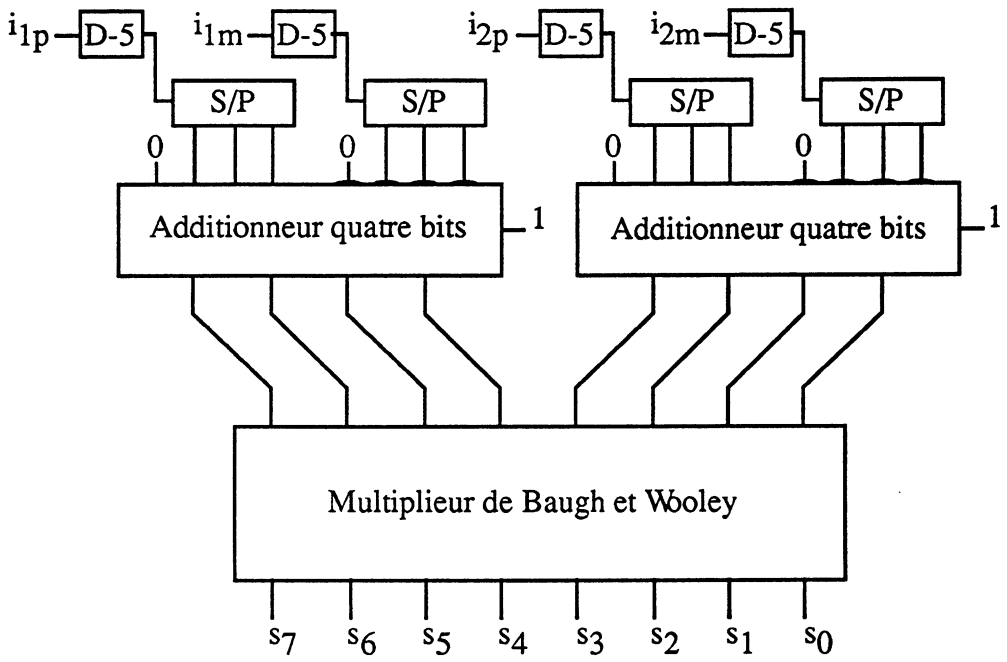
Les entrées séries de JANUS ((i1p,i1m), (b1p,b1m)) retardées de cinq périodes d'horloge sont fournies à deux convertisseurs série/parallèle trois bits, figure(3.11), dont les sorties alimentent deux additionneurs quatre bits qui transforment les chiffres signés en nombre codé en complément à deux. Les deux nombres en complément à deux sont multipliés par le multiplieur de Baugh et Wooley. Les sorties de ce multiplieur sont considérées comme spécification fonctionnelle de JANUS, figure(3.12).

```
entity baugh_wooley is
        port(a : in bit_vector(0 to 3); b : in bit_vector(0 to 3); p : out bit_vector(0 to 7));
end baugh_wooley;

architecture impl of baugh_wooley is

signal i1,i2,i3,o1,o2 : bit_vector(0 to 14);
signal carry:bit;

component full_adder
        port (x,y,cin : in bit;sum,cout : out bit);
end component;

begin
Mult: block for all:full_adder use entity full_adder(impl);
  begin
        p(0) <= a(0) and b(0);i1(0) <= a(0) and b(1);
        i2(0) <= a(1) and b(0); i3(0) <= '0';
        f0:full_adder port map(i1(0),i2(0),i3(0),p(1),o2(0));
        i1(1) <= a(1) and b(1); i2(1) <= a(2) and b(0); i3(1) <= '0';
        f1: full_adder port map(i1(1),i2(1),i3(1),o1(1),o2(1));
        i1(2) <= a(2) and b(1);i2(2) <= a(3) and not b(0);i3(2) <= '0';
        f2: full_adder port map(i1(2),i2(2),i3(2),o1(2),o2(2));
        i1(3) <= a(0) and b(2);i1(4) <= a(1) and b(2);
        f3: full_adder port map(i1(3),o2(0),o1(1),p(2),o2(3));
        f4:full_adder port map(i1(4),o2(1),o1(2),o1(4),o2(4));
        i1(5) <= a(2) and b(2); i3(5) <= a(3) and not b(1);i1(6) <= a(0) and b(3);
        f5:full_adder port map(i1(5),o2(2),i3(5),o1(5),o2(5));
        f6:full_adder port map(i1(6),o2(3),o1(4),o1(6),o2(6));
        i1(7) <= a(1) and b(3);i1(8) <= a(2) and b(3);i3(8) <= a(3) and not b(2);
        f7:full_adder port map(i1(7),o2(4),o1(5),o1(7),o2(7));
        f8:full_adder port map(i1(8),o2(5),i3(8),o1(8),o2(8));
        i1(9) <= a(3) and b(3); i2(9) <= not a(3); i3(9) <= not b(3);
        i1(10) <= a(3); i3(10) <= b(3);i3(14)<= '1';
        f9:full_adder port map(i1(9),i2(9),i3(9),o1(9),o2(9));
        f10:full_adder port map(i1(10),o1(6),i3(10),p(3),o2(10));
        f11:full_adder port map(o2(10),o2(6),o1(7),p(4),o2(11));
        f12:full_adder port map(o2(11),o2(7),o1(8),p(5),o2(12));
        f13:full_adder port map(o2(12),o2(8),o1(9),p(6),o2(13));
        f14:full_adder port map(o2(13),o2(9),i3(14),p(7),carry);
  end block;

end impl;
```

Figure(3.10): La description VHDL du multipleur de Baugh et Wooley



Figure(3.11): Convertisseur série/parallèle(S/P)



Figure(3.12): Le circuit de la spécification

Nous définissons une entité de vérification Verif. Cette entité a comme ports d'entrée les deux chiffres signés fournis en entrée de JANUS et comme ports de sortie le résultat de la multiplication de ces deux chiffres après conversion en un nombre binaire codé en complément à deux.

```
entity Verif is
        port(i1p,i1m,i2p,i2m : in bit; clk: in clock;
                s : out bit_vector(0 to 7));
end Verif;
```

L'entité Verif a deux architectures: Spec qui décrit le circuit spécifiant Janus figure (3.12), et Impl qui décrit le circuit contenant Janus figure(3.15).

Dans l'architecture Spec figure(3.13), le bloc synchrone S_P décrit les quatre convertisseurs série/parallèle et les quatre lignes de retard D-5. Les deux instances du composant ad4 dans le bloc d'instanciation ConvSigBin transforment les chiffres signés en complément à deux. L'instance C3 du composant baugh_wooley exécute la multiplication.

La sémantique de l'architecture Verif(Spec) est modélisée par un ensemble de fonctions qui ont comme paramètres:

- les ports d'entrée de l'entité Verif: i1p, i1m, i2p et i2m;
- les 32 variables d'état de l'architecture Spec: les vecteurs ap, am, bp, bm et les quatre vecteurs i1pr, i1mr, i2pr et i2mr.

Les quatre fonctions Verif-Spec-s<1>, Verif-Spec-a<1>, Verif-Spec-ap<1> et Verif-Spec-ap<2> font partie de cet ensemble figure(3.14).

Le corps de la fonction Verif-Spec-s<1> est un appel à la fonction baugh_wooley-impl-p<1> qui modélise le premier élément de la sortie vectorielle p de l'entité baugh_wooley calculé par l'architecture impl. Le corps de la fonction Verif-Spec-a<1> est un appel à la fonction ad4-impl-c<1> qui décrit une sortie du composant ad4.
Les deux fonctions Verif-Spec-ap<1> et Verif-Spec-ap<2> modélisent les deux registres ap(1) et ap(2). Elles utilisent l'opérateur passé pour exprimer les anciennes valeurs des signaux sources dans les deux affectations gardées.

```
architecture Spec of Verif is

component baugh_wooley
    port(a : in bit_vector(0 to 3); b:in bit_vector(0 to 3);p : out bit_vector(0 to 7));
end component;
component ad4
    port(a : in bit_vector(0 to 3); b : in bit_vector(0 to 3); cin : in bit; c : out bit_vector(0 to 3));
end component;
signal a,b,nam,nbm:bit_vector(0 to 3);signal carry:bit;
signal ap,am,bp,bm:reg_vector (0 to 3);
signal i1pr,i1mr,i2pr,i2mr: reg_vector(0 to 5);

begin
S_P:  block(clk = '1' and not clk'stable)
        begin
        i1pr(5) <= guarded i1pr(4);  i1pr(4) <= guarded i1pr(3) ;
        i1pr(3) <= guarded i1pr(2) ; i1pr(2) <= guarded i1pr(1) ;
        i1pr(1) <= guarded i1p;
        i1mr(5) <= guarded i1mr(4);  i1mr(4) <= guarded i1mr(3) ;
        i1mr(3) <= guarded i1mr(2) ;i2pr(3) <= guarded i2pr(2) ;
        i1mr(2) <= guarded i1mr(1) ; i1mr(1) <= guarded i1m;
        i2pr(5) <= guarded i2pr(4);  i2pr(4) <= guarded i2pr(3) ;
        i2pr(2) <= guarded i2pr(1) ; i2pr(1) <= guarded i2p;
        i2mr(5) <= guarded i2mr(4);  i2mr(4) <= guarded i2mr(3) ;
        i2mr(3) <= guarded i2mr(2) ;
        i2mr(2) <= guarded i2mr(1) ; i2mr(1) <= guarded i2m;
        ap(2) <= guarded i1pr(5); am(2) <= guarded i1mr(5);
        bp(2) <= guarded i2pr(5); bm(2) <= guarded i2mr(5);
        ap(1) <= guarded ap(2); ap(0) <= guarded ap(1);
        am(1) <= guarded am(2); am(0) <= guarded am(1);
        bp(1) <= guarded bp(2); bp(0) <= guarded bp(1);
        bm(1) <= guarded bm(2); bm(0) <= guarded bm(1);
        ap(3) <= guarded '0'; am(3) <= guarded '0';
        bp(3) <= guraded '0'; bm(3) <= guarded '0';
end block;

carry <= '1'; nam <= not am; nbm <= not bm;
ConvSigBin: block
for all : ad4 use entity ad4(impl);
    begin
    C1:ad4 port map(ap,nam,carry,a);
    C2:ad4 port map(bp,nbm,carry,b);
    end block;
MultBl: block
for all : baugh_wooley use entity baugh_wooley(impl);
    begin
    C3:baugh_wooley port map(a,b,s);
    end block;
end Spec;
```

Figure (3.13): La description VHDL de la spécification.

```
Verf-Spec-s<1> =
   λ (i1p i1m I2p i2m Verif-Spec-ap<0:3> Verif-Spec-am<0:3> Verif-Spec-bp<0:3> Verif-Spec-bm<0:3>
    Verif-Spec-i1pr<0:5> Verif-Spec-i1mr<0:5> Verif-Spec-i2pr<0:5> Verif-Spec-i2mr<0:5>).
     baugh_wooley-impl-p<1>
        (Verif-Spec-a<0:3>      (i1p, i1m, I2p, i2m, Verif-Spec-ap<0:3>, Verif-Spec-am<0:3>,
                                Verif-Spec-bp<0:3>, Verif-Spec-bm<0:3>,Verif-Spec-i1pr<0:5>,
                                Verif-Spec-i1mr<0:5>, Verif-Spec-i2pr<0:5>,Verif-Spec-i2mr<0:5>),
        Verif-Spec-b<0:3>       (i1p, i1m, I2p, i2m, Verif-Spec-ap<0:3>, Verif-Spec-am<0:3>,
                                Verif-Spec-bp<0:3>, Verif-Spec-bm<0:3>,Verif-Spec-i1pr<0:5>,
                                Verif-Spec-i1mr<0:5>, Verif-Spec-i2pr<0:5>,Verif-Spec-i2mr<0:5>) )
Verif-Spec-a<1> =
   λ (i1p i1m I2p i2m Verif-Spec-ap<0:3> Verif-Spec-am<0:3> Verif-Spec-bp<0:3> Verif-Spec-bm<0:3>
     Verif-Spec-i1pr<0:5> Verif-Spec-i1mr<0:5> Verif-Spec-i2pr<0:5> Verif-Spec-i2mr<0:5>).
     ad4-impl-c<1>
        (Verif-Spec-ap<0:3>     (i1p, i1m, I2p, i2m, Verif-Spec-ap<0:3>,
                                Verif-Spec-am<0:3>,Verif-Spec-bp<0:3>,Verif-Spec-bm<0:3>,
                                Verif-Spec-i1pr<0:5> ,Verif-Spec-i1mr<0:5>, Verif-Spec-i2pr<0:5>,
                                Verif-Spec-i2mr<0:5>),
        Verif-Spec-nam<0:3> (i1p, i1m, I2p, i2m, Verif-Spec-ap<0:3>,
                                Verif-Spec-am<0:3>,Verif-Spec-bp<0:3>,Verif-Spec-bm<0:3>,
                                Verif-Spec-i1pr<0:5>,Verif-Spec-i1mr<0:5>, Verif-Spec-i2pr<0:5>,
                                Verif-Spec-i2mr<0:5>),
        Verif-Spec-carry        (i1p, i1m, I2p, i2m, Verif-Spec-ap<0:3>, Verif-Spec-am<0:3>,
                                Verif-Spec-bp<0:3>,Verif-Spec-bm<0:3>,Verif-Spec-i1pr<0:5>,
                                Verif-Spec-i1mr<0:5>, Verif-Spec-i2pr<0:5>,Verif-Spec-i2mr<0:5>))
Verif-Spec-ap<1> =
   λ (i1p i1m I2p i2m Verif-Spec-ap<0:3> Verif-Spec-am<0:3> Verif-Spec-bp<0:3> Verif-Spec-bm<0:3>
     Verif-Spec-i1pr<0:5> Verif-Spec-i1mr<0:5> Verif-Spec-i2pr<0:5> Verif-Spec-i2mr<0:5>).
     P Verif-Spec-ap<2>

Verif-Spec-ap<2> =
   λ (i1p i1m I2p i2m Verif-Spec-ap<0:3> Verif-Spec-am<0:3> Verif-Spec-bp<0:3> Verif-Spec-bm<0:3>
     Verif-Spec-i1pr<0:5> Verif-Spec-i1mr<0:5> Verif-Spec-i2pr<0:5> Verif-Spec-i2mr<0:5>) .
     P Verif-Spec-i1p
```

Figure (3.14): La sémantique de l'architecture Verif(Spec)

3.4 La preuve de l'équivalence entre la spécification et la réalisation

Les sorties séries de JANUS, fournies à deux convertisseurs sept bits, alimentent ensuite un additionneur huit bits qui transforme les chiffres signés en un nombre binaire codé en complément à deux. L'équivalence entre les fonctions représentant ces sorties et celles du circuit de la spécification, prouve formellement le bon fonctionnement de JANUS.

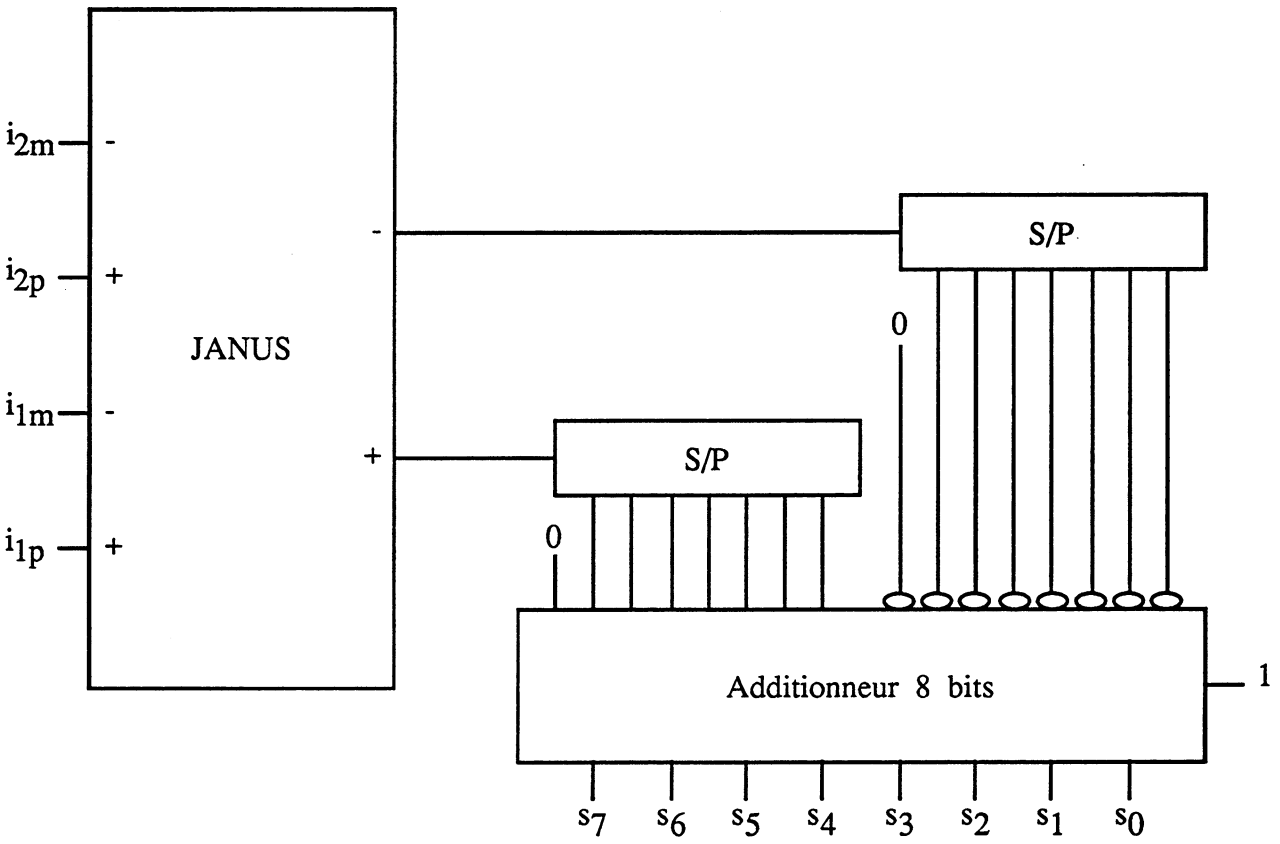


Figure (3.15): La réalisation du circuit de vérification

L'architecture Impl modélise ce circuit figure (3.16). Le bloc S_P décrit les deux convertisseurs série/parallèle. Le composant ad8 est utilisé pour transformer les deux chiffres signés en complément à deux. Le bloc MultBl contient l'instance à vérifier du composant JANUS.

```
architecture Impl of Verif is
component janus
        port(ap,am,bp,bm,st : in bit; clk: in clock;  rp,rm : out bit );
end component;
component ad8
        port(a : in bit_vector(0 to 7); b : in bit_vector(0 to 7); cin : in bit; c : out bit_vector(0 to 7));
end component;
signal carry,sp,sm:bit;
signal spp,smp:reg_vector (0 to 7);
signal nsmp:bit_vector(0 to 7);
begin
S_P: block(clk = '1' and not clk'stable)
begin
        spp(6) <= guarded sp; spm(6) <= guarded sm; spp(5) <= guarded spp(6);
        spp(4) <= guarded spp(5); smp(4) <= guarded smp(5); spp(3) <= guarded spp(4);
        smp(3) <= guarded smp(4);spp(2) <= guarded spp(3); smp(2) <= guarded smp(3);
        spp(1) <= guarded spp(2); smp(1) <= guarded smp(2);spp(0) <= guarded spp(1);
        smp(0) <= guarded smp(1);spp(7) <= guarded '0'; smp(7) <= guarded '0';smp(5) <= guarded smp(6);
end block;
carry <= '1'; nsmp <= not smp;
ConvSigBin: block for all: ad8 use entity ad8(impl);
begin
        C1: ad8 port map(spp,nsmp,carry,s);
end block;
MultBl:block for all: Janus use entity Janus(Stru);
begin
        c2: Janus port map(i1p,i1m,i2p,i2m,clk,sp,sm);
end block;
end Impl;
```

Figure (3.16): L'architecture Verif(Impl)

La fonction suivante modélise la sortie s(1) de l'entité Verif calculée par l'architecture Impl. Elle a comme paramètres les quatre ports d'entrée de l'entité, les deux vecteurs de variables d'état spp et smp ainsi que les variables d'état de l'exemplaire c2 du composant Janus utilisé dans l'architecture.

```
Verif-Impl-s<1> =
    λ   (i1p i1m I2p i2m Verif-Impl-spp<0:7> Verif-Impl-smp<0:7>Verif-c2-kpr<1:3> Verif-c2-kmr<1:3>
        Verif-c2-lp<1:6> Verif-c2-lm<1:6> Verif-c2-cl<0:2> Verif-c2-adp Verif-c2-admVerif-c2-Sad<1:5>) .
        ad8-Impl-c<1>
            (Verif-Impl-spp<0:7>,Verif-Impl-smp<0:7>,
            Verif-Impl-carry   (i1p,i1m, i2p, i2m,Verif-Impl-spp<0:7>,Verif-Impl-smp<0:7>,
                                    Verif-c2-kpr<1:3>, Verif-c2-kmr<1:3>, Verif-c2-lp<1:6,
                                    Verif-c2-lm<1:6>,Verif-c2-cl<0:2>,Verif-c2-adp,Verif-c2-adm,
                                        Verif-c2-Sad<1:5>))
```

Pour prouver que la réalisation de JANUS est équivalente à sa spécification, il faut prouver que

les deux fonctions modélisant le signal s dans les architectures Spec et Impl de l'entité Verif sont équivalentes. Autrement dit, il faut prouver que la relation suivante est une tautologie.

Verif-Impl-s<0:7> <=> Verif-Spec-s<0:7>

En fait, ces deux machines ne sont pas équivalentes à tout moment. Le circuit de spécification décrit le comportement de JANUS à un instant spécifique, celui de la fin du processus de multiplication de deux nombres. Les techniques classiques pour prouver l'équivalence entre deux machines synchrones ne sont pas applicables dans ce cas pour deux raisons:

1. Elles vérifient que les sorties des deux circuits sont identiques pour tous les états atteignables. Autrement dit, les résultats intermédiaires sont pris en compte.

2. Le nombre des variables d'état (32 pour la spécification, 38 pour la réalisation) est important. De ce fait, la taille de l'automate représentant la machine produite (Spécification X réalisation) est trop grande pour que l'automate puisse être manipulée par ces techniques (y compris les techniques symboliques).

Nous utilisons la technique du déroulement symbolique[Pa90] pour prouver JANUS. La preuve est effectuées en trois étapes:

1. Le déroulement symbolique

Nous déplions les deux fonctions vectorielles: Verif-Impl-s<0:7> et Verif-Spec-s<0:7>, en les simulant symboliquement pendant huit périodes d'horloge en partant de l'état initial suivant:

1. Tous les latchs doivent être initialisés à zéro.

2. Toutes les bascules sont mises à zéro.

3. Le signal st est mis à 1 au début du calcul, ensuite il est fixé à 0.

De plus, le multiplieur doit être alimenté par des zéros après l'arrivée du troisième chiffre pour calculer les sorties pour des entrées à trois chiffres.

2. La projection combinatoire

Nous éliminons l'opérateur temporel **P** en remplaçant chaque puissance de **P** appliquée à une variable par une nouvelle variable. Le résultat de cette opération est composé de deux vecteurs d'expressions booléennes, une pour chaque vecteur de fonctions. En fait, le comportement séquentiel du multiplieur est interprété comme répétition dans l'espace et il est transformé en comportement combinatoire où la notion de temps a disparu.

3. La démonstration de tautologie

L'équivalence entre les deux ensembles d'expressions booléennes est prouvée avec le démonstrateur de tautologies TACHE sur un Sparc-2 en 4 minutes.

Chapitre 4

Définition Formelle de P-VHDL

Dans ce chapitre, nous proposons une sémantique formelle pour P-VHDL. Le but de cette définition est de donner une spécification précise et complète de ce langage. A l'opposé du modèle formel qui nous avons associé à P-VHDL au chapitre 2, la sémantique définie ici est indépendante de tout outil, autrement dit elle n'est pas liée à une technique de preuve spécifique. Elle est destinée à assurer que la transition entre la syntaxe de ce sous-ensemble et les modèles formels utilisés par les outils de preuve, s'effectue de manière non ambiguë et cohérente. Nous choisissons la technique dénotationelle pour cette définition. Trois raisons justifient ce choix:

1. La sémantique dénotationnelle décrit la fonctionnalité des primitives des langages sans décrire comment elles doivent être réalisées.

2. Elle est compositionnelle. Une signification d'une primitive syntaxique composite est exprimée par les significations des éléments qui la composent.

3. Chaque élément syntaxique a une seule signification. Les problèmes d'ambiguïté, d'incohérence et d'incomplétude ne se présentent pas.

4. Les fonctions sémantiques sont des objets mathématiques, donc les techniques formelles peuvent être utilisées pour raisonner sur les propriétés des langages.

4.1 La sémantique dénotationnelle

Une définition dénotationelle d'un langage consiste en trois parties [Sc88]: la syntaxe abstraite, les domaines sémantiques et les fonctions d'évaluation.

La **syntaxe abstraite** est destinée à décrire la structure du langage. Dans ce type de syntaxe, chaque non-terminal dans la grammaire est représenté par un ensemble de phrases qui ont la structure spécifiée par la règle de grammaire de ce non-terminal. Cet ensemble est appelé un **domaine syntaxique.** Donc, un domaine syntaxique est une collection de valeurs ayant la même structure syntaxique.

Les ensembles qui sont utilisés comme espaces de valeurs dans un langage de programmation sont appelés les **domaines sémantiques**. Un domaine sémantique peut être un **domaine primitif** ou un **domaine composé**. Nous utilisons quatre types de domaines composés :

- Les domaines produit A X B.
- Les domaines union A + B.
- Les domaines de fonction A -> B.
- Les Treillis $A_{\perp}$, où $A_{\perp} = A \cup \{\perp\}$

Les **fonctions d'évaluation** font correspondre à chaque élément de la syntaxe abstraite une signification tirée des domaines sémantiques. Une fonction d'évaluation **X** (en geneva gras) pour un domaine syntaxique **X** (en times gras) est un ensemble d'équations, une équation pour chaque option dans la règle de grammaire de **X**.

Dans le reste de chapitre, nous donnons une définition dénotationnelle de P-VHDL. Les types définis dans le paquetage **proof** font partie de la syntaxe de ce langage.

4.2 La Syntaxe abstraite de P-VHDL

4.2.1 Les domaines syntaxiques

Une description P-VHDL est un couple formé d'une entité et d'une architecture. Les domaines syntaxiques **Entité** et **Architecture** représentent ces deux primitives respectivement. **U** et **A** sont les non-terminaux représentant des membres arbitraires de chaque domaine.

U ∈ Entité
A ∈ Architecture
AB ∈ ArchCorps

Une architecture est composée des déclarations et d'un corps. Le corps d'une architecture **AB** est composé des instructions concurrentes représentées par le domaine **InstructionConcurrente**.

C ∈ InstructionConcurrente

Trois types d'instructions concurrentes existent en **P-VHDL:** Les affectations concurrentes, les blocs et les processus. Le domaine **AffectationConcurrente** représente les instructions d'affectation concurrentes.

AC ∈ AffectationConcurrente

Deux types de bloc existent dans P-VHDL: Les blocs d'instanciation et les blocs séquentiels. Les deux domaines **BlocDInstanciation** et **BlocSéquentiel** représentent ces deux primitives respectivement.
Un BlocDinstanciation **BI** est composé des **InstructionDInstanciation.** Un BlocSéquentiel **BS** est une séquence **InstructionGardée.**

BI ∈ BlocDInstanciation
BS ∈ BlocSéquentiel
CI ∈ InstructionDInstanciation
M ∈ InstructionGardée

Deux types de processus existent en P-VHDL: Les **processus combinatoires** et les **processus séquentiels**. Un processus combinatoire **PC** est composé d'une séquence d'instructions séquentielles combinatoires. Un processus Synchrone **PS** est composé d'une séquence d'**instructions gardées**. Le domaine **InstProcFonc** représente les instructions qui se trouvent dans les procédures et les fonctions.

PS ∈ ProcessusSynchrone
PC ∈ ProcessusCombinatoire
SC ∈ InstructionSéquentielleCombinatoire
SG ∈ InstructionSéquentielleGardée
SP ∈ InstProcFonc

Déclaration est le domaine de toutes les déclarations autorisées dans une architecture P-VHDL. **DéclarationDeVariable** est le domaine syntaxique représentant l'instruction de déclaration de variables dans un processus, dans une fonction où dans une procédure.
Les spécifications de configuration définissant le lien entre les composants et les couples entité-architecture sont représentées par le domaine **SpécificationDeConfiguration.**

D ∈ Déclaration
V ∈ DéclarationDeVariable
CS ∈ SpécificationDeConfiguration

Type est le domaine syntaxique des types autorisés dans P-VHDL. Trois types de données font partie de la syntaxe: **bit**, **reg** et **clock**. P-VHDL supporte deux structures de données: **bit_vector** et **reg_vector**.
Expression est le domaine syntaxique des expressions. Les formes des expressions **E** sont: deux constantes, '0' et '1'; une opération unaire **not**, cinq opérations binaires: **and**, **or**, **xor**, **nand** et **nor**; l'indexation $I_1(IX)$; et l'invocation de fonctions $I' (E_1,... E_n)$.

Condition est le domaine syntaxique des conditions. Dans une condition, les opérations autorisées sont = , /= , **and** , **or** , **xor**, **nand**, **nor** et **not**.

T ∈ Type
E ∈ Expression
CD ∈ Condition

Indice est le domaine syntaxique des indices.
IX ∈ Indice

Numéral est le domaine des chiffres. **Identificateur** est le domaine des identificateurs autorisés dans VHDL.

I ∈ Identificateur
N ∈ Numéral

4.2.2 Les règles de grammaires

Une entité **U** décrit l'interface entre un couple (entité, architecture) et le monde extérieur. Dans P-VHDL, seuls les ports **in** et **out** sont autorisés.

U ::= **entity** I' **port** (I_1: in T_1; ...; I_m: in T_m; I_{m+1}:out T_{m+1}; ....; I_n : out T_n); **end** I';

Une architecture **A** est composée des déclarations **D** et d'un corps **AB**. Le corps d'une architecture est composé des instructions concurrentes **C**.

A ::= **architecture** I_1 **of** I_2 **is** D; **begin** AB; **end** I_1;

AB ::= C

C ::= AC | I: BI | I: BS | I : PC | I : PS | C_1 ; C_2

D ::= **signal** I: T |
procedure I' (I_1: in T_1; ...; I_n: in T_n; I_1:out T_1; ....; I_n : out T_n) **is**
V **begin** SP ; **end** I' |
function I' (I_1:T_1;..... I_n:T_n) **is** V; **begin** SP; end I' |
component I' **is port** (I_1: **in** T_1; ...; I_n: **in** T_n; I_1:**out** T_1; ....; I_n : **out** T_n);
end component |
D_1; D_2

Les trois règles suivantes donnent la syntaxe des types, des indices, des expressions et des conditions dans P-VHDL.

T ::= **bit** | **reg** | **bit_vector**(N_1 **to** N_2) |**bit_vector**(N_1 **downto** N_2) |
reg_vector(N_1 **to** N2) | **reg_vector**(N_1 **downto** N_2) | **clock**

IX ::= N | (N1 **to** N2) | (N1 **downto** N2)

E ::= '0' | '1' | **not** E | E_1 **and** E_2 | E_1 **or** E_2 | E_1 **xor** E_2 | E_1 **nand** E_2 | E_1 **nor** E_2 | I |
| I(IX) | I (E_1,... E_n)

CD ::= E_1 = E_2 | E_1 /= E_2 | CD_1 **and** CD_2 | CD_1 **or** CD_2 | CD_1 **xor** CD_2 |
CD_1 **nand** CD_2 | CD_1 **nor** CD_2 | **not** CD

Un bloc d'instantiation **BI** est destiné à configurer et instancier des composants. La spécification de configuration **CS** définit le lien entre le composant et le couple entité-architecture. Une séquence d'instructions d'instanciation **CI** constitue le corps du bloc.

BI ::= **block** CS ; **begin** CI **end**

CS ::= **for all** :I1 **use** I'(I") | **for** I1:I2 **use** I'(I") | CS ; CS

CI::= I1: I2 **port map** (I_1, ..., I_n) | I1: I2 **port map** ($I_1(IX_1)$, ..., $I_n(IX_n)$) | CI_1; CI_2

Les blocs séquentiels **BS** modélisent les circuits séquentiels dans un style flot de données. Un bloc séquentiel est gardé, la condition de garde est une condition de synchronisation c'est à dire le front montant d'un signal de type clock. Deux types d'instructions gardées **M** sont autorisés: les instructions gardées non conditionnées, et les instructions gardées conditionnées.

BS ::= block (I = '1' **and not** I'**stable**) **begin** M; **end**

M ::= I <= **guarded** E | I(IX) <= **guarded** E |
I <= **guarded** E_1 **when** CD **else** E_2 | I(IX) <= **guarded** E_1 **when** CD **else** E_2|
M_1 ; M_2

Deux types d'instructions d'affectation concurrente **AC** sont autorisés en P-VHDL: Les affectations non-conditionnées et les affectations conditionnées.

AC ::= I <= E | I(IX) <= E |
I <= E_1 **when** CD **else** E_2 | I(IX) <= E_1 **when** CD **else** E_2

Un processus combinatoire **PC** a une liste de sensibilité (I_1, ..., I_n), où $I_{1..n}$ sont les identificateurs des signaux qui composent cette liste. La déclaration de variables **V** suit directement la liste de sensibilité. Une séquence d'instructions séquentielles combinatoires **SC** constitue le corps du processus.

PC ::= **process**(I_1, ..., I_n) V; **begin** SC; **end**
V ::= **variable** I:T | V_1;V_2
Dans un processus synchrone **PS**, la première instruction dans le processus est un **wait** avec une condition de synchronisation. Une séquence d'instructions séquentielles gardées **SG** constitue le corps du processus.

PS ::= **process** V **begin wait on** I **until** I = '1' ; SG; **end**

Quatre types d'instructions séquentielles combinatoires **SC** sont autorisés dans P-VHDL: l'affectation des signaux, l'affectation des variables, l'instruction **if** et l'invocation de procédure.

SC ::= I<= E | **if** CD **then** SC_1 **else** SC_2 **end if** | I(IX)<= E |
if CD_1 **then** SC_1 **elsif** CD_2 **then** SC_2 **else** SC_3 **end if** |
I' := E | I'(E_1,...., E_n) | I'(IX) := E |
SC_1; SC_2

Deux types d'instructions séquentielles gardées sont autorisées dans P-VHDL: l'affectation des signaux et l'instruction **if.**

SG ::= I <= E | I(IX) <= E | **if** CD **then** SG **end if** | SG_1 ; SG_2

Dans les procédures et les fonctions, trois types d'instructions sont autorisés: l'affectation de variable, l'instruction **if** et l'invocation de procédure.

SP ::= I := E | I(IX) := E | **if** CD **then** SP_1 **else** SP_2 **end if** |
if CD_1 **then** SP_1 **elsif** CD_2 **then** SP_2 **else** SP_3 **end if** |
I'(E_1,...., E_n) |
SP_1;SP_2

4.3 Les domaines sémantiques

Trois domaines de base sont utilisés dans la définition de P-VHDL: **Nat, Bool** et **Ide.** **Bool** est le domaine de valeurs booléens {vrai, faux}. **Nat** est le domaine des entiers naturels. **Ide** est le domaine des identificateurs.

n ∈ Nat = {0,1,2, ...}
b ∈ Bool = {vrai, faux}
i ∈ Ide = Identificateur

Un signal en P-VHDL peut être un port d'entrée, un port de sortie, un signal interne, un registre ou une horloge. Le domaine **signal** représente cette classe d'objets.

s ∈ Signal = Input + Output + Internal + Register + Clock

Input est le domaine des ports d'entrées, **Output** est celui des ports de sorties, **Internal** est celui des signaux internes et **Clock** est celui des signaux ayant le type **clock.**

in ∈ Input
o ∈ Output
l ∈ Internal
k ∈ Clock

Le domaine **Register** est l'union de deux domaines primitifs: **IntReg** et **OutReg.**
g ∈ Register = OutReg + IntReg

OutReg est le domaine des ports de sorties de type **reg. IntReg** est le domaine des signaux internes de type **reg.**

r ∈ IntReg
t ∈ OutReg
Les deux domaines de fonction **Current** et **New** associent les valeurs des signaux et des registres avec leurs identificateurs. **Current** est le domaine des fonctions qui associent les valeurs courantes des signaux avec leurs identificateurs.

c ∈ Current = Signal -> Bool

New est le domaine des fonctions qui associent les nouvelles valeurs de registres avec leurs identificateurs.

n ∈ New = Register -> Bool
RegNewCurrent est le domaine des couples de fonctions associés aux registres.

nc ∈ RegNewCurrent =New x Current

SigNewCurrent est l'ensemble des fonctions qui associent à chaque signal soit sa valeur courante et sa nouvelle valeur s'il est un registre soit sa valeur courante seulement s'il n'en est pas.

sc ∈ SigNewCurrent = RegNewCurrent + Current

Le domaine **Variable** représente l'ensemble des variables déclarées dans les processus.

v ∈ Variable

Store est le domaine des fonctions qui associent les valeurs de variables avec leurs identificateurs.

$s \in$ Store = Variable -> Bool

CurrentStore est l'ensemble des fonctions qui associent à chaque variable sa valeur et à chaque signal sa valeur courante.

$cs \in$ CurrentStore = Current + Store

SigStore est l'union du domaine **SigNewCurrent** et du domaine **Store.**

$ss \in$ SigStore = SigNewCurrent + Store

L'ensemble de domaines $\mathbf{Proc_n}$ est l'ensemble des domaines des procédures. Où $Proc_3$, par exemple, représente une procédure qui a trois paramètres.

$p \in Proc_n$ = (Signal + Variable) ^{n} -> CurrentStore -> CurrentStore

L'ensemble de domaines $\mathbf{Fun_n}$ est l'ensemble des domaines des fonctions booléennes . Où Fun_3 représente une fonction qui a trois paramètres.

$f \in Fun_n$ = (Signal + Variable) ^{n} -> CurrentStore -> Bool

L'ensemble de domaines $\mathbf{Comp_n}$ est l'ensemble des domaines des composants. Où $Comp_3$ représente un composant qui a trois paramètres.

$cmp \in Comp_n$ = Signal ^{n} -> SigNewCurrent -> SigNewCurrent

Le domaine **Arch** est le domaine des architecture

ch ∈ Arch = Env x Nat -> Env

Le domaine **Ent** est le domaine des entités

nt ∈ Ent = Env -> Env

Le domaine des fonctions **array** est le domaine des vecteurs. Le premier domaine dans ce domaine produit représente les éléments du vecteur. Le deuxième et le troisième domaines représentent ses deux bornes.

y ∈ array = (Nat -> (Signal + Variable)) x Nat x Nat

Le domaine des valeurs dénotables **Dv** représente les valeurs dénotées par des identificateurs.

$$d \in Dv = Signal + Variable + Bool + Nat + Array + \bigcup_{0}^{\infty} Proc_n + \bigcup_{0}^{\infty} Fun_n + \bigcup_{0}^{\infty} Comp_n + Ent + Arch$$

Env est le domaine de fonctions qui fait correspondre à chaque identificateur la valeur qu'il dénote.
u ∈ Env = Ide -> Dv

4.4. Les fonctions d'évaluation

Les équations des fonctions d'évaluation utilisent les opérations sur les valeurs booléennes et sur les entiers(non, ou, et, xou, nor, nand,+ , - , =). De plus, les abréviations suivantes[Te81] sont utilisées:
1. Pour tester la compatibilité de domaine, il est necessaire de tester le type des éléments dans un domaine d'union, tel que **Signal = Input + Output + Internal + Register + Clock**
On définit des prédicats postfixés '.?input' , '.?output', '.?Internal', '.?Register' et '.?Clock' sur Signal.

Exemple:

$$s?Input = \begin{cases} \text{vrai, si s appartient au domaine Input} \\ \text{faux, autrement,} \end{cases}$$

2. Un opérateur de séléction '. -> ., .' est défini comme suit:

$$e \text{ - > } x_1 \text{ , } x_2 = \begin{cases} x_1, si(e = vrai) \\ x_2, si(e = faux) \\ \perp \text{ , si (e?Bool) = faux} \end{cases}$$

3. Un opérateur d'affectation '.[. ↦ .]'

Exemple:

Pour le domaine Store l'opérateur d'affectation est défini comme suit:

$\mapsto$: Ide -> Bool -> Store -> Store

s[I $\mapsto$ r] est le même domaine que **s** sauf pour **I** où la valeur de s[I $\mapsto$ r] est égale à **r** ; c'est-à-dire, pour tout I',

$$s[I \mapsto r]\ [[I']] = \begin{cases} r, \text{ si } (I = I') \\ s[[I']], \text{ autrement} \end{cases}$$

Dans la suite de ce chapitre, nous définissons les fonctions d'évaluation de P-VHDL. Dans un premier temps, nous nous limitons aux scalaires pour garder la clarté de la définition. Ensuite, deux exemples montrant les traitement de l'indexation et la surcharge des fonctions seront donnés dans la section (4.4.15).

4.4.1 Les Expressions

La fonction d'évaluation **E** fait correspondre à chaque forme d'expression dans le domaine syntaxique **E** une valeur booléenne ou elle rend erreur. Les doubles crochets sont utilisés pour bien séparer les morceaux syntaxiques des notions sémantiques. La fonction d'évaluation pour les expressions est interprétée par rapport à un environnement **u** et un CurrentStore **cs**.

E : Expression -> Env -> CurrentStore -> $Bool_{\perp}$

E [['0']] u cs = faux

E [['1']] u cs = vrai

E [[not E]] u cs = e? Bool -> non(e) , $\perp$
où e = **E** [[E]] u cs

E [[E_1 and E_2]] u cs = e_1? Bool et e_2?Bool -> e_1 et e_2 , $\perp$
où e_i = **E** [[E_i]] u cs

E [[E_1 or E_2]] u cs = e_1?Bool et e_2?Bool -> e_1 ou e_2, $\perp$
où e_i = **E** [[E_i]] u cs

E [[E_1 xor E_2]] u cs = e_1?Bool et e_2?Bool -> e_1 xou e_2 , $\perp$
où e_i = **E** [[E_i]] u cs

E [[E_1 nand E_2]] u cs = e_1?Bool et e_2?Bool -> e_1 nand e_2, $\perp$
où e_i = **E** [[E_i]] u cs

E [[E_1 nor E_2]] u cs = e_1?Bool et e_2?Bool -> e_1 nor e_2 , $\perp$
où e_i = **E** [[E_i]] u cs

E [[I]] u cs = d?Signal -> c(d), d?Variable -> s(d),
d?Bool -> d, $\perp$ où d = u[[I]]

E [[I (E_1,... E_n)]] u cs = d?fun_n et ?e_j Bool -> d(e_1,...,e_n), $\perp$
où d = u[[I]] , e_j = **E**[[E_j]] , j = 1 jusque n

La fonction u[[I]] dans les deux équations précédentes fait correspondre à **I** la valeur qu'il dénote.

4.4.2 Les Conditions

La fonction d'évaluation **CD** fait correspondre à chaque forme de condition dans le domaine syntaxique **CD** une valeur booléenne ou elle rend erreur.

CD : Condition -> Env -> CurrentStore -> $\text{Bool}_{\perp}$

CD [[E_1= E_2]]u cs = e_1 ? bool et e_2? bool -> e_1 = e_2 , $\perp$
où e_1=**E** [[E_1]]u cs et e_2= **E** [[E_2]]u cs

CD [[E_1 /= E_2]] u cs = e_1 ? bool et e_2? bool -> non (e_1 = e_2) , $\perp$
où e_1=**E** [[E_1]] u cs et e_2= **E** [[E_2]] u cs

CD [[not CD]] u cs = cd? Bool -> non(cd) , $\perp$
où e = **CD** [[CD]] u cs

CD [[CD_1 and CD_2]] u cs = cd_1?Bool et cd_2?Bool -> cd_1 et cd_2 , ⊥
où cd_i = **CD** [[CD_i]] u cs

CD [[CD_1 or CD_2]] u cs = cd_1?Bool et cd_2?Bool -> cd_1 ou cd_2, ⊥
où cd_i = **CD** [[CD_i]] u cs

CD [[CD_1 xor CD_2]] u cs = cd_1?Bool et cd_2?Bool -> cd_1 xou cd_2 , ⊥
où cd_i = **E** [[E_i]] u cs

CD [[CD_1 nand CD_2]] u cs = cd_1?Bool et cd_2?Bool -> cd_1 nand cd_2, ⊥
où cd_i = **CD** [[CD_i]] u cs

CD [[not CD]] u cs = cd?Bool -> non cd , ⊥
où cd= **CD** [[CD]] u cs

CD [[CD_1 nor CD_2]] u cs = cd_1?Bool et cd_2?Bool -> cd_1 nor cd_2 , ⊥
où cd_i = **CD** [[CD_i]] u cs

4.4.3 Les Affectations Concurrentes

La fonction d'évaluation **AC** fait correspondre à chaque forme d'instruction d'affectation concurrente dans le domaine syntaxique **AC** une valeur dans le domaine Current ou elle rend erreur. La fonction d'évaluation pour les affectations concurrentes est interprétée par rapport à une environnement **u** et un Current **c**.

AC : AffectationConcurrente -> Env -> Current -> $\text{Current}_{\perp}$
AC [[I <= E]] u c = d?Signal et e?Bool -> c[d↦e]), ⊥
où d = u[[I]] et e = **E** [[E]] u c

La fonction c[d ↦ e] change la valeur courante du signal dénoté par **I** par la valeur booléenne calculée par l'expression **E**.

AC [[I <= E_1 when CD else E_2]] u c =
d?Signal et e_1?Bool et e_2?Bool -> cd -> c[d ↦ e_1] , c[d ↦ e_2], ⊥
où e_1=**E** [[E_1]] u c , e_2= **E** [[E_2]]u c et cd = **CD** [[CD]] u c

4.4.4 Les Affectations Gardées

La fonction d'évaluation **M** fait correspondre à chaque forme d'instruction d'affectation gardée dans le domaine syntaxique **M** une valeur dans le domaine New ou elle rend erreur. La fonction d'évaluation pour les affectations concurrentes sont interprétées par rapport à une environnement **u** et un SigNewCurrent **sc**.

M : InstructionGardée -> Env -> Current -> $New_{\perp}$

M [[I <= guarded E]] u sc =
d?Register et e?Bool-> n[d ↦ e]), ⊥
où d = u[[I]] et e = **E** [[E]] u c

La fonction n[d↦e] affecte la valeur booléenne calculée par l'expression E à la nouvelle valeur du registre I.

M [[I <= guarded E_1 when CD else E_2]]u sc =
d?Register et e_1?Bool et e_2?Bool -> cd -> n[d ↦ e1] ,
n[d ↦ e2] ,
⊥

où d = u[[I]] , e_1= **E** [[E_1]] u c , e_2 = **E** [[E_2]]u c et cd = **CD** [[CD]]u c

M [[M1; M2]] u sc = **M** [[M2]] u sc1 où sc_1= **M**[M1]] u sc

4.4.5 Les blocs séquentiels

Un bloc séquentiel est contrôlé par une expression de garde. La fonction d'évaluation de ce bloc teste si le signal de synchronisation est de type Clock, si oui la séquence des instructions gardées qui compose le bloc est évaluée.

BS : BlocSéquentiel -> Env -> SigNewCurrent -> $RegNewCurrent_{\perp}$

BS [[**block** (I = '1' **and not** I'**stable**) **begin** M; **end**]] $_{sc}$ =
d?Clock -> **M**[[M]]u sc , ⊥
où d = u[[I]]

4.4.6 Les instructions séquentielles gardées

Une instruction séquentielle gardée se trouve dans un processus synchrone. La fonction d'évaluation **SG** est interprétée par rapport à un environnement **u** et un SigNewCurrent **sc**.

SG : InstructionSéquentielleGardée -> Env -> Current -> $New_{\perp}$

SG [[I <= E]]u sc =
d?Register et e? Bool-> n[d ↦ e], ⊥

où d = u[[I]] et e = **E** [E] u c

SG [[**if** CD **then** I <= E **end if**]] u sc =
d?Register et e?Bool et cd?Bool -> cd -> n[d ↦ e], n[d ↦ c(d)], ⊥

où d = u[[I]] , e = **E** [[E_1]] u c et cd = **CD** [[CD]]u c

SG [[SG1 ; SG2]] u sc = **SG**[[SG2]] u sc_1 où sc_1= **SG**[[SG1]] u sc

4.4.7 Les processus Synchrones

Un processus synchrones est une séquence d'instructions gardées, une instruction wait est la première instruction dans le processus. La fonction d'évaluation **PS** teste si le signal I a le type Clock, si oui la séquence d'instructions séquentiels gardées est évaluée.

PS : ProcessusSynchrone -> Env -> SigNewCurrent -> $RegNewCurrent_{\perp}$

PS [[**process** V; **begin wait on** I **until** I = '1' ; SG; **end**]]u sc =
d?Clock -> **SG**[[SG]]u' sc , ⊥

où d = u[[I]] et u' = **D**[[V]]u

4.4.8 Les instructions combinatoires

Une instruction combinatoire se trouve dans un processus combinatoire. La fonction d'évaluation **SC** est interprétée par rapport à un environnement **u** et un CurrentStore **cs**.

SC : InstructionCombinatoire -> Env -> CurrentStore -> CurrentStore$_\perp$

SC [[I <= E]]u cs = d?Signal et e?Bool -> c[d $\mapsto$ e]), $\perp$
où d = u[[I]] et e = **E** [[E]] u cs

SC [[I := E]]u cs = d?Variable et e?Bool -> s[d $\mapsto$ e]), $\perp$
où d = u[[I]] et e = **E** [[E]] u cs

SC [[**if** CD **then** SC_1 **else** SC_2 **end if**]]u cs
= **CD**[[CD]]u cs -> **SC**[[SC_1]]u cs, **SC**[[SC_2]]u cs

SC [[SC_1; SC_2]]u cs = **SC**[[SC_2]]u cs_1 où cs_1 = **SC**[[SC_1]]u cs

SC [[**if** CD_1 **then** SC_1 **elsif** CD_2 **then** SC_2 **else** SC_3 **end if**]]u cs
= **CD**[[CD_1]]u cs -> **SC**[[SC_1]]u cs,
CD[[CD_2]]u cs -> **SC**[[SC_2]]u cs, **SC**[[SC_3]]u cs

SC [[I' (I_1,... I_n)]]u cs = d'?$Proc_n$ -> d'(d_1,...,d_n) , $\perp$
où d' = u[[I']] et d_j = u[[I_j]] j = 1 jusque n

4.4.9 Les processus combinatoires

Un processus combinatoire peut avoir une partie de déclaration. La fonction d'évaluation **PC** est interprétée par rapport à un environnement enrichi par l'exécution de la fonction **D**[[V]].

PC [[**process**(I_1, ..., I_n) V; **begin** SC; **end**]]u c = **SC**[[SC]] u' cs
où u' = **D**[[V]]u

4.4.10 Les instructions dans les procédures et les fonctions

Les instructions dans les procédures et les fonctions sont exécutées par rapport à un environnement **u** et un store **s**.

SP : InstProcFonc -> Env -> Store -> Store$_\perp$

SP [[I := E]] u s = d?Variable et e?Bool -> s[d $\mapsto$ e]), $\perp$
où d = u[[I]] et e = **E** [[E]] u s

SP [[**if** CD **then** SP_1 **else** SP_2 **end if**]]u s
= **CD**[[CD]]u s -> **SP**[[SP_1]]u s, **SP**[[SP_2]]u s

SP [[SP_1; SP_2]] u s= **SP**[[SP_2]]u s_1 où s_1 = **SP**[[SP_1]]u s

SP [[**if** CD_1 **then** SP_1 **elsif** CD_2 **then** SP_2 **else** SP_3 **end if**]]u s
= **CD**[[CD_1]]u s -> **SC**[[SP_1]]u s,
CD[[CD_2]]u s -> **SC**[[SP_2]]u s, **SC**[[SP_3]]u s

4.4.11 Les instructions concurrentes

La fonction d'évaluation C est interprétée par rapport à un environnement **u** et un SigNewCurrent **sc.**

C : InstructionConcurrente -> Env -> SigNewCurrent-> $SigNewCurrent_{\perp}$

C [[AC]]u sc = **AC**[[AC]]u sc

C [[I: BS]]u sc = **BS**[[BS]]u sc

C [[I: PS]]u sc = **PS**[[PS]]u sc

C [[I: PC]]u sc = **PC**[[PC]]u c

C [[I: BI]]u sc = **BI**[[BI]]u c

C [[C1 ; C2]] u sc = **C** [[C2]]u sc_1 où sc_1= **C** [[C1]] u sc

4.4.12 Les déclarations

La fonction d'évaluation **D** est interprétée par rapport à un environnement, et elle rend un environnement.

D : Déclaration -> Env -> Env

D[[**signal** I: T]] u = ?**T**[[T]] Clock -> u [i ↦k] où k ∈ Clock ,
?**T**[[T]] Register -> u [I ↦r] c[r ↦InV(**T**[[T]])],
où r ∈ IntReg,
u [I ↦l] c[l ↦InV(**T**[[T]])]
où l ∈ Internal

La fonction **T** rend le domaine sémantique qui correspond au type T: Clock pour le type **clock**; IntReg, OutReg pour **reg** et **reg_vector** et Internal pour **bit** et **bit_vector**. La fonction **InV** rend la valeur initiale associée avec le type.

V : DéclarationDeVariable -> Env -> Env

D[[V]]u = **V**[[V]]u

V[[**variable** I:T]]u = u [I ↦ v] ou v ∈ Variable

La déclaration du composant **I'** rend un environnement enrichi par l'association de l'identificateur i' au composant cmp.

D[[**component** I' **is port** (I_1: **in** T_1; ...; I_m: **in** T_m; I_{m+1}:**out** T_{m+1}; ....; I_n : **out** T_n);
end component;]]u = u [i' ↦ cmp] où i' =[[I']] et cmp∈ $Comp_n$

La déclaration de la fonction **I'** rend un environnement enrichi par l'association de l'identificateur **i'** à la fonction **f**. La fonction **f** est interprétée par rapport à un environnement composé de celui produit par les paramètres formels et celui produit par les déclaration de variables dans la fonction.

D[[**function** I' (I_1:T_1;..... ; I_n:T_n) **is** V; **begin** SP; end I']]u
= u [i' ↦ f] ou i' =[[I']] et f = **SP**[[SP]](u'+u'') c où u'=u [i_j ↦ v] ,
v ∈ Variable , i_j =[[I_j]] , j = 1 jusque n, u'' = **D**[[V]] et f ∈ Fun_n

La déclaration de la procedure **I'** rend un environnement enrichi par l'association de l'identificateur **I'** à la fonction **p.** La fonction **p** est interprétée par rapport à un environnement composé de celui produit par les paramètres formels et celui produit par les déclarations de variables dans la procédure.

D[[**procedure** I' (I_1: in T_1; ...; I_m: in T_m; Im+1:out Tm+1; ....; In : out Tn) **is**
V **begin** S ; **end** I']]u
= u [i' ↦ p] ou i' =[[I']] et p = **SP**[[SP]](u'+u") c où u'=u [i_j ↦ v] ,
v ∈ Variable et i_j =[[Ij]] j = 1 jusque n , u" = **D**[[V]] et p ∈ $Proc_n$

4.4.13 L'Entité

La fonction d'évaluation **U** ajoute l'entité **nt** à l'environnement **u.** L'entité **nt** est une fonction qui ajoute les ports d'entrée et les ports de sorties dans l'environnement **u.**

U : Entité -> Env -> Env

U[[**entity** I" **port** (I_1: in T_1; ...; I_n: in T_n; I'1:out T'1; ....; I'_m : out T'_m); **end** I";]]
= u [i" ↦ nt] où nt = u [i_i ↦ d_i] c[d_i ↦ InV(d_i)] + u [i'j ↦ d'j] c[dj ↦ InV(dj)]
où
d_i = Input et i_i = **I**[[I_i]], i = 1 jusque n ,
d'j = ?**T**[[T'j]] Bool -> Output , ?**T**[[T'j]] Reg -> OutReg , ⊥, i'j = **I**[[I'j]] , j = 1 jusque m

4.4.14 L'architecture

La fonction d'évaluation **AB** représente la simulation du corps de l'architecture à un cycle d'horloge fixé. Elle est interprétée par rapport à un environnement u qui représente l'ensemble de la déclaration d'entité et de toutes les déclaration contenues dans l'architecture. Le calcul de cet environnement est réalisé par les fonction d'évaluation **D** et **U** et correspond à ce qui est appelé "élaboration" dans le manuel de référence de VHDL. L'effet de **AB** est de fournir les nouvelles valeurs des signaux et des registres dans le domaine SigNewCurrent.

AB : ArchCorp -> Env -> SigNewCurrent-> $SigNewCurrent_{\perp}$

AB[[C]]u sc= cor(u_1) où u_1 = **C**[[O(C)]]u sc

- **O** est la fonction de séquencement de la section (4.4.16)
- **cor**(u_1) rend un environnement identique à u_1 sauf pour les registres où les nouvelles valeurs de ces registres sont affectées à leurs valeurs courantes.

Elle est interprétée par rapport à un environnement qui contient au minimum l'entité à laquelle l'architecture appartient et par rapport au temps de simulation. Cette fonction rend cet environnement enrichi par l'architecture **ch.**

La fonction d'évaluation A représente l'algorithme de simulation d'une architecture, pour un nombre de cycles d'horloge fixé (MaxTime). Elle est interprétée par rapport à l'environnement élaborés par les déclarations d'entité et d'architecture, auquel est rajoutée la variable Now qui représente la date courante de simulation.

A : Arch -> Env x Nat -> Env

A[[**architecture** I_1 **of** I_2 **is** D; **begin** AB; **end** I_1]]u = d_2 ? Ent -> u[$i_1 \mapsto$ ch], $\perp$

où ch = Now = 0
tantque Now $\leq$ MaxTime faire
AB[[AB]] (u'+u") sc
Now = Now + 1
fin tantque

et u" = d_2, u' = **D** [[D]] , d_2 = u[[I_2]] , i_1= [[I_1]]

4.4.15 L'indexation et la surcharge des fonctions

La fonction d'évaluation **IX** fait correspondre à chaque forme d'indice soit un signal, soit une variable, soit un vecteur. La fonction **IX** est interprétée par rapport à un array **y**.

IX: Indice-> array -> (Nat + (Nat x Nat)) -> Dv

IX[[N]] $_y$ = ?d signal ou d ? variable -> d , $\perp$
où d = el(y,na) , na = **N**[[N]] où la fonction **el** rend l'élément na du vecteur y.

IX[[N1 to N2]]$_y$ = ?d_i signal ou d_i ? variable et ($n_{a1} > n_{a2}$) -> y' , $\perp$
où el(y',i) = d_i , d_i = el(y,i), n_{a1}= **N**[[N1]] , n_{a2} = **N**[[N2]] , i = n_{a1} jusqu'à n_{a2}

IX[[N1 downto N2]]$_y$ = ?d_i signal ou d_i ? variable et ($n_{a2} > n_{a1}$) -> y' , $\perp$
où el(y',i) = d_i , d_i = el(y,i), n_{a1}= **N**[[N1]] , n_{a2} = **N**[[N2]] , i = n_{a2} jusqu'à n_{a1}

L'équation suivante donne la définition sémantique pour l'expression I(IX):

$\mathbf{E}[[I(IX)]]_u\ cs$ = a?array -> ic ?signal -> c(ic) , -- si IX sélectionne un signal
ic ? variable -> s(ic) , -- si IX sélectionne une variable
ic ?array -> ic_i? signal -> e, ou e= $c(el(ic_{(i+\ lo(ic))},i))$
-- si IX sélectionne un vecteur et $1 < i < hi(ic_i) - lo(ic_i)$
ic_i? variable -> e', ou e'= $s(el(ic_{(i+\ lo(ic))},i))$
et $1 < i < hi(ic_i) - lo(ic_i)$
⊥
⊥

où les fonction **lo** et **hi** rendent les deux bornes du vecteur.
N.B: Les résultats des expressions ont des bornes standardisées (1 jusqu'à taille).

Les deux équations suivantes donnent les définitions sémantiques complètes pour deux formes d'expression qui ont été traitées pour les scalaires dans le paragraphe (4.4.1). L'équation qui définit l'expression **E1 and E2** montre comment la surcharge de la fonction **and** est traitée dans notre sémantique.

$\mathbf{E}[[I]]_u\ cs$ = d? Signal -> c(d), -- I est un signal
d? Variable -> s(d), -- I est une variable
d? Bool -> d, -- I est une constante
d? array -> d_i ?signal -> d' où $el(d',i-lo(d)) = c(d_i)$, $d_i = el(d,i)$ et $lo(d) \le i \le hi(d)$
d_i ?variable -> d", ⊥ où $el(d",i) = s(d_i)$, $d_i = el(d,i)$ et $lo(d) \le i \le hi(d)$
⊥
où d = u[[I]]

$\mathbf{E}[[E_1$ and $E_2]]_{u\ cs}$ = e_1? Bool et e_2?Bool -> e_1 et e_2 ,
e_1? array et e_2?array et e_{1i} ? bool et e_{2i} ? bool -> y ,
⊥

où $y_1 = \mathbf{E}[[E_1]]$, $y_2 = \mathbf{E}[[E_2]]$, $e_{1i} = el(y_1,i)$, $e_{2i} = el(y_2,i)$, $el(y,i) = e_{1i}$ et e_{2i},
$e_i = \mathbf{E}\ [[E_i]]\ u\ cs$ et $1 \le i \le hi(y_1)$

Nous ajoutons l'équation suivante à la fonction **AC** du paragraphe (4.4.2) pour exprimer la sémantique de l'affectation: I(IX) <= E.

AC [[I(IX) <= E]] u c =

a? array -> ic ? signal e ? bool

-> c[ic ↦ e]), -- si IX sélectionne un élément

-> ic ?array et $ic_{(i+lo(ic))}$? signal e_i ?bool -- si IX sélectionne un vecteur

∀ i dans (1 .. hi(ic) - lo(ic)) c[$ic_{(i+lo(ic))}$ ↦ e_i] , ⊥

où $ic_{(i+lo(ic))}$ = el(ic,i+lo(ic)) et e_i = el(e,i)

, ⊥

où a = u[[I]] et e = **E** [[E]] u c d et ic = **IX**[[IX]] a.les fonction **lo** et **hi** rendent les deux bornes du vecteur.

L'équation suivante est une version complète de la définition sémantique de l'affectation I <= E donnée dans le paragraphe (4.4.2).

AC[[I <= E]]u c =

d?Signal et e?Bool -> c[d ↦ e]) -- I est un Signal

d? array et $ic_{(i+lo(d))}$? signal et e_i?bool et e?array-> -- I est un

∀ i dans (0 .. hi(d) - lo(d)) c[$ic_{(i+lo(d))}$ ↦ e_i] -- vecteur

où $ic_{(i+lo(d))}$ = el(d,i+lo(d)) et e_i = el(d_1,i)

, ⊥

, ⊥

où d = u[[I]] et d_1 = **E** [[E]] u c

4.4.16 La fonction de séquencement O

La fonction de séquencement O a comme entrée la séquence des instructions concurrentes d'une architecture, et elle produit une séquence ordonnée pour cette liste.
On définit la suite des instructions concurrentes dans une architecture par le quintuplet

SUC = <S, C, SignauxLu, SignauxEc, RegistresEc> où

S est un ensemble qui contient tous les signaux internes (y compris les registres) de l'architecture, les ports d'entrée-sorties de l'entité et toutes les constantes(bit et vecteurs de bits) utilisées en partie droite d'affectations.

C est un ensemble qui contient toutes les instructions concurrentes de l'architecture, augmenté par deux instructions C_{in} et C_{out}. Dans la suite, in et out représentent le numéro de ces deux instructions.

SignauxLu est une fonction qui rend le sous-ensemble de S lu par chaque instruction $C_i \in C$. (partie droite des affectations, ports d'entrée effectifs des connexions de composants, liste de sensibilité des processus)

SignauxEc est une fonction qui rend le sous-ensemble de S (excluant les registres) écrit par chaque instruction $C_i \in C$. (partie gauche des affectations, ports de sortie effectifs de connexions de composants)

RegistresEc est une fonction qui rend les registres modifiés par chaque instruction $C_i \in C$. (partie gauche des affectations gardées)

C_{in} est une instruction artificielle pour modéliser le noeud de départ avec:

SignauxEc = les ports d'entrée
RegistresEc = les registres
SignauxLu = $\varnothing$

C_{out} est une instruction pour modéliser le noeud d'arrivé avec:

SignauxEc = $\varnothing$
RegistresEc = $\varnothing$
SignauxLu = les ports de sortie + les registres

On définit le graphe de séquencement **SG** de SUC par le graphe orienté

SG=<Noeud,Arrête> où

Noeud = { Ci | Instruction $C_i \in C$};

Arrête = $\text{Arrête}_{\text{lit-de}} \cup \text{Arrête}_{\text{lit-de-cin}} \cup \text{Arrête}_{\text{lit-de-cout}}$;

$\text{Arrête}_{\text{lit-de}}$ = {<C_i,C_j> où $\text{SignauxLu}_j \cap \text{SignauxEc}_i \neq \varnothing$ et $i \neq in$ et $i \neq out$}
Une arrête de l'ensemble **$\text{Arrête}_{\text{lit-de}}$** est un couple <$C_i$,$C_j$> tel que au moins un signal écrit par C_i est lu par C_j.

$\text{Arrête}_{\text{lit-de-cin}}$ = {<C_{in},C_i> où SignauxLu_i c ($\text{SignauxEc}_{in} \cup \text{RegistresEc}_{in}$) et $i \neq in$}
Une arrête de l'ensemble **$\text{Arrête}_{\text{lit-de-cin}}$** est un couple <$C_{in}$,$C_i$> tel que tous les signaux lus par C_i sont des entrées ou des variables d'état.

Arrête$_{lit\text{-}de\text{-}cout}$ = {<C_i,C_{out}> où $SignauxLu_{out} \cap (SignauxEc_i \cup RegistresEc_i) \neq \varnothing$}
Une arrête de l'ensemble **Arrête$_{lit\text{-}de\text{-}cout}$** est un couple <$C_i$,$C_{out}$> tel que au moins un signal(y compris les registres) écrit par C_i est lu par C_{out}.

La fonction **Construit-Graphe** a comme entrée la suite des instructions de l'architecture et comme sortie le graphe de séquencement.

Construit-Graphe : C -> SG

fonction **Construit-Graphe**(C)
Noeud = { Ci | Instruction $C_i \in$ C, i = 1 jusque n}; -- l'ensemble des instructions.
pour i = 1 à n faire -- la construction de Arrête$_{lit\text{-}de\text{-}cin}$
 si i ≠ j alors
  si $SignauxLu_i$ c ($SignauxEc_{in} \cup RegistresEc_{in}$) alors ajouter <$C_{in}$,$C_i$> à Arrête$_{lit\text{-}de\text{-}cin}$
 fin si
fin pour
pour i = 1 à n faire -- la construction de Arrête$_{lit\text{-}de}$
 pour j = 1 à n faire
  si $SignauxLu_j \cap SignauxEc_i \neq \varnothing$ alors ajouter <C_i,C_j> à Arrête$_{lit\text{-}de}$
 fin pour
fin pour
pour i = 1 à n faire -- la construction de Arrête$_{lit\text{-}de\text{-}cin}$
 si $SignauxLu_{out} \cap (SignauxEc_i \cup RegistresEc_i) \neq \varnothing$ ajouter <C_i,C_{out}> à Arrête$_{lit\text{-}de\text{-}cout}$
fin pour
SG = <Noeud,Arrête> -- le graphe de séquencement
return(SG);
fin **Construit-Graphe**;

Dans la suite, on suppose définies les trois fonctions suivantes:

- **détecter-boucle** : SG -> Bool
  La fonction **détecter-boucle** teste s'il y a des boucles dans le graphe . S'il n'y en a pas elle rend faux, si elle en trouve elle rend vrai.
- **suc**: C x SG -> Noeud
  La fonction **suc** renvoie l'ensemble des instructions C_k qui suivent immédiatement une instruction C_i dans le graphe, c'est à dire telles que <C_i,C_k> est une arrête du graphe.
- **pre:** C x SG -> Noeud

La fonction **pre** renvoie l'ensemble des instructions C_k qui précèdent immédiatement une instruction C_i dans le graphe, c'est à dire telles que $<C_k,C_i>$ est une arrête du graphe.

La fonction de séquencement **O** fait appel aux fonctions construit-graphe et détecter-boucle pour construire le graphe de séquencement et vérifier l'absence de boucles combinatoire. Ensuite, elle ordonne la liste des affectations dans une séquence d'exécution.

O: C -> C

```
fonction O(C)
SG =construit-graphe(C)
si détecter-boucle(SG) alors return (⊥)
Séquence = {Cin} -- Séquence est la séquence ordonnée
noeuds-de-top = Cin
noeud-à-calculer = {Ci | Ci ∈ Noeud, i ≠ in}
tantque noeud-à-calculer ≠ {Cout}faire
      pour tous Cj ∈ noeuds-de-top faire
            S=suc(Cj,SG) -- S est un ensemble de noeuds
            pour tous Ci ∈ S  et Ci ≠ Cout faire
                  M = pre(Ci,SG) -- M est un ensemble de noeuds
                  si M c Séquence alors
                        ajouter(Ci, Séquence)
                        ajouter(Ci, nouv-noeuds-de-top)
                        enlever(Ci,noeud-à-calculer)
            fin pour
      fin pour
      noeuds-de-top = nouv-noeuds-de-top
fait
enlever(Cin,Séquence)
return(Séquence)
fin O;
```

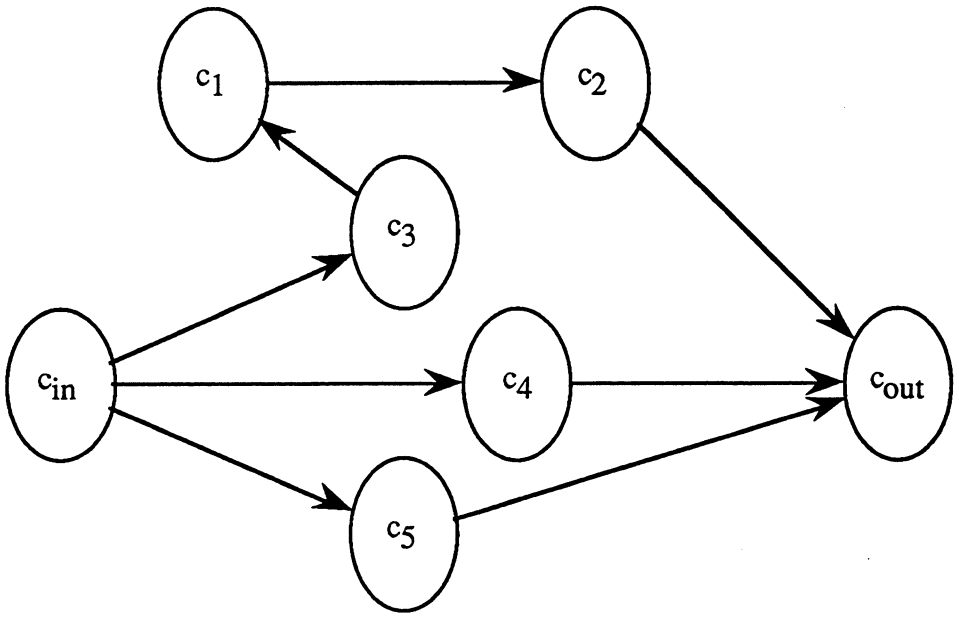
Exemple

Considerons la description P-VHDL suivante:

```
entity xx is
      port (a,b : in bit; clk : in clock; o,s: out bit);
end xx;
```

```
architecture yy of xx is
signal x,y : bit; signal r:reg;
begin
      y <= x and a;                    -- C1
      process                          -- C2
      begin
      wait on clk until clk = '1';
            r <=y;
      end process;
      x <= b or r;                     -- C3
      o <= r;                          -- C4
      s <= a and b;                    -- C5
end;
```

1) Le graphe produit par la fonction **construit-graphe** est le suivant



$SignauxEc(C_{in}) = \{a,b\}$	$SignauxLu(C_{in}) = \varnothing$	$RegistresEc(C_{in}) = \{r\}$
$SignauxEc(C_1) = \{y\}$	$SignauxLu(C_1) = \{x,a\}$	$RegistresEc(C_1) = \varnothing$
$SignauxEc(C_2) = \varnothing$	$SignauxLu(C_2) = \{y,clk\}$	$RegistresEc(C_2) = \{r\}$
$SignauxEc(C_3) = \{x\}$	$SignauxLu(C_3) = \{b,r\}$	$RegistresEc(C_3) = \varnothing$
$SignauxEc(C_4) = \{o\}$	$SignauxLu(C_4) = \{r\}$	$RegistresEc(C_4) = \varnothing$
$SignauxEc(C_5) = \{s\}$	$SignauxLu(C_5) = \{a,b\}$	$RegistresEc(C_5) = \varnothing$
$SignauxEc(C_{out}) = \varnothing$	$SignauxLu(C_{out}) = \{o,s,r\}$	$RegistresEc(C_{out}) = \varnothing$

La séquence produite par la fonction de séquencement **O** : $C = C_5\ C_4\ C_3\ C_1\ C_2$

Chapitre 5

Sémantique Formelle des Primitives Temporelles du Langage VHDL

5.1 Introduction

Le modèle temporel de VHDL est décrit, dans le manuel de référence [Ie88], par une machine de simulation dirigée par événements. Cette description, que l'on peut considérer comme une sémantique opérationnelle, n'est pas formelle.
Cette sémantique définit un simulateur spécifique et décrit comment une primitive temporelle doit être exécutée par ce simulateur, sans décrire réellement la fonctionnalité de la primitive elle même.

Après la présentation de cette sémantique, nous montrons dans ce chapitre pourquoi elle est inadaptée au raisonnement formel sur les primitives qu'elle décrit. Enfin, nous proposons une sémantique, fonctionnelle celle là, et donc mieux adaptée à la preuve.

5.2 Le modèle temporel de VHDL

VHDL utilise un modèle temporel à deux niveaux: le temps physique et le retard unitaire *delta*. Le temps physique est discret avec comme unité la plus petite la femtoseconde, fs en abrégé.
Le retard unitaire est utilisé dans le langage pour sérier les exécutions parallèles des processus. Ces opérations sont nommées, dans le manuel de référence, *les cycles de simulation*. Tous les processus actifs dans le modèle sont exécutés une fois chaque *cycle de simulation* et leur exécution peut initialiser un autre cycle de simulation dans le même instant physique mais un delta plus tard. Les résultats de ces cycles ne sont pas cachés et tous les simulateurs doivent les calculer de la même façon et dans le même ordre.
Un signal peut donc avoir plusieurs valeurs dans le même instant physique, ces différentes valeurs étant séparées par des retards unitaires (deltas). Le rôle de retard unitaire est illustré par la description suivante.

```
B:block
        signal A,B: bit := '1';
        signal Y,Z: bit;
begin
        Z <= transport not Y;
        Y <= transport A and B;
end;
```

A l'instant t = 0 fs + 0 delta, les valeurs des signaux Z et Y sont '0'. Un delta plus tard, les valeurs des signaux Y et Z passent à '1'. L'événement sur Y réactive la première affectation et le signal Z vaut alors '0' à l'instant t =0 fs + 2 delta. Donc, au même instant physique, le signal Z a trois valeurs séparées par deux deltas.

Comme le montre cet exemple, VHDL ne calcule pas les valeurs des signaux en continu, mais seulement lorsqu'un événement se produit. Il prédit les valeurs futures d'un signal en utilisant sa valeur courante et les actions prévues sur elle. Quand le temps avance, la valeur prévue devient une valeur courante. Ce changement de valeur, en d'autres termes cet événement, initie un nouveau cycle de simulation. Les valeurs prévues, nommées en VHDL *transactions*, sont stockées dans les *pilotes*
La *transaction* est un couple, le premier élément de ce couple est le temps programmé pour cette action, le deuxième est la valeur prévue à cet instant.

Le signal pourrait avoir plusieurs *pilotes* s'il est affecté par plusieurs processus. En fait, il n'en a qu'un seul par processus même s'il est affecté plusieurs fois dans celui-ci.

Les notions de pilotes et de transactions sont illustrées par la description VHDL suivante.

```
B:block
        signal A:bit;
begin
        A <= transport '1' after 10 fs;
end;
```

A l'instant t = 0 fs, la valeur courante du signal A est '0' et il est prévu qu'elle vaudra '1' après 10 fs. Donc, son pilote doit avoir deux transactions: la transaction courante (,'0') et celle qui est prévue dans 10 fs (10 fs,'1'), figure (5.1).

	10 fs
'0'	'1'

Figure (5.1) Le pilote du signal **A** à l'instant t = 0 fs

A l'instant t = 10 fs, la transaction (10 fs,'1') devient courante, et le pilote n'aura qu'une seule transaction (,'1'), figure (5.2).

'1'

Figure(5.2) Le pilote du signal **A** a l'instant t = 10 fs

5.2.1 Les deux modèles de transmission en VHDL

Le délai de la transmission s'exprime en VHDL par la présence du mot clé *after* dans l'instruction d'affectation du signal. Le langage supporte deux modèles de transmission: *transport* et inertiel (*inertial*).

Le délai transport est une caractéristique d'un organe électronique à fréquence de coupure infinie, une impulsion même de très courte durée est transmise à travers lui. Par contre, le délai inertiel modélise les circuits logiques où une impulsion de durée plus courte que le temps de commutation n'est pas transmise.

Les sémantiques des deux types de délais sont définies dans [Ie88], par deux algorithmes de remplissage et vidage des pilotes.

Algorithme A: Sémantique du délai transport, [Ie88]

1. Toutes les transactions qui sont projetées à ou après le temps auquel la plus ancienne transaction est projetée, sont effacées du pilote. (N.B ancienne transaction = transaction déjà calculée)

2. Les nouvelles transactions sont ajoutées au pilote dans l'ordre de leurs projections.

Algorithme B: Sémantique du délai inertiel[Ie88]

1. Toutes les nouvelles transactions sont marquées.

2. Une ancienne transaction est marquée si elle précède immédiatement une transaction marquée et si les valeurs des deux transactions sont les mêmes.

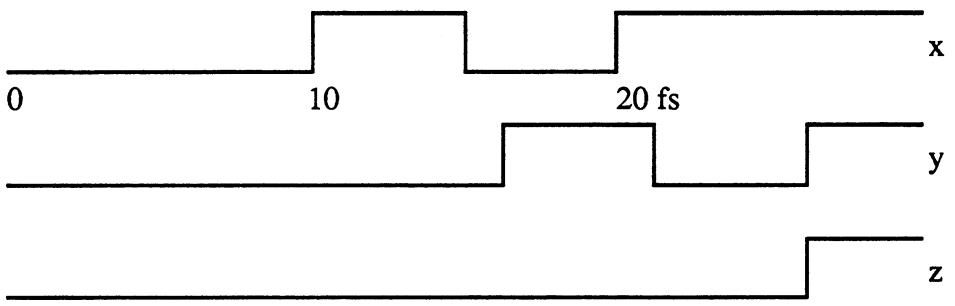
3. La transaction qui détermine la valeur courante du pilote est marquée.

4. Toute transaction qui n'est pas marquée est effacée du pilote.

Pour clarifier le fonctionnement des deux algorithmes, considérons la description VHDL suivante:

```
B:block
        signal X, Y, Z:bit;
begin
        X <= transport '1' after 10 ns, '0' after 15 ns, '1' after 20 ns;
        Y <= transport X after 6 ns;
        Z <= inertial X after 6 ns;
end;
```

Dans cette description, les signaux Y et Z ont la même source mais avec deux types de transmissions différentes: *transport* pour Y et *inertiel* pour Z. Les chronogrammes montrent les trois formes d'onde pour les trois signaux X, Y et Z, figure(5.3).



Figure(5.3) Les chronogrammes de X, Y et Z

Les états des pilotes Z et Y sont donnés figure(5.4). A l'instant t = 0 (ou plus précisément à 0 + 1 delta), les contenus des deux pilotes sont identiques. Chacun a deux transactions: la transaction courante(,'0') et la transaction (6 ns,'0').
A t = 15 ns, la transaction (21 ns,'0') est ajoutée en conservant la transaction (16 ns,'1') dans le pilote Y et en effaçant cette dernière du pilote Z.
Cet effacement a lieu car la transaction (16 ns,'1') ne peut pas être marquée dans le pilote du signal Z en appliquant l'algorithme B. Donc, la valeur du signal Y devient '1' à l'instant t = 16 ns, par contre le signal Z reste '0'.
La transaction (21 ns,'0') subira, à son tour, le même sort dans le même pilote pour la même raison à l'instant t = 20 ns. Mais cet effacement n'aura pas d'incidence sur Z car la valeur de la transaction et la valeur courante du signal sont égales.

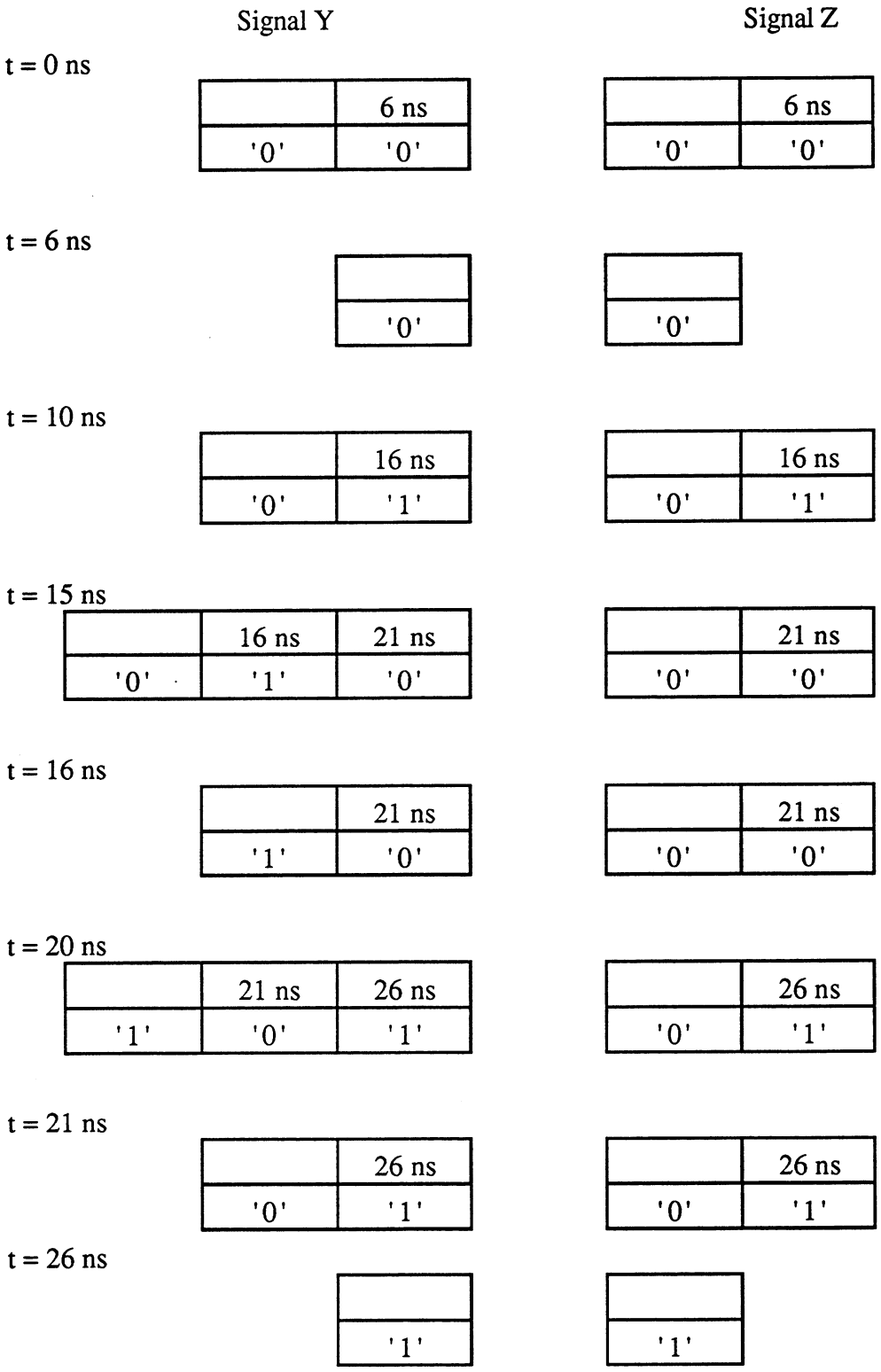


Figure 5.4 Les pilotes des signaux Y, Z.

5.2.2 Les primitives temporelles de VHDL

Nous pouvons distinguer trois catégories de primitives temporelles dans VHDL:

1. La primitive *after* dans les instructions d'affectation de signaux.
2. Les attributs de signaux.
3. La primitive *wait.*

Dans ce qui suit, nous n'effectuons pas une étude complète de ces primitives. Seuls les cas pour lesquels nous définirons une sémantique formelle seront discutés.

5.2.2.1 Les instructions d'affectation de signaux

Le mode d'exécution d'une instruction d'affectation dépend de sa position dans la description. Dans un processus, les instructions s'exécutent en séquence, et ailleurs en parallèle. L'instruction d'affectation concurrente peut être gardée (ou/et) conditionnée.

Le signal a un seul pilote par processus, et les séquences d'affectations contribuent à ce pilote. Par contre, chaque instruction d'affectation concurrente a son pilote et comme un signal peut être affecté plusieurs fois, il peut disposer de plusieurs pilotes arbitrés par une *fonction de résolution.*

La description VHDL suivante [Au89] illustre la sémantique de l'instruction d'affectation concurrente conditionnée. Cette description décrit un tampon qui a un temps de montée de 10 fs et un temps de descente de 14 fs en faisant appel au délai *transport* puis au délai *inertiel.*

```
deux_tampons:block
signal entree, sortie_i, sortie_t:bit;
begin
    tampon_inertiel:block
    begin
        sortie_i <= '1' after 10 fs when entree = '1' else '0' after 14 fs;
    end block;
    transport_tampon:block
    begin
        sortie_t <= transport '1' after 10 fs when entree = '1' else '0' after 14 fs;
    end block;
end block;
```

Maintenant, supposons que le signal entree à cette forme d'onde:

'1' after 0 fs,'0' after 12 fs, '1' after 18 fs,'0' after 20 fs, '1'after 22 fs;

Au temps t = 0, les états des pilotes entree, sortie_i, sortie_t sont les suivants:

	12 fs	18 fs	20 fs	22 fs
'1'	'0'	'1'	'0'	'1'

Le signal entree

	10 fs
'0'	'1'

Le signal sortie_i

	10 fs
'0'	'1'

Le signal sortie_t

Le temps avance de 10 fs car aucune action n'est prévue entre les instants 0 et 10 fs. Les valeurs de sortie_i et sortie_t deviennent '1'

	20 fs
'1'	'1'

L'état du pilote sortie_i et sortie_t à t = 10 fs

Lorsque le temps avance de 2 fs, le signal entree devient '0', la transaction (26 fs,'0') est ajoutée aux pilotes sortie_t et sortie_e, et la transaction (20 fs,'1') est effacée du dernier pilote.

	20 fs	26 fs
'1'	'1'	'0'

	26 fs
'1'	'0'

Les pilotes sortie_t et sortie_i à t = 12 fs

Au temps t = 18 fs, la valeur du signal entree redevient '1', par conséquent la transaction (28 fs,'1') est ajoutée aux pilotes sortie_t et sortie_i. Cette addition efface la transaction (26 fs,'0') du pilote sortie_i.

	20 fs	26 fs	28 fs
'1'	'1'	'0'	'1'

	28 fs
'1'	'1'

Les pilotes sortie_t et sortie_i à t = 18 fs

Le temps avance de 2 fs, la valeur du signal entree change et devient '0'. La transaction (34 fs,'0') est ajoutée aux pilotes sortie_t et sortie_i. La transaction (28 fs,'1') est effacée du pilote sortie_i. La transaction (20 fs,'1') dans le pilote sortie_t devient la transaction courante.

	26 fs	28 fs	34 fs
'1'	'0'	'1'	'0'

	34 fs
'1'	'0'

Les pilotes sortie_t et sortie_i à t = 20 fs

A t = 22 fs, le signal entree revient à '1'. Ce changement ajoute la transaction (32 fs,'1') aux pilotes sortie_t et sortie_i en effaçant la transaction (34,'0') des deux pilotes.

	26 fs	28 fs	32 fs
'1'	'0'	'1'	'1'

	32 fs
'1'	'1'

Les pilotes sortie_t et sortie_i à t = 22 fs

Lorsque le temps avance de 4 fs, la valeur de sortie_t passe à '0', le signal sortie_i reste à '1'. A l'instant t = 28 fs, sortie_t revient à '1'. La simulation s'arrête à l'instant t = 32 fs et à ce moment là, les valeurs des trois signaux sont à '1'.

En conclusion, le tampon avec le retard inertiel n'a pas transmis les impulsions qui ont des durées plus courtes que le temps de transmission. Par contre celui avec le retard transport les a transmises à condition qu'elles ne modifient pas l'ordre des événements.

5.2.2.2 Les Attributs des Signaux

VHDL a un certain nombre de fonctions prédéfinies sur les signaux appelées attributs. Chaque attribut a au minimum un paramètre, le nom du signal, et celui-ci est préfixé. Le deuxième paramètre, s'il existe, a le type TIME.
Deux classes d'attributs peuvent être distinguées:

1. Les attributs qui rendent des signaux.
2. Les attributs qui rendent des valeurs.

Au total, huit attributs de signaux sont prédéfinis en VHDL: S'Delayed(T), S'Stable(T), S'Event, S'Last_Value, S'Last_Event, S_Active, S'Last_Active et S'Transaction. Les trois derniers sont directement liés au mécanisme de simulation; nous n'en définirons donc pas de sémantique formelle et leur fonctionnement sera présenté ultérieurement

5.2.2.2.1 L'attribut Delayed

La sémantique opérationnelle de VHDL ne stocke pas les anciennes valeurs du signal. Par contre, elle permet de les recalculer par l'intermédiaire de l'*attribut* **delayed**. La présence de celui-ci dans une description implique la création d'un signal et d'une instruction d'affectation de type transport avec un temps de transmission égal au retard. L'instruction d'affectation a comme cible le signal créé et comme but le signal attribué. Le signal créé est calculé comme tout autre signal. Ce calcul peut être illustré en considérant la description VHDL suivante:

```
B:block
        signal a,b:bit;
begin
        a <= transport '1' after 10 ns, '0' after 15 ns, '1' after 20 ns;
        b <= transport a'delayed(4 ns);
end;
```

Au moment de la compilation, un signal, noté ci-dessous a_delayed_4ns est créé pour l'attribut a'delayed(4ns). Ce signal est la cible de l'affectation suivante:

```
        a_delayed_4ns <= transport a after 4ns;
```

et l'ensemble du bloc **B** est équivalent au bloc suivant:

```
b_sans_attribut:block
        signal a,b,a_delayed_4ns:bit;
begin
        a <= transport '1' after 10 ns, '0' after 15 ns, '1' after 20 ns;
        a_delayed_4ns <= transport a after 4 ns;
        b <= transport a_delayed_4ns;
end;
```

Comme le montre cet exemple, les anciennes valeurs du signal ne sont pas stockées. Eventuellement, l'attribut delayed calculera une partie de son historique.

5.2.2.2.2 L'attribut Stable

La stabilité d'un signal est calculable selon un principe similaire, en utilisant l'attribut **stable.** Cet attribut crée un signal pendant la compilation, mais à la différence de celui créé par l'attribut **delayed**, il n'est pas traité comme un signal ordinaire. Il appartient à une classe

spéciale nommée les *signaux implicites*. Cette classe contient aussi les signaux S'Quiet et S'Transaction.

La valeur courante et le pilote du signal implicite S'Stable(T) sont modifiés, si et seulement si une des deux conditions suivantes est vraie, [Ie88]:

1. Un événement a lieu sur S pendant le cycle de simulation.
2. Le pilote S'Stable(T) est actif.

Dans le premier cas, la valeur courante de S'Stable(T) devient False et la transaction (True, T) est affectée au pilote S'Stable(T). Dans le second, la valeur prévue dans le pilote devient courante. La valeur initiale de S'Stable(T) est True.

Considérons la forme d'onde suivante:

a <= transport '1' after 10 ns, '0' after 15 ns, '1' after 22 ns;

L'état du pilote a'stable(6 ns) à l'instant t = 0 ns est:

	6 ns
True	True

Au temps t = 6 ns, le pilote est actif. La valeur prévue devient courante.

True

Au temps t = 10 ns, un événement se produit sur le signal A. Le pilote A'Stable(6 ns) est modifié sa valeur devient False.

	16 ns
False	True

Au temps t = 15 ns, le signal A passe à '0'. La valeur courante de A'stable(6 ns) devient False et la transaction (True,21 ns) est affectée à son pilote.

	21 ns
False	True

Au temps t = 21 ns, le pilote A'stable est actif. La valeur courante de A devient True.

True

Lorsque le temps avance de 1 ns, le signal A change de valeur. Donc, le signal implicite A'stable(6 ns) devient False et son pilote est affecté par la transaction (True,28 ns).

	28 ns
False	True

Ici encore une fois, la sémantique opérationnelle de VHDL ne regarde pas le passé. Elle prévoit d'avance la stabilité du signal. Cette prévision peut être modifiée dans le futur en fonction des valeurs réelles des signaux.

5.2.2.2.3 L'attribut Event

Event appartient à la deuxième classe des attributs des signaux, celle qui rend une Valeur. L'attribut S'Event rend une valeur booléenne. Il vaut True si un événement s'est produit sur S pendant le dernier cycle de simulation, sinon il retourne False.

5.2.2.2.4 L'attribut Last_Value

Le résultat rendu par l'attribut S'Last_Value, [Ie88], est la valeur précédente du signal S avant son dernier changement. En plus de cette définition plutôt fonctionnelle, le manuel de référence [Ie88] donne la sémantique axiomatique suivante:
"Un signal S, S'Last_value = S'delayed(T) où T ≥ 0ns est la plus petite valeur de temps telle que S'stable(T) = False. Si un tel T n'existe pas, alors S'Last_Value(T) = S".
L'équivalence entre les deux sémantiques n'est pas prouvée dans ce manuel.

5.2.2.2.5 L'attribut Last_Event

L'attribut Last_Event est défini, lui aussi, dans [Ie88] par deux types de sémantiques, l'une fonctionnelle "L'attribut S'Last_event rend la durée qui s'est écoulée depuis le dernier événement sur S", l'autre axiomatique "Pour un signal S, S'Last_event retourne la plus grande valeur de T pour laquelle S'delayed(T)'stable = True, si une telle valeur de T n'existe pas alors il rend 0ns".
Pour éclaircir ces deux définitions, considérons la forme d'onde suivante:

Y <= transport '1' after 10 ns, '0' after 15 ns, '1' after 20 ns;

A l'instant t = 20 ns, la valeur de l'attribut Y'Last_Event = 0 ns. Une nanoseconde plus tôt cette valeur a été 4 ns car le dernier événement s'est produit à l'instant t = 15 ns.

5.2.2.3 L'Instruction WAIT

L'instruction **Wait** suspend l'exécution du processus. Cette suspension intervient pour une période donnée de temps ou dans l'attente d'un événement sur un ou plusieurs signaux. La reprise de l'exécution est toujours conditionnée même si la clause de condition n'est pas apparente et dans ce cas la condition TRUE est supposée. La primitive Wait joue un rôle primordial dans la sémantique opérationnelle de VHDL. Elle représente en effet le seul moyen de faire avancer le temps, soit implicitement en demandant au simulateur d'attendre un événement sur un signal, alors le temps avance jusqu'à ce qu'il se produise, soit explicitement en l'avançant à un instant donné. La description suivante illustre ce rôle.

```
b:block
        signal X:bit;
begin
        process
        begin
                X<= transport '1' after 10 ns;
        end process
end block;
```

La valeur du signal X dans cette description devrait changer après 10 ns, pourtant ce changement n'aura pas lieu. Le processus continuera à programmer cet événement, i.e. stocker la transaction ('1',10ns) dans le pilote, mais cette transaction prévue ne sera jamais activée car le temps n'avance pas. L'instruction Wait va le forcer à avancer. Les deux types d'utilisation, explicite et implicite, sont décrites dans les exemples suivants:

```
b1:block
        signal X:bit;
begin
        process
        begin
                X <= transport '1' after 10 ns;
                wait for 1 ns;
        end process
end block;
```

```
b2:block
        signal x:bit;
begin
        process
        begin
                X <= transport '1' after 10 ns;
                wait on X;
        end process
end block;
```

Dans la première description, l'instruction "wait for 1 ns" force le simulateur à exécuter dix cycles de simulation avant que la valeur de X soit '1' . Par contre, dans la deuxième description ce changement se produit au début du deuxième cycle de simulation.

5.3 Le modèle temporel de VHDL et le raisonnement Formel

Le raisonnement formel sur le fonctionnement temporel d'un circuit décrit en VHDL nécessite une sémantique formelle de ce langage alors que la sémantique de celui-ci est informelle. Une solution envisageable pourrait être de définir formellement l'algorithme de simulation présenté dans le manuel de référence. Cette définition, par ailleurs trés importante pour les réalisateurs de simulateurs, ne résoudra pas le problème de la preuve pour les raisons suivantes:

1. Le retard unitaire *delta* n'a aucune signification fonctionnelle, car la valeur réelle d'un signal à un moment donné, est celle du dernier delta. Les valeurs du signal aux autres deltas, c'est à dire les valeurs intermédiaires produites et utilisées par l'algorithme de calcul, n'ont qu'un rôle opérationnel.

2. Les valeurs des signaux ne sont pas calculées continuellement, par conséquent le signal ne peut pas être traité comme une fonction du temps.

3. Certaines primitives temporelles (les attributs transaction, active, last_active et quiet) sont fortement liées au mécanisme de simulation et leur fonctionnement ne peut être défini que dans le cadre de ce mécanisme.

4. La sémantique des deux modes de transmission transport et inertiel, est donnée par deux algorithmes spécifiques de remplissage et vidage des pilotes. Par contre aucune définition de leur signification n'est proposée.

Il faudra donc définir un autre type de sémantique pour les primitives temporelles de VHDL avec pour contraintes:

1. La sémantique doit être à la fois formelle et fonctionnelle.

2. Toutes les notions et les primitives liées seulement au mécanisme de simulation seront ignorées. Ainsi le retard unitaire delta, les transactions et les cycles de simulation, voire même quelques attributs de signaux, ne sont pas à considérer.

3. La sémantique doit calculer les valeurs des signaux en permanence et stocker leur historique.

4. Elle doit être compositionnelle. La sémantique d'une instruction composée de plusieurs primitives doit pouvoir s'exprimer en utilisant les sémantiques de ces primitives.

Dans le paragraphe suivant, nous proposons une sémantique formelle pour les primitives temporelles du VHDL qui satisfait ces exigences.

5.4 Sémantique Fonctionnelle des Primitives Temporelles du VHDL

Une description VHDL est un ensemble de processus qui s'exécutent en parallèle. Les processus communiquent entre eux par l'intermédiaire de signaux et peuvent être considérés comme un ensemble de fonctions, le nombre d'éléments de cet ensemble étant égal au nombre de signaux affectés dans chaque processus. Ces fonctions ont comme paramètres la liste de sensibilité du processus. La valeur d'un signal ne peut pas être changée par deux processus en même temps. Ceci bien qu'un signal puisse être affecté dans plusieurs processus, car dans ce cas, l'utilisation d'une fonction de résolution est obligatoire pour arbitrer les deux processus.

Nous adoptons la sémantique de Kahn[Ka74] pour formaliser ce modèle de calcul parallèle. Dans cette sémantique, un programme parallèle est un ensemble de stations de calcul connectées par des lignes de communication. Un observateur est placé sur chaque ligne de communication pour témoigner de son trafic. Ce témoin verra une séquence de valeurs; le type de ces valeurs est celui des données qui peuvent transiter par cette ligne. Cette séquence est appelée l'*historique* de la ligne. La station de calcul a sa mémoire propre représentée par une fonction des historiques de ses lignes d'entrée et de ses lignes de sortie. Dans notre formalisme, chaque processus VHDL est considéré comme une machine de calcul, et chaque signal est considéré comme une ligne de communication.

A l'opposé de la sémantique opérationnelle de VHDL, la référence temporelle dans notre sémantique va du présent vers le passé [Pa86]. En conséquence, le calcul des valeurs des signaux est définitif, car toutes les informations nécessaires sont disponibles au moment du calcul. De plus, notre modèle de temps a un seul niveau, le temps physique, par conséquent nous imposons que toutes les affectations dans un source VHDL supposé prouvé par cette sémantique, aient un retard physique non nul.

5.4.1 La Sémantique des Signaux

L'historique d'un signal est une séquence finie de valeurs; toutes ces valeurs ont le même type qui est le type de valeurs du signal. La distance temporelle entre deux éléments de la séquence est 1 fs.

Le domaine des signaux **Signal** est défini par:

Signal = Integer* + Boolean* + ...

Le domaine des valeurs **Value** des signaux est défini par:

Value = Integer + Boolean + ...

et le domaine de temps **Time** est défini par:

Time = Natural

La longueur d'une séquence X ***len***(X) est égale au temps écoulé depuis le début de la simulation + 1. La tête de la séquence ***hd*** (X) est la valeur courante du signal. La valeur initiale du signal ***initial-value(X)*** est le premier élément de la séquence, géographiquement l'élément le plus à droite. Elle est calculée par la fonction el(len(X) - 1, X), où ***el*** (n,X) rend le nième élement de la séquence X.

Ainsi, l'historique du signal x de la figure(5.5) est répresenté par la séquence : x = (1 1 1 0 1 0 0).

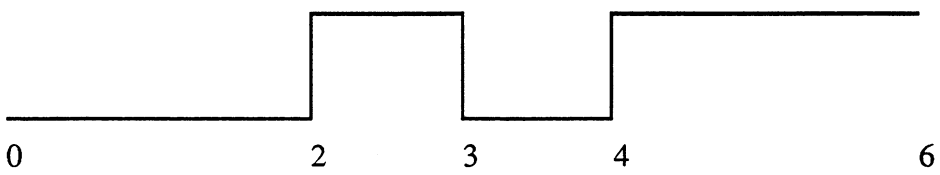


Figure (5.5) Chronogramme du signal x

La valeur courante est hd(x) = 1 et la valeur initiale est el(len(x) - 1, x) = el (7-1, x) =0.

L'avantage de mettre la valeur courante à la tête de séquence est de faciliter la modélisation des instructions d'affectation. La nouvelle séquence représentant le signal cible sera calculée simplement en ajoutant la valeur courante du signal source (la tête de séquence) au signal cible retardé d'une unité de temps.

5.4.2 La sémantique des Attributs

Nous définissons dans ce paragraphe une sémantique fonctionnelle pour les attributs: Delayed, Stable, Event, Last_Value et Last_Event. Cette sémantique traduit les définitions fonctionnelles de ces attributs. L'équivalence entre cette sémantique et les définitions axiomatiques pour les même attributs fera l'objet du chapitre suivant. Toutes les fonctions sémantiques de ce paragraphe sont monotones et préservent la longueur des séquences.

5.4.2.1 L'attribut delayed

S'delayed(T) est un signal S1 équivalent au signal S, mais retardé de T unités de temps. Dans notre modèle, S1 est obtenu en supprimant les T plus récentes valeurs de S et en ajoutant la séquence restante à une séquence composée de T éléments identiques à la valeur initiale de S. La fonction ***delayed*** donne la sémantique de cet attribut.
Le domaine sémantique:

delayed: Signal x Time -> Signal

La fonction sémantique:

```
delayed = λs.λt. si len(s) = 0 alors nil
                 sinon
                     si len(s) ≤ t
                       alors initial-value(s) . delayed(tl(s),t)
                       sinon el(t,s) . delayed(tl(s),t)
                       fsi
                 fsi
```

où tl(s) rend le reste de la séquence après l'enlèvement du premier élément et '.' ajoute un élément en tête de séquence.

5.4.2.2 L'attribut Stable

S'stable(T) est un signal booléen positionné à True, lorsque la valeur de S n'a pas évolué pendant les dernières T unités de temps; sinon sa valeur courante est False.
La fonction ***Stable*** donne la sémantique de cet attribut.

Le domaine sémantique:

Stable: Signal x Time -> Signal

La fonction sémantique:

Stable = λs.λt . si len(s) = 0 alors nil
sinon Sta(s,t) . Stable(tl(s),t)
fsi

La fonction auxiliaire *Sta* rend la valeur booléenne True lorsque le signal S est stable pendant T unités de temps, autrement elle retourne False.

Le domaine sémantique:

Sta: Signal x Time -> Boolean

La fonction sémantique sta:

Sta = λs.λt. si t = 0 alors True
sinon
si len(s) = 1 alors True
sinon Sta(tl(s), t -1) et (hd(s) = hd(tl(s)))
fsi
fsi

5.4.2.3 L'attribut Event

L'attribut Event d'un signal S rend True si la valeur de S vient de changer, sinon il rend False. La fonction ***Event*** donne la sémantique de cet attribut.
Le domaine sémantique:

Event: Signal -> Boolean

La fonction sémantique:

Event(s) = λs. not(hd(s) = hd(Delayed(s,1))

5.4.2.4 L'attribut Last_value

L'attribut Last_Value d'un signal S retourne la valeur de S juste avant le dernier événement sur lui.La fonction **last_value** donne la sémantique de cet attribut.

Le domaine sémantique:

Last_Value: Signal -> Value

La fonction sémantique:

```
Last_Value = λ.s.  si len(s) = 0 alors nil
                   sinon
                        si len(s) = 1 alors hd(s)
                        sinon
                             si hd(s) = hd(tl(s)) alors Last_Value(tl(s))
                             sinon hd(tl(s))
                             fsi
                        fsi
                   fsi
```

5.4.2.5 L'attribut Last_Event

L'attribut S'Last_event indique le temps écoulé depuis le dernier événement sur S. La fonction **Last_Event** donne la sémantique de cet attribut.

Le domaine sémantique:

Last_Event: Signal -> Time

La fonction sémantique:

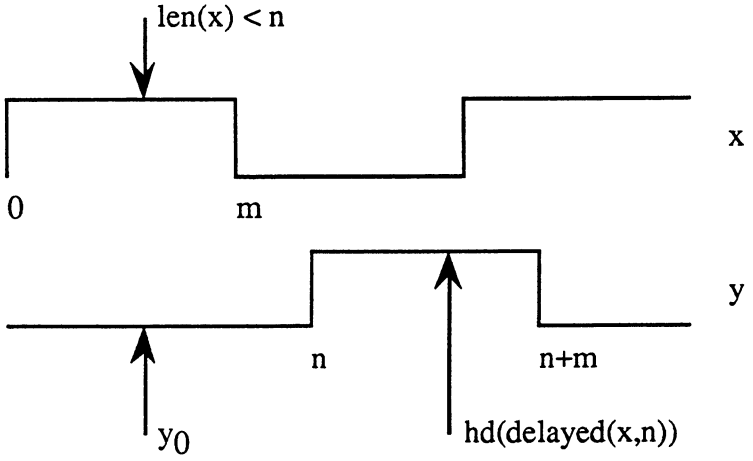
```
Last_Event = λx.si len(x) = 0 alors 0
             sinon
               si len(x) = 1 alors 0
               sinon
                  si hd(x) ≠ hd(tl(x)) alors 0
                  sinon Last_Event(tl(x)) + 1
                  fsi
               fsi
             fsi
```

5.4.3 La sémantique des instructions d'affectation concurrente

Dans cette section, nous donnons une sémantique formelle pour quatre types d'instruction d'affectation concurrente: L'instruction d'affectation avec le retard transport, l'instruction d'affectation avec le retard inertiel, l'instruction d'affectation conditionnée avec le retard transport et l'instruction d'affectation conditionnée avec le retard inertiel.

5.4.3.1 L'instruction d'affectation avec le retard transport

L'instruction d'affectation avec un retard transport:**y <= transport x after n fs** affecte le signal x retardé n fs au signal y figure (5.6).



Figure(5.6): L'affectation avec le retard transport

La fonction **transport** modélise l'instruction d'affectation avec le retard transport:
Le domaine sémantique:

transport:Signal x Time x Value -> Signal

La fonction sémantique:

```
transport = λ (x n y0) .
                si len(x) = 0 alors nil
                sinon
                    si len(x) ≤ n alors y0.transport(tl(x), n, y0)
                    sinon  hd(delayed(x,n)).transport(tl(x),n,y0)
                    fsi
                fsi
```

5.4.3.2 L'instruction d'affectation avec le retard inertiel

L'instruction d'affectation avec un retard inertiel: **y <= x after n fs** affecte le signal x retardé n fs au signal y si le changement dans x persiste pour une période supérieur ou égale à n fs, si non le signal y garde son ancienne valeur.
La fonction **inertial** modélise l'instruction d'affectation avec le retard inertial:

Le domaine sémantique:

inertial: Signal x Time x Value -> Signal

La fonction sémantique:

inertial $=\lambda$ **(x n y0) .**

```
Si len (x) = 0 alors nil
sinon
    si len (x) ≤ n alors  y0.inertial(tl(x),n,y0)
    sinon
        si last_event(x) ≥ n
            alors  hd(delayed(x,n)).inertial(tl(x),n,y0)
        sinon
            si last_event(delayed(x,last_event (x))) ≥ n
                alors  hd(delayed(x,n)).inertial(tl(x),n,y0)
            sinon  hd(inertial(tl(x),n,y0)).inertial(tl(x),n,y0)
            fsi
        fsi
    fsi
fsi
```

Exemple:

Supposons que X a le chronogramme indiqué dans le figure(5.7) et que Y est affecté par l'instruction y <= x after 2 fs: En appliquant la fonction **inertial** sur X et en supposant que la valeur initiale de Y est égale à '0', le signal y aura le chronogramme donné dans le figure(5.7).

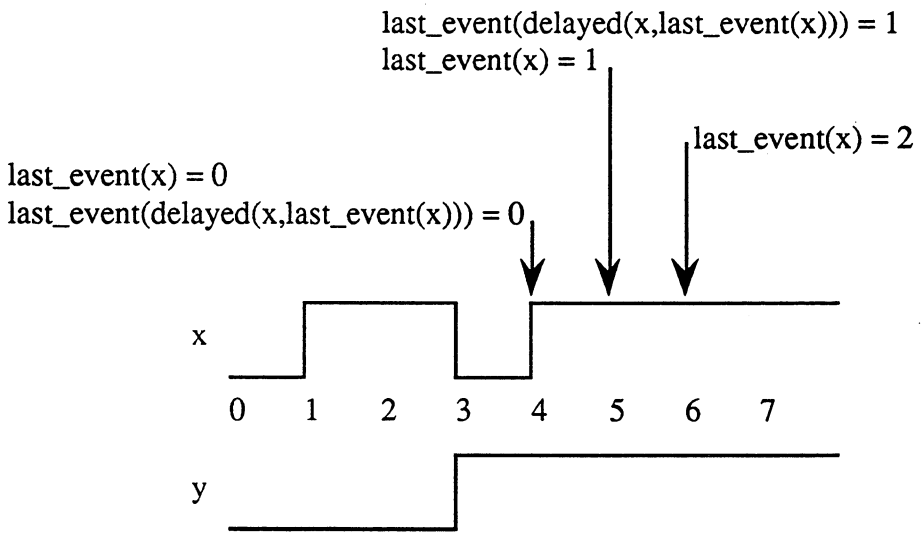


Figure (5.7) L'affectation avec le retard inertiel

5.4.3.3 L'instruction d'affectation conditionnée avec le retard transport

Le cible d'instruction d'affectation conditionnée: **z <= transport x after n fs when c else y after m fs** où m > n, est affecté soit par x ou par y selon le condition c.
Le signal c doit être stable quand il est faux pendant m-n fs pour permettre au signal z d'être affecté par le signal Y.

La fonction **transcond** modélise l'instruction d'affectation conditionnée avec le retard transport:

Le domaine sémantique:

transcond: Signal x Signal x Signal x Time x Time x Value -> Signal

La fonction sémantique:

```
transcond = λ (x y c n m z0) .
    si len(x) = 0 alors nil
    sinon
        si len(x) ≤ n alors z0.transcond(tl(x),tl(y),tl(c), n, m, z0)
        sinon
            si hd(delayed(c,n))
                alors hd(delayed(x,n)).transcond(tl(x),tl(y),tl(c),n,m,z0)
            sinon
                si (last_event(delayed(c,n)) ≥ m-n)
                    et not (hd(delayed(c,m)))
                alors
                    hd(delayed(y,m)).transcond(tl(x),tl(y),tl(c),n,m,z0)
                sinon hd(transcond(tl(x),tl(y),tl(c),n,m,z0)).
                        transcond(tl(x),tl(y),tl(c),n,m,z0)
                fsi
            fsi
        fsi
    fsi
```

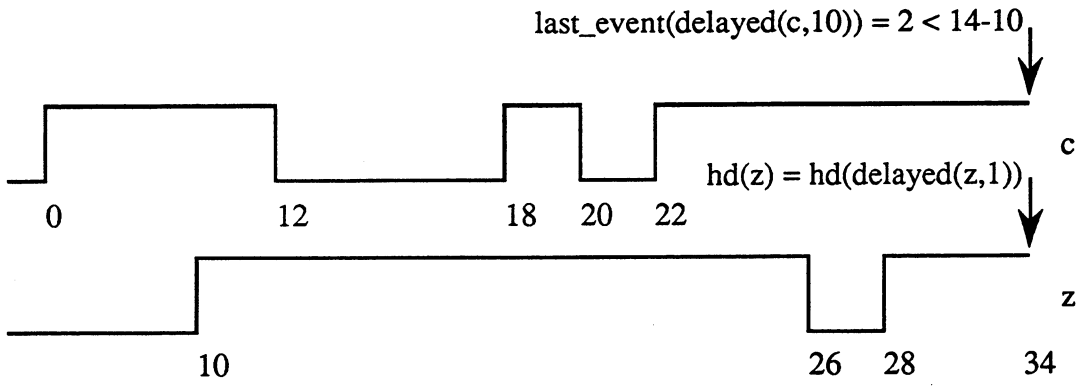
Exemple

Considérons la description VHDL suivante: les signaux x et y sont de type bit et leurs valeurs sont fixées à '1' et '0' respectivement. Le signal c lui aussi a le type bit et il reçoit cette forme d'onde:

'1' **after** 0 fs,'0' **after** 12 fs, '1' **after** 18 fs,'0' **after** 20 fs, '1' **after** 22 fs

Le changement au signal c à l'instant 20 fs n'aura pas d'effet sur le signal z car la durée de sa stablité est inférieure à 4 fs figure(5.8).

```
b:block
    signal x,y,c:bit;
begin
    x <= '1';
    y <= '0';
    c <= '1' after 0 fs,'0' after 12 fs, '1' after 18 fs,'0' after 20 fs, '1' after 22 fs;
    z <= transport x after 10 fs when c= '1' else y after 14 fs;
end;
```



Figure(5.8) L'affectation conditionnée avec un retard transport

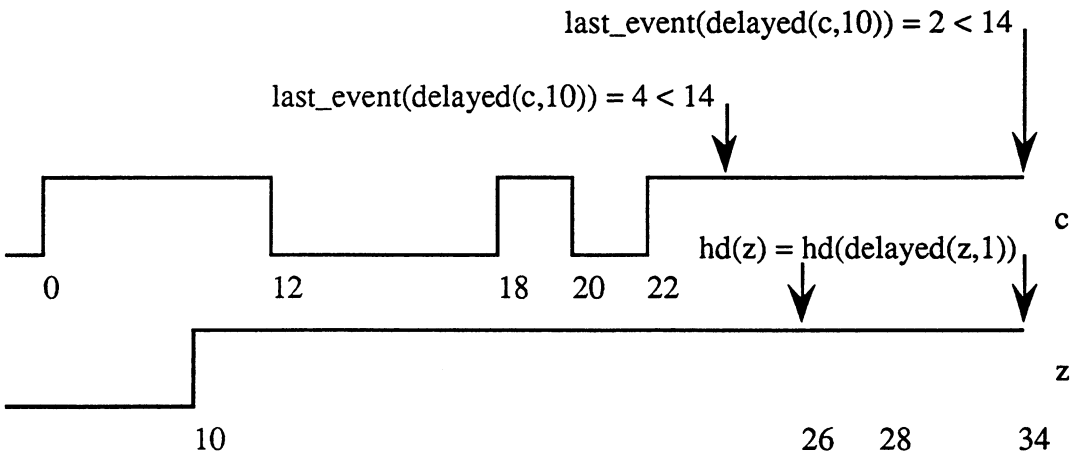
5.4.3.4 L'instruction d'affectation conditionnée avec le retard inertiel

La cible d'instruction d'affectation conditionnée: **z <= x after n fs when c else y after m fs** où m > n, est affectée soit par x ou par y selon la condition c. Le signal c doit être stable quand il est faux pendant m fs pour permettre au signal z d'être affecté par le signal Y à condition que le signal y lui-même soit stable pendant m fs. Le signal z reçoit x si x et c sont stable pendant n fs et la valeur du signal c est égale à '1'.

Exemple

Considérons la description VHDL suivante: le signal x et y sont de type bit et leurs valeurs sont fixées à '1' et '0' respectivement. Le signal c lui aussi a le type bit et il reçoit cette forme d'onde: '1' **after** 0 fs,'0' **after** 12 fs, '1' **after** 18 fs,'0' **after** 20 fs, '1' **after** 22 fs
Le changement au signal c aux instants 12 et 20 fs n'auront pas d'effet sur le signal z car la durée de sa stablité dans les deux cas est inférieure à 14 fs figure(5.9).

```
b:block
      signal x,y,c:bit;
begin
      x <= '1'; y <= '0';
      c <= '1' after 0 fs,'0' after 12 fs, '1' after 18 fs,'0' after 20 fs, '1'after 22 fs;
      z <=  x after 10 fs when c= '1' else y after 14 fs;
end;
```



Figure(5.9) L'affectation conditionnée avec un retard inertiel

La fonction **inercond** modélise l'instruction d'affectation conditionnée avec le retard inertiel:

Le domaine sémantique:

inercond: Signal x Signal x Signal x Time x Time x Value -> Signal

La fonction sémantique:

```
inercond = λ (x y c n m z0) .
  si len(x) = 0 alors nil
  sinon
     si len (x) ≤ n alors  z0.inercond(tl(x),tl(y),tl(c),n,m,z0)
     sinon
          si last_event(delayed(c,m)) > n et hd(delayed(c,n))
          alors si last_event (x) ≥ n alors
                    hd(delayed(x,n)).inercond(tl(x),tl(y),tl(c),n,m,z0)
               sinon
                    si last_event(delayed(x,last_event(x)) ≥ n alors
                         hd(delayed(x,n)).inercond(tl(x),tl(y),tl(c),n,m,z0)
                    sinon
                         hd(inercond(tl(x),tl(y),tl(c),n,m,z0)).
                         inercond(tl(x),tl(y),tl(c),n,m,z0)
                    fsi
               fsi
          sinon
               si last_event(delayed(c,n)) > m et not (hd(delayed(c,m)))
               alors si last_event (y) ≥ m alors
                    hd(delayed(y,m)).inercond(tl(x),tl(y),tl(c),n,m,z0)
                    sinon
                         si last_event(delayed(y,last_event(y)) ≥ m alors
                              hd(delayed(y,m)).inercond(tl(x),tl(y),tl(c),n,m,z0)
                         sinon
                              hd(inercond(tl(x),tl(y),tl(c),n,m,z0)).
                               inercond(tl(x),tl(y),tl(c),n,m,z0)
                         fsi
                    fsi
               fsi
          fsi
     fsi
  fsi
```

Chapitre 6

La Validation du Modèle Formel et l'Application de ce Modèle à la Vérification Temporelle

Dans ce chapitre, en premier lieu, nous validons partiellement les définitions fonctionnelles des primitives temporelles présentées dans le chapitre précédent. Ensuite, nous montrons comment cette sémantique peut établir une base pour la construction d'un système de vérification temporelle. Enfin, nous discutons la limitation de notre modèle en montrant comment cette limitation est, en partie, en accord avec le domaine d'application visé.

6.1. La validation du modèle formel

Le manuel de référence fournit des relations entre les différents attributs de signaux afin de mieux comprendre leurs définitions. Nous considérons ces relations comme une sémantique axiomatique de ces primitives. Et nous prouvons que notre sémantique fonctionnelle satisfait cette définition axiomatique. Autrement dit, nous considérons ces relations comme des lemmes, et nous les prouvons dans notre modèle formel. Dans ce but, nous modélisons notre sémantique dans la logique de Boyer-Moore[BM88] et nous formalisons les relations de notre modèle comme des prédicats de premier ordre. Puis, nous prouvons ces prédicats à l'aide du démonstrateur automatique de Boyer et Moore.

6.1.1 La Logique de Boyer et Moore

La logique de Boyer et Moore est une extension du premier ordre du calcul propositionnel, [BM88]. L'utilisateur de cette logique peut ajouter de nouveaux axiomes avec la garantie que le modèle peut s'étendre pour s'adapter à cette extension. Deux principes d'extension existent dans la logique:

1) Le "*principe de shell*" pour introduire de nouveaux types de données.
2) Le "*principe définitionnel*" pour introduire de nouvelles fonctions.

Un **shell** est défini par:

- Un constructeur,
- Un objet de base,
- Un reconnaisseur,
- Un ou plusieurs accesseurs avec éventuellement des restrictions de types.

La définition d'un shell ajoute une liste d'axiomes dans la logique. Ces axiomes sont obtenus par l'instanciation d'un ensemble de schèmas pré-définis dans la logique. D'autres axiomes peuvent être ajoutés en utilisant le principe définitionnel, c'est à dire par la définition d'un ensemble de fonctions, la plupart d'entre elles étant récursives.

Exemple: Le shell des nombres naturels[BM88]

```
(add-shell add1 zero numberp
    ((sub1 (one-of numberp) zero)))
```

où le constructeur est **add1**, l'objet de base est **zéro**, le reconnaisseur est **numberp** et l'accesseur est **sub1**. L'invocation du shell **add1** ajoute une liste d'axiomes dans la logique. Quelques éléments de cette liste sont:

```
1 - (numberp (add1 x1))  ; x1+1 est un nombre naturel
2 - (numberp (zero))     ; zéro est un nombre naturel
3 - (not (equal (add1 x1)(zero))) ; x1+1 ne peut pas être égal à zéro
```

La fonction **plus** est définie récursivement sur le shell par:

```
(defn plus(x y)                    ; la fonction plus a deux paramètres x et y
    (if  (zerop x)                 ; le test d'arrêt x égale 0
        (if (numberp y) y 0)       ; la valeur de la fonction plus quand x = 0 est égale
                                   ; à y si y est un nombre, sinon 0
        (add1 (plus (sub1 x) y))))  ; x + y = ((x-1) + y )+ 1
```

La fonction **zerop** est définie par:

```
(defn zerop(x)(or (equal x 0)(not (numberp x))))   ; renvoie vrai si x = 0 ou
                                                   ; si x n'est pas un nombre
```

Un démonstrateur automatique est associé avec la logique. La fonction **prove-lemma** fournie les lemmes à prouver au démonstrateur.

Exemple: Si x est un nombre naturel non nul alors x-1 est un nombre naturel.

```
(prove-lemma num-sub1 (rewrite)  ; le lemme a pour nom num-sub1 et
                                 ; pourra être utilisé comme règle de réécriture
    (implies ( and (numberp x) (not(zerop x)))(numberp (sub1 x)))
```

Le démonstrateur utilise les six techniques de preuve suivantes:

1. La simplification: une combinaison de procédures de décision (pour le calcul propositionnel, l'égalité, et l'arithmétique linéaire) et de réécritures.
2. L'elimination de destructeur: le remplacement des termes difficiles à manipuler par des termes facile à manipuler.
3. La fertilisation croisée: l'exploitation des hypothèses d'égalité puis leur élimination.
4. La généralisation: l'adoption d'un but plus général obtenu en substituant des termes par de nouvelles variables.
5. L'elimination des hypothèses non pertinentes.
6. L'induction.

6.1.2. La modélisation de la sémantique dans Boyer-Moore

Nous définissons un shell avec un constructeur **up** pour représenter les séquences finies qui modélisent les historiques des signaux. L'objet de base est **es** et le reconnaisseur est **signal.** Deux accesseurs sont définis: **hd** qui rend le premier élément dans une séquence et **tl** qui rend le reste de la séquence.

Définition 1: Le shell up

```
(add-shell up es signal
      (( hd (none-of) es)
            (tl (one-of signal) es )))))
```

Nous définissons les deux fonctions suivantes **len** et **nth** sur le shell .

Définition 2: La fonction len

```
(defn len(x)(if (or (not(signal x))(equal x (es)))
            (zero)
            (add1 (len (tl x)))))
```

Définition 3: La fonction nth

```
(defn nth (n x)(if (zerop n)
            (hd x)
            (nth (sub1 n)(tl x))))
```

Ces deux fonctions, les accesseurs **hd** et **tl**, le reconnaissseur **signal** et le constructeur **up** formalisent les six fonctions de base utilisées dans le chapitre précédent. Les définition 4 à 11

sont l'écriture dans la logique de Boyer et Moore des fonctions correspondant aux attributs et aux affectations retardées.

Définition 4: La fonction initial-value

```
(defn initial-value(x)(nth (sub1 (len x)) x))
```

Définiton 5: La fonction delayed

```
(defn delayed(x n)
      (if (zerop (len x))
         es
         (if (leq (len x) n)
            (up (initial-value x) (delayed (tl x) n))
            (up (nth n x)(delayed (tl x) n)))))
```

Définition 6: La fonction sta

```
(defn sta(s time)
      (if (zerop time)
         t
         (if (equal (len s) 1)
            t
            (and (sta (tl s) (sub1 time))
                 (equal (hd s) (hd (tl s)))))))
```

Définition 7: La fonction stable

```
(defn stable(s time)
      (if (zerop (len s))
         es
         (up (sta s time)(stable (tl s) time)))))
```

Définition 8: La fonction last_value

```
(defn last_value(x)
      (if (zerop (len x))
        es
        (if (equal (len x) 1)
           (hd x)
           (if (equal (hd x)(hd(tl x)))(last_value(tl x))
                                       (hd(tl x))))))
```

Définition 9: La fonction last_event

```
(defn last_event (x)(if (zerop (len x))
            zero
            (if (equal (len x) 1)
                zero
                (if (not (equal(hd x)(hd (tl x))))
                    zero
                    (add1(last_event(tl x)))))
```

Définition 10: La fonction transport

```
(defn transport(x n y0)
    (if (zerop (len x))
        es
        (if  (leq (len x) n)
            (up y0 (transport (tl x) n y0))
            (up (hd(delayed x n))(transport (tl x) n y0)))))
```

Définition 11: La fonction inertiel

```
(defn inertial(x n y0)
    (if (zerop (len x))
        es
        (if (leq (len x) n)
            (up y0 (inertial (tl x) n y0))
            (if (leq n (last_event x))
                (up (hd(delayed x n))(inerial (tl x) n y0))
                (if (leq n (last_event(delayed x (last_event x)))
                    (up (hd(delayed x n))(inertial (tl x) n y0))
                    (up (hd (inertial(tl x) n y0))(inertial(tl x) n y0))))))
```

6.1.3 Une validation partielle du modèle formel

Pour valider notre sémantique fonctionnelle, nous devons prouver qu'elle est fonctionnellement équivalente à la sémantique donnée dans le manuel de référence.
La sémantique de VHDL du manuel de référence est essentiellement opérationnelle et, pour un petite, partie axiomatique. Les définitions de ces deux parties ne sont pas formelles. Nous ne formalisons pas la partie opérationnelle de la sémantique; nous nous contentons de valider notre définition fonctionnelle par rapport à la partie axiomatique que nous formalisons.

La sémantique axiomatique est définie dans le manuel de référence par des relations entre les attributs de signaux, et les instructions d'affectation des signaux. Dans cette section, nous modélisons six de ces relations dans notre modèle formel. Puis nous les exprimons comme des lemmes à prouver par la logique de Boyer et Moore. La preuve de ces six lemmes est effectuée sur un sparc-2 par le démonstrateur automatique de cette logique. Nous considérons cette preuve comme une validation partielle de notre sémantique.

Lemme 1:

Si T_S est la plus petite valeur de temps telle que S'Stable(T_S) est faux, alors pour tout t_i où $0 \leq t_i < T_S$, S'Delayed(t_i) = S.

Ce lemme est exprimé dans notre modèle fonctionnel par le prédicat suivant:

$$\forall\ t_i,\ t_i > 0\ \&\ t_i \leq T_S\ \&\ \mathbf{signal(s)}\ \&\ \mathbf{hd(stable(s,}T_S\mathbf{)}\ \&$$
$$\mathbf{not(hd(stable(s,}T_S\mathbf{+1)))}\ \Rightarrow\ \mathbf{hd(delayed(s,}t_i\mathbf{)) = hd(s)}$$

Ce prédicat est exprimé dans la logique de Boyer-Moore par la fonction suivante:

```
(prove-lemma lemme-1()
    (implies    (and    (lessp 0 ti)
                        (signal s)
                        (leq ti ts)
                        (hd (stable s ts))
                        (not (hd (stable s (add1 ts)))))
                (equal (hd (delayed s ti)) (hd s))))
```

Lemme 2:

Si S'Stable(Ts) est faux, alors par définition ß t_i où $0 \leq t_i \leq$ Ts, S'Delayed(t_i) ≠ S.

La version du démonstrateur de Boyer et Moore utilisée ne supporte pas le quantificateur existentiel. Pour éliminer ce quantificateur, nous transformons ce lemme en:

Si pour tous t_i tel que $0 \leq t_i \leq$ Ts, S'Delayed(t_i) = S alors S'Stable(Ts) est vrai.

Qui peut être exprimé par le prédicat de premier ordre:

$$\forall\ t_i,\ t_i > 0\ \&\ t_i \leq t_S\ \&\ \mathbf{signal(s)}\ \&\ \mathbf{hd(delayed(s,}t_i\mathbf{)) = hd(s)}$$
$$\Rightarrow\ \mathbf{hd(stable(s,}t_S\mathbf{))}$$

Ce prédicat s'exprime dans la logique de Boyer-Moore par la fonction suivante:

```
(prove-lemma lemma-2()
    (implies    (and    (lessp 0 ti)
                        (signal s)
                        (leq ti ts)
                        (equal (hd(delayed s ti )) (hd s)))
                (hd(stable s ts)))
```

Lemme 3:

Pour un signal S, S'last_value = S'delayed(T_i) où $T_i \geq 0$ ns est la plus petite valeur telle que S'Stable(T_i) est faux.

Ce lemme peut être exprimé en utilisant notre modèle fonctionnel avec le prédicat suivant:

$\forall$ t_i, t_i > 0 & signal(s) & hd(stable(s,ti)) & not hd(stable(s,t_i+1))
=> last_value(s) = hd(delayed(s,t_i+1))

Ce prédicat est exprimé par la fonction suivante, où **(induct (sta s ti))** force le démonstrateur à commencer par faire l'induction sur la fonction **sta.**

```
(prove-lemma lemme-3()
    (implies  (and  (lessp 0 ti)
                    (signal s)
                    (hd (stable s ti))
                    (not (hd (stable s (add1 ti) ))))
              (equal (last_value s)(hd (delayed s (add1 ti)))))
    ((induct (sta s ti))))
```

Lemme 4:

Pour un signal S, S'last_value = S'delayed(T_i) où $T_i \geq 0$ ns est la plus petite valeur de temps telle que S'Stable(T_i) est faux, si un tel T_i n'existe pas, alors S'last_value est égal à S.
Ce lemme est exprimé dans notre sémantique par les deux prédicats suivants:

4.1) **$\forall$ t_i, t_i > 0 & signal(s) & hd(stable(s,t_i)) & not(hd(stable s (t_i+1)))**
=> last_value(s) = hd(delayed(s,(t_i+1)))

4.2) $\forall$ t_i, t_i > 0 & signal(s) & true-sig(stable(s,t_i)) => hd(s) = last_value(s)

Nous formalisons le lemme(4.1) par le prédicat suivant:

```
(prove-lemma lemma-4-1 ()
      (implies      (and    (lessp 0 ti)
                            (signal s)
                            (hd (stable s  ti))
                            (not (hd(stable s (add1 ti) ))))
                    (equal (last_value s)(hd (delayed s (add1 ti)))))
      ((induct (sta s t))))
```

Pour modéliser le prédicat(4.2) dans la logique de Boyer et Moore, nous définissons la fonction **true-sig**:

```
(defn true-sig(s)(if (zerop (len s))
      t
      (and (hd s)(true-sig (tl s)))))
```

La fonction **true-sig** rend "vrai" si son paramètre est un signal qui a toujours la valeur "vrai", sinon elle rend "faux". En utilisant cette fonction auxiliaire, le lemme 4.2 est modélisé dans la logique de Boyer-Moore par la fonctions suivante:

```
(prove-lemma lemme-4-2()
      (implies      (and    (lessp 0 ti)
                            (not(zerop (len s)))
                            (true-sig (stable s ti))
                            (signal s))
                    (equal (hd s)(last_value s))))
```

Lemme 5:

Pour un signal S, S'last_event retourne la plus grande valeur Ti de type Time pour laquelle S'delayed(T_i)'stable peut retourner "vrai".
Nous formalisons ce lemme par le prédicat suivant:

signal(s) & last_event(s) = t_i =>
hd(delayed(stable(s,1),t_i)) & not(hd(delayed(stable(s,1),t_i+1)))

Dans la logique de Boyer et Moore, où (induct (nth ti s)) force le démonstrateur à commencer l'induction sur la fonction nth, ce prédicat est modélisé par la fonction:

```
(prove-lemma lemme-5()
    (implies    (and    (signal s)
                        (lessp 0 ti)
                        (equal (last_event s) ti))
                        (and (hd (delayed (stable s 1) ti))
                (not (hd (delayed (stable s 1) (add1 ti))))))
    ((induct (nth ti s))))
```

Lemme 6:

Soit R le même sous type que S, soit $T_i \geq 0ns$ et soit P un processus de la forme:

```
P: process(S)
    begin
        R<= transport S after Ti;
    end process;
```

Supposons que les valeurs initiales de R et de S sont égales, alors l'attribut delayed est défini tel que S'delayed(T_i) = R pour tout T_i.

Nous exprimons ce lemme par le prédicat suivant:

$\forall$ti,ti> 0 & signal(s)& r0 = initial-value(s) => delayed(s,t_i) = transport(s,t_i,r0)

Selon la logique de Boyer-Moore, où (induct(len s)) force le démonstrateur à commencer l'induction sur la fonction len, ce prédicat s'exprime par la fonction.

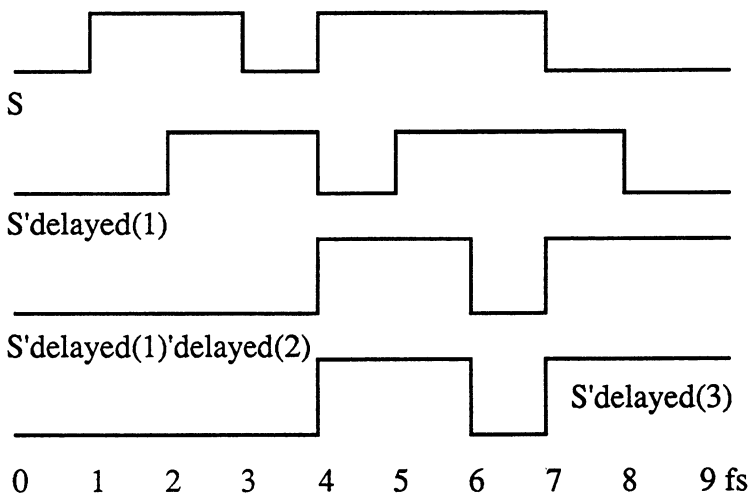
```
(prove-lemma lemma-6()
    (implies    (and  (signal s)
                      (equal r0 (initial-value s))
                      (lessp 0 ti))
                (equal (delayed s ti) (transport s ti r0)))
    ((induct (len s))))
```

6.2 La sémantique et la vérification temporelle

La plupart des outils de vérification temporelle sont basés sur des méthodes analytiques [HSC82]. Ces outils calculent, entre autre, le chemin critique de retard, la marge de temps et les violations contre le temps de mise en route et le temps de maintien.
Une autre vision de la vérification temporelle proposée dans [EM90] est basée sur la transformation formelle de descriptions HDL. Dans cette approche le problème de vérification temporelle consiste en en la résolution d'un grand nombre de sous-problèmes en utilisant un petit ensemble de lemmes. Eveking a appliqué cette approche sur son langage SMAX en construisant un système de vérification temporelle appelé VERTICO. Pour appliquer cette approche à VHDL, ces lemmes de transformation doivent être prouvés conformes à la sémantique de ce langage. Nous proposons d'utiliser notre modèle formel pour démontrer une telle conformité. Et, pour montrer l'applicabilité de notre choix, nous avons choisi deux lemmes proposés dans [EM90] et un autre lemme proposé dans [BJ83] que nous prouvons dans notre sémantique.

Lemme 7:

Soit S un signal, soit m,n ≥ 0, alors S'delayed(n)'delayed(m) = S'delayed(m+n) [EM90].



Figure(6.1): S'delayed(n)'delayed(m) = S'delayed(m+n)

Le lemme peut être exprimé dans notre sémantique par le prédicat suivant:

m ≥ 0 & n ≥0 & signal(s) => delayed(delayed(s,m),n) = delayed(s,n+m)

Nous vérifions ce lemme avec Boyer et Moore en deux étapes. La première est de prouver le lemme:

m ≥ 0 signal(s) => delayed(delayed(s,m),1) = delayed(s,m+1)

Ce lemme est fournit au démonstrateur par la fonction suivante, où l'option "rewrite" stocke le lemme comme une règle de réécriture qui sera utilisée dans la deuxième étapes de la preuve.

```
(prove-lemma lemme-7-1(rewrite)
    (implies    (and    (numberp m)
                        (signal s)
                        (not (zerop m)))
                (equal (delayed s (add1 m) )(delayed (delayed s m) 1)))
    ((induct (len s))))
```

Ensuite, nous prouvons le lemme original en forçant le démonstrateur à faire l'induction sur la fonction **plus.**

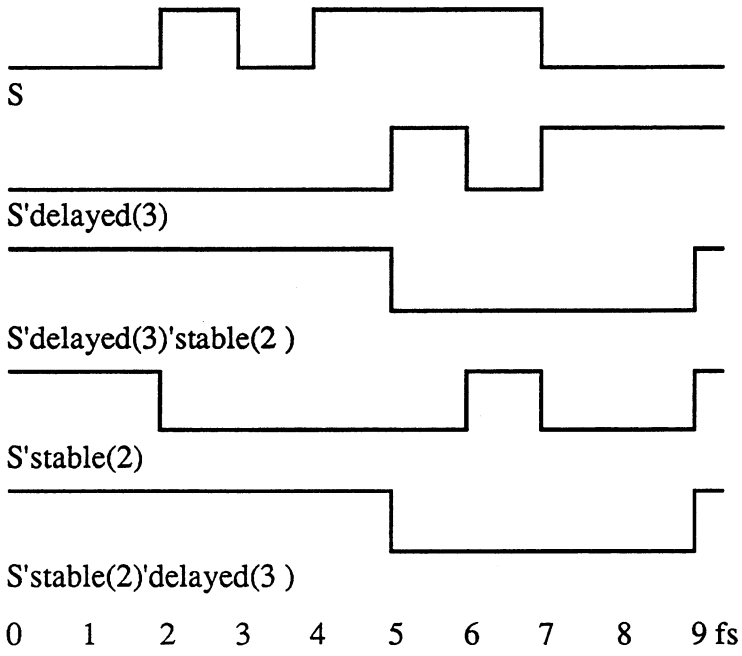
```
(prove-lemma lemme-7()
    (implies    (and   (lessp 0 m)
                       (signal s)
                       (lessp 0 n))
                (equal (delayed s (plus n m))(delayed (delayed s m) n )))
    ((induct (plus n m))))
```

Lemme 8:

Soit S un signal, soit m,n ≥ 0, alors S'delayed(n)'stable(m) = S'stable(m)'delayed(n) [EM90].

Le lemme peut être exprimé dans notre sémantique par le prédicat suivant:

n ≥ 0 & m ≥ 0 & signal(s) =>
hd(delayed(stable(s,m),n)) = hd(stable(delayed(s,n),m))



Figure(6.2): S'delayed(n)'stable(m) = S'stable(m)'delayed(n)

Nous formalisons ce lemme dans Boyer et Moore par la fonction suivante:

```
(prove-lemma lemme-8()
      (implies (and  (not(zerop n))
                     (not(zerop(len s)))
                     (numberp n)
                     (not(zerop m))
                     (signal s))
            (equal   (hd(delayed(stable s m) n ))
                     (hd (stable (delayed s n) m))))))
```

Lemme 9:

Soit R un signal du même sous-type que S, soit n ≥ 0 ns, et soit P un processus de la forme:

```
P: process(S)
begin
      R<= S after n fs;
end process;
```

Et supposons que le signal S est stable pendant m fs où n ≤ m, alors S'delayed(n) = R, [BJ83].

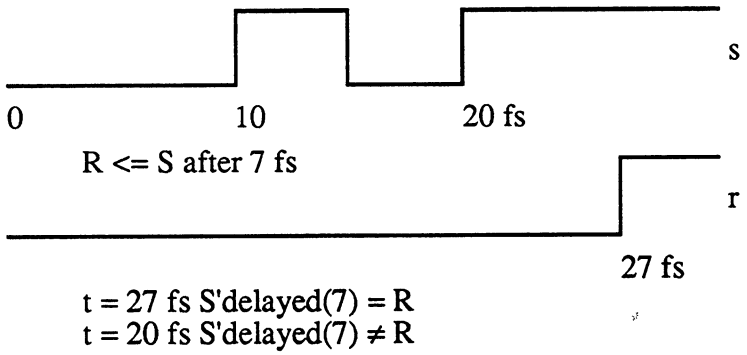


Figure (6.3): Le retard inertiel

Le lemme peut s'exprimé dans notre sémantique par le prédicat suivant:

signal(s) & n > 0 & m > 0 & hd(stable(s,m)) & n ≤ m
=> hd(delayed(s,n)) = hd(inertial(s,n,r0))

La fonction suivante modélise ce lemme dans la logique de Boyer et Moore:

```
(prove-lemma lemme-9()
      (implies (and  (lessp 0 n)(lessp 0 m)
                     (leq n m)(signal s)
                     (hd (stable s m)))
               (equal (hd(delayed s n))
                     (hd(inertiel s n r0)))))
```

6.3 La limitation de notre modèle formel

Dans la définition sémantique que nous avons donnée au chapitre 5, deux types de restriction sont imposés:

a) des restrictions syntaxiques.

b) des restrictions sémantiques.

En fait, les restrictions syntaxiques impliquent que notre définition fonctionnelle ne couvre pas la totalité des primitives temporelles de VHDL.

Nous n'avons pas traité:

1. la séquence d'instructions d'affectation dans un processus.
2. les signaux avec plusieurs pilotes et les fonctions de résolution.

Par contre, les restrictions sémantiques signifient que notre définition exclut une partie de la fonctionnalité supposée être associé à une primitive syntaxique!

Nous discutons maintenant de ces deux types de restrictions:

a) Les restrictions sémantiques

La sémantique que nous avons proposée traite seulement le temps physique. Elle ignore le retard delta et la sémantique du cycle de simulation.
Le retard delta intervient dans l'exécution d'une description VHDL en l'absence de retard physique. Nous avons imposé que toutes les instructions d'affectation dans un source VHDL, supposé prouvé par cette sémantique, aient un retard physique, i.e.une primitive **after** avec un temps supérieur à zéro. Ce choix nous permet d'une part, d'ignorer la sémantique de cycle de simulation car un seul cycle de simulation est suffisant pour calculer les valeurs des signaux à chaque instant de temps et, d'autre part, l'utilisation des affectations avec un retard supérieur à zéro est en accord avec l'hypothèse fondamentale de l'application de notre sémantique à la vérification temporelle, où toutes les portes logiques utilisées dans un modèle ont des retards de propagation.

b) Les restrictions syntaxiques

Les restriction syntaxiques que nous avons imposées, autrement dit le sous-ensemble des primitives temporelles traitées, sont elles aussi en accord avec le domaine de vérification temporelle, car une description de type comportemental est souvent utilisée pour décrire un niveau d'abstraction assez élevé en ignorant le comportement temporel du circuit.
Par contre, l'utilisation des fonctions de résolution est toujours associé avec la modélisation des bus de données et des registres, où les conditions d'utilisation des bus ou de chargements de registres sont souvent liées au temps.
Dans le cadre de notre sémantique, le traitement des affectations multiples d'un signal dans le même processus ne devrait pas poser de problème, car la composition des fonctions modélisant les différentes affectations peut être faite statiquement. Et, pour les affectations multiples d'un signal dans plusieurs processus, la fonction sémantique dénotant chaque affectation ne représentera plus le signal lui même, mais un de ses pilotes, une fonction globale englobera toutes ces fonctions arbitrées par la fonction de résolution.

Conclusion

Dans cette thèse, nous avons contribué à l'intégration des outils de vérification formelle d'un environnement de CAO basé sur VHDL.

Notre contribution a porté sur deux points:

1. La définition de P-VHDL.
2. La proposition d'une sémantique formelle pour les primitives temporelles de VHDL.

La définition de P-VHDL

Nous avons défini P-VHDL, un sous-ensemble de VHDL destiné à décrire les circuits combinatoires et les circuits séquentiels synchrones. Dans ce sous-ensemble, seules les primitives qui peuvent décrire une réalisation au niveau transfert de registres et une spécification algorithmique non temporisé sont retenues. Un sous ensemble avec ces restrictions a bien évidement une sémantique beaucoup plus simple que celle de VHDL complet. En fait, le retard unitaire delta a été remplacé par une simple fonction de séquencement. Et grâce à notre style de description, l'échelle de temps devient la période de l'horloge. Ainsi, la machine d'état fini a pu être utilisée comme modèle formel pour le sous-ensemble.

L'équivalence entre ce modèle et la sémantique de VHDL a été montrée sous les restrictions syntaxiques et sémantiques imposées par P-VHDL. Cette sémantique est à la base de l'écriture d'un compilateur en vue de la preuve qui constitue le coeur de l'environnement de vérification formelle PREVAIL. A partir d'une description d'un couple entité-architecture, nous produisons le format d'entrée d'un démonstrateur(preuve de tautologie, ou preuve d'équivalence de machine d'état fini, ou simulation symbolique), en fonction du type de circuit (combinatoire ou séquentiel) , et de l'objectif de vérification envisagé (équivalence permanente entre deux circuits, résultat obtenu au bout d'un temps fixé).

Ensuite nous avons défini une sémantique dénotationnelle pour P-VHDL. Pour cela, nous avons proposé trois domaines différents pour les trois objets porteurs des valeurs: les variables, les signaux et les registres. En nous basant sur ces trois domaines, nous avons formalisé toutes les primitives de P-VHDL, ainsi qu'un algorithme efficace de simulation pour ce langage. Cette formalisation serait transposable à d'autres langages de description de circuits synchronisées par une seule horloge.

La sémantique formelle des primitives temporelles de VHDL

Nous avons proposé un modèle formel pour les primitives temporelles de VHDL. Dans un premier temps, nous avons étudié la définition opérationnelle informelle de ces primitives, en montrant pourquoi elle est inadaptée au raisonnement formel. Ensuite, nous avons proposé une sémantique fonctionnelle pour ces primitives, basée sur la notion de séquence souvent utilisée dans les langages flot de données.

Nous avons traité un des deux niveaux du modèle temporel de VHDL, celui de temps physique. Le retard unitaire delta, autrement dit la sémantique du cycle de simulation, n'a pas été considéré. Deux raisons ont justifié notre choix d'ignorer ce niveau du modèle temporel. Le premier est le caractère opérationnel de celui-ci, car il décrit une machine de simulation spécifique. De plus, le domaine d'application visé par notre modèle formel, à savoir la vérification temporelle, exige un style de description dans lequel toutes les affectations ont un retard physique; par conséquent la sémantique du retard unitaire delta n'est pas utilisée pour calculer les valeurs des signaux.

Puis nous avons validé notre définition sémantique de ces primitives par rapport aux quelques définitions axiomatiques du manuel de référence de VHDL, en considérant ces axiomes comme des lemmes à prouver dans notre modèle formel. Les preuves ont été réalisée à l'aide du démonstrateur de théorème de Boyer et Moore.

Finalement, nous avons montré comment notre sémantique peut établir une base pour construire un système de vérification temporelle.

Perspectives

Deux axes de travaux ultérieurs sont envisagés :

1. La définition d'une sémantique formelle pour VHDL complet. Celle-ci nécessite de résoudre les problèmes liés à la coexistence des deux niveaux de temps: le retard unitaire delta et le temps physique. Une solution serait de formaliser l'algorithme de simulation de VHDL dans l'algèbre des processus, qui permet de modèliser l'exécution concurrente de processus ne pouvant être réduite à une simple fonction de séquencement. Ainsi un opérateur de garde dans une telle algèbre exprimerait la liste de sensibilité et un opérateur de choix pourrait représenter la fonction de résolution.

Une telle sémantique pourrait servir de base pour valider qu'un modèle formel associé avec une description VHDL dans le cadre d'utilisation d'un démonstrateur donné est cohérent avec la définition originale de VHDL.

Une application de cette définition serait de prouver que notre sémantique de P-VHDL et notre modèle formel pour les primitives temporelles sont totalement (et non partiellement) cohérents avec l'algorithme de simulation de VHDL sous réserve des restrictions syntaxiques et sémantiques proposées dans cette thèse.

2. L'étude et la mise en oeuvre d'un système de vérification temporelle basé sur notre modèle formel. Pour cela, il faudrait définir l'ensemble des règles de réécriture nécessaires pour un tel système, et les prouver dans notre modèle formel. De plus, une comparaison entre cette approche de vérification temporelle et les techniques analytiques utilisées par les systèmes actuels serait indispensable.

Références Bibliographiques

[ACM91] C. ANGELO, L. CLAESEN, H. DE MAN: "A Methodology for Proving Correctness of Parametrized Hardware Modules in HOL", dans "Computer Hardware Description Languages and their applications", D.Borrione et R.Waxman ed., North Holland 1991.

[ACV92] C. ANGELO, L. CLAESEN, D. VERKEST, H. DE MAN: "On the comparison of HOL and Boyer-Moore for formal hardware verification", dans "Correct Hardware design methodologies", P. Prinetto et P.Camurati Ed., North Holland, 1992.

[Ak78] S. AKERS: "Binary Decision Diagrams", IEEE Transaction on Computers, Vol C-27, N°.6, Juin 1978.

[Au89] L. AUGUSTIN: "Timing Models in VAL/VHDL", Proc. ICCAD'89, pp. 122-125, Santa-Clara, Novembre 1989.

[Ba78] J. BACKUS: "Can programming be liberated from the Von Neumann Style? A functional style and its Algebra of Programs", CACM, Vol.21 N° 8, Août 1978.

[Ba84] H.G. BARROW: "Verify: A program for proving correctness of digital hardware Designs", Artificial Intelligence, Vol.24, pp. 437-491, 1984.

[BCD85] M. BROWNE, E. CLARKE, D. DILL, B. MISHRA: "Automatic verification of sequential circuits using temporal logic", dans "CHDL and their application", C.Koomen et T.Moto-oko ed.,Tokyo, North Holland, 1985.

[BCL91] D. BORRIONE, H. COLLAVIZZA, C. LE FAOU: "μSPEED: a Framework for Specifying and Verifying Microprocessors", Proc. ACM International Workshop on Formal Methods in VLSI Design, Miami, 9-11 Janvier 1991.

[BCM90] J.R. BURCH, E.M. CLARKE, K.L. McMILLAN, D.L. DILL: "Sequential circuits verification using symbolic model checking", Proc. 27th DAC, IEEE CS Press,1990.

[BEF90] A. BARTSCH, H. EVEKING, H.J. FAERBER, M. KELELATCHEW, J. PINDER, U. SCHELLIN: "LOVERT: A Logic Verifier of Register-Transfer Descriptions", dans "Formal VLSI Correctness Verification", L.Claesen ed., North Holland 1990.

[BGL92] H. BARRINGER, G. GOUGH, T. LONGSHAW, B. MONAHAN, M. PEIM, A. WILLIAMS: "Semantics and verification for Boolean Kernel ELLA using IO automata", dans "Correct Hardware design methodologies", P. Prinetto et P.Camurati Ed., North Holland, 1992.

[Bi87] J.P. BILLON: "Perfect Normal Forms for Discrete Functions", technical report, DSG/CRG/87014, centre de recherche BULL, France, 1987.

[BJ83] J. BARROS, B. JOHNSON: "Equivalence of the Arbiter, the Synchroniser, the Latch and the Inertial Delay", IEEE transaction on Computers, pp. 603-614, Juillet 1983.

[BL85] D. BORRIONE, C. LE FAOU: "Overview of CASCADE multi-level hardware description language and its mixed-mode simulation mechanisms", dans "CHDL and their application", C.Koomen et T.Moto-oko ed.,Tokyo, North Holland, 1985.

[BM88] R.S. BOYER, J S. MOORE: "A computational logic handbook", Academic Press 1988.

[Bo88] R. BOUTE: "System semantics: principles, applications and implementation", ACM Transaction on Programming Languages and Systems, Vol.10, N° 1, Janvier 1988.

[BP87] D. BORRIONE, J.L. PAILLET: "An approach to the formal verification of VHDL descriptions", research report N°683-I, IMAG/ARTEMIS, Grenoble, Novembre 1987.

[BP90] C. BAYOL, J.L. PAILLET: "Using TACHE for proving circuits", dans "Formal VLSI Correctness Verification", L.Claesen ed., North Holland 1990.

[BPS92a] D. BORRIONE, L. PIERRE, A. SALEM: "PREVAIL: "A Proof Environment for VHDL Descriptions", dans "Correct Hardware design methodologies", P. Prinetto et P.Camurati Ed., North Holland, 1992.

[BPS92b] D. BORRIONE, L. PIERRE, A. SALEM: "Formal Verification of VHDL Descriptions in the PREVAIL Environment", IEEE Design & Test of Computers, Vol.9, No. 2,Juin 1992.

[BPS92c] D. BORRIONE, L. PIERRE, A. SALEM: "Formal verification of VHDL descriptions in Boyer-Moore : first results", dans "VHDL for simulation, synthesis, formal proof of Hardware", J. Mermet Ed., Kluwer Academic Publishers, 1992.

[BS90] D. BORRIONE, A. SALEM: "Proving an on-line multiplier with OBJ3 and TACHE : a practical experience". dans "Formal VLSI Correctness Verification", L.Claesen ed., North-Holland, 1990.

[Br86] R.E. BRYANT: "Graph-based algorithms for Boolean functions manipulation", IEEE Trans. on Computers, Vol.C-35, N° 8, pp. 677-691, Août 1986.

[Br90] A. BRONSTEIN: "String-Functional Semantics and Boyer-Moore Mechanisation for the Formal Verification of Synchronous Circuits", PhD, Université de Stanford, rapport N° STAN-CS-89-1293, Décembre 1989.

[CBM89] O. COUDERT, C. BERTHET, J.C. MADRE: "Verification of synchronous sequential machines based on symbolic execution", dans "Automatic Verification Methods for Finite State Systems", LNCS N°407, pp 365-373, Springer-Verlag 1989.

[Cl90] CLSI:"VHDL Tool Integration Platform (VTIP)", CAD Language Systems, Rockville, MD, Avril 1990.

[Co88] A. COHN: "A proof of correctness of the VIPER microprocessor: the first level", dans "VLSI Specification, Verification and Synthesis", G.Birtwistle and P.A.Subrahmanyam ed., Kluwer Academic Publishers 1988.

[Co90] H. COLLAVIZZA: "Functional Semantics of Microprocessors at the Micro program Level and Correspondence with the Machine Instruction Level", Proc. EDAC'90, Glasgow, 12-15 Mars 1990.

[CP88] P. CAMURATI, P. PRINETTO: "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research", IEEE Computer, Juillet 1988.

[CST91] R.CAMPSANO, L.F. SAUNDERS, R.M.TABET: "VHDL as input for high-level synthesis", IEEE Design & Test of Computers, Mars 1991.

[DAC90] Proceedings of ACM IEEE 27th Design Automation Conf., IEEE CS Press, 1990.

[DAC91] Proceedings of ACM IEEE 28th Design Automation Conf., IEEE CS Press, 1991.

[DBJ92] A.DEBREIL, C.BERTHET, A.JERRAYA : "Symbolic Computation of Hierarchical and Interconnected FSMs", dans "VHDL for simulation, synthesis, formal proof of Hardware", J. Mermet Ed., Kluwer Academic Publishers, 1992.

[Da79] J.A. DARRINGER: "The application of Program Verification Techniques to Hardware Verification", Proc. ACM IEEE 16th Design Automation Conf., pp.375-381, Juin 1979.

[DD68] J. DULEY, D. DIETMEYER: "A digital system design language (DDL)", IEEE Transaction on computers, Vol. C-17, pp. 650-861, 1986.

[DJ92] A.DEBREIL, D.JAILLET: "Synchronous description in VHDL for formal proof and resulting guidelines proposed by BULL", BULL S.A., Juillet 1992.

[EM90] H. EVEKING, C. MAI: "Formal Verfication of Timing Conditions", Proc. EDAC'90, pp. 512-517, Glasgow, 12-15 Mars 1990.

[EV77] M.D. ERCEGOVAC, K.S.TRIVEDI : "On line algorithms for division and multiplication", IEEE Transaction on Computers, Vol. C-26, No. 7, pp 681-687, Juillet 1977.

[Ev85] H. EVEKING: "The application of CHDL's to the abstract specification of hardware", dans "CHDL and their application", C.Koomen et T.Moto-oko ed.,Tokyo, North Holland, 1985.

[Ev90] H. EVEKING: "Axiomatizing hardware description languages", Internationl Journal of VLSI Design, Vol.2, N° 3, 1990.

[Es85] Estelle: A formal description technique based on the extended transition model ISO, 1984. ISO/TC97/SC21.

[FP84] J. FONLUPT, M. PREISSMANN: "Ordering with conditions", CASCADE, rapport de recherche N° cRSI-IMAG-RR-001-23, Août 1984.

[GHM89] A. GUYOT, Y. HERREROS, J.M MULLER: "JANUS, an online multiplier / divider for manipulating large numbers", Proc. 9th Symposium on computer arithmetic, Santa monica, USA, Sept. 1989.

[Go79] M. GORDON: "The denotational description of programming languages", Springer-Verlag, 1979.

[Go85] M. GORDON: "A machine Oriented Formulation of Higher-order Logic", technical report 68, Computer Laboratory, University of Cambridge, 1985.

[Go86] M. GORDON: "Why Higher Order Logic is a good formalism for specifying and verifying hardware", dans "Formal aspects of VLSI design", 1985, G.Milne and P.A.Subrahmanyam ed., North Holland 1986.

[GW88] J.A. GOGUEN, T. WINKLER: "Introducing OBJ3", technical report SRI-CSL-88-9, Menlo Park, Août 1988.

[He90] J. HERBET: "Formal Reasoning about Timing and Function of Basic Memory Devices, dans "Formal VLSI Correctness Verification", L.Claesen ed., North Holland 1990.

[HLP86] N. HALBWACHS, A. LONCHAMP, D. PILAUD: "Descriping and Designing Circuits by means of a synchronous declarative language", dans "From HDL descriptions to guaranteed correct circuit designs", North-Holland, Grenoble, Septembre 1986.

[HSC82] R.HITCHCOCK, G.SMITH, D.CHENG "Timing Analysis of Computer Hardware", IBM Journal of Research and Development, Vol. 26,No.1, Jan. 1982,pp. 100-105

[Hu86] W.A. HUNT: "FM8501: A verified microprocessor", Institute for Computing Science, University of Texas, Austin, technical report N°47, Février 1986.

[Ie88] IEEE: IEEE Standard VHDL Language Reference Manual, IEEE, 1988.

[JM87] D. JAILLET, P. MERTENS: "LDS Reference Manual", BULL S.A., Mai 1987.

[Kh74] G. KAHN: "The Semantics of a Simple Language for Parallel Programming", Proc. IFIP Congress 74. Amsterdam, 1974.

[Ko87] Z. KOHAVI: "Switching and finite automata theory", McGraw-hill, 1987.

[Ma90] J.C MADRE: "PRIAM: Un Outil de vérification formelle de circuits intégrés digitaux", Thèse de doctorat, E.N.S. des Télécommunications, Paris, Juin 1990.

[MB88] J.C. MADRE, J.P. BILLON: "Proving circuit correctness using formal comparison between expected and extracted behaviour", Proc. 25th ACM-IEEE Design Automation Conference, Juin 1988.

[Mc89] W. McCUNE: "OTTER 1.0 Users' Guide", Argonne National Laboratory, Argonne, Illinois, Janvier 1989.

[Mi85] G. MILNE: "CIRCAL and the representation of communication, concurrency and time", ACM Transaction on Programming Languages and Systems, Vol.7, N° 2, Avril 1985.

[MPT85] J.D. MORISON, N.E. PEELING, T.L. THORP: "The design rationale of Ella, a hardware design and description language", dans "CHDL and their application", C.Koomen et T.Moto-oko ed.,Tokyo, North Holland, 1985

[Pa86] J.L. PAILLET: "A Functional Model for description and specifications of digital devices", dans "From HDL descriptions to guaranteed correct circuit designs", North-Holland, Grenoble, Septembre 1986.

[Pa89] J.L. PAILLET: "Functional Semantics of Microprocessors at the Machine Instruction Level", Proc. IFIP WG10.2 9th Int. Symp. on Computer Hardware Description Languages CHDL'89, Washington, Juin 1989.

[Pa90] J.L. PAILLET: "Algèbre de fonction: propriétés logiques et preuve de circuits digitaux", mémoire d'habilitation à diriger des recherches, Université de Provence, Marseille, Février 1990.

[Pl88] J.A.PLAICE: "Sémantique et compilation de LUSTRE, un language déclaratif synchrone", Thèse de doctorat, INP, Grenoble, Mai 1988.

[Pi90] L. PIERRE: "Représentation fonctionnelle et preuve automatisée de circuits digitaux", Thèse de doctorat, Université de Provence, Marseille, Décembre 1990.

[Pi91] L. PIERRE: "One Aspect of Mechanizing Formal Proof of Hardware: the Generalization of Partial Specifications", Proc. ACM International Workshop on Formal Methods in VLSI Design, Miami, 9-11 Janvier 1991.

[QS82] J.P.QUEILLE, J.SIFAKIS: "Specification and verification of concurrent system in CESAR", LCS-137 , Springer-Verlag, Avril 1982.

[Ro88] C.RODRIGUEZ : "Spécification et validation de système en XESAR" . Thèse de doctorat, INPG, Mai 1988.

[RRSV87] J.L.RICHIER, C.RODRIGUEZ, J.SIFAKIS, J.VOIRON : "Verification in XESAR of the sliding window protocol", 17th International workshop on protocol specification testing and verification, Zurich, Suisse, North Holland, 1987.

[Sa89] A. SALEM: "Formal verification of the JANUS on-line multiplier", rapport de recherche N° 785-I, IMAG/ARTEMIS, Grenoble, Juillet 1989.

[SB90] A. SALEM, D. BORRIONE: "Automatic Formal Verification of VHDL descriptions: a first prototype", rapport de recherche N° 823-I, IMAG/ARTEMIS, Grenoble, Juin 1990.

[SB91] A. SALEM, D. BORRIONE: "Formal semantics of VHDL timing constructs", dans Proc. Euro-VHDL, Stockholm, Septembre 1991, et dans "VHDL for simulation, synthesis, formal proof of Hardware", J. Mermet Ed., Kluwer Academic Publishers, 1992.

[Sc88] D. SCHMIDT: "Denotational Semantics", Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[SGE92] V. STAVRIDOU, J. GOGUEN, S. M.EKER, S. ALONEFIS: "FUNNEL: A CHDL with Formal Semantics", dans "Correct Hardware design methodologies", P. Prinetto et P.Camurati Ed., North Holland, 1992.

[Sh84] M. SHEERAN: "muFP, a language for VLSI design", ACM Symposium on LISP and FUNCTIONAL Programming, Austin, Texas, 1984.

[Ta90] J. TASSEL: "The semantics of VHDL with VAL and HOL: Towards practical verification tools", Technical Report 196, University of Cambridge, Cambridge, Juin 1990.

[Ta92] J. TASSEL: "A Formalisation of the VHDL simulation cycle", Technical Report 249, University of Cambridge, Cambridge, Mars 1992.

[Te81] R. TENNENT: "Principles of Programming Languages", Prentice-Hall, Englewood Cliffs, N.J., 1981.

[UMS81] T. UEHARA, F. MARUYAMA, T. SAITO, N. KAWATO: "DDL verifier", dans "Computer hardware description languages and their applications", M.Breuer et R.Hartenstien ed., North-Holland, 1981.

[VCM90] D. VERKEST, L. CLAESEN, H. DE MAN: "Correctness proof of parameterized hardware modules in the Cathedral-II synthesis environment", Proc. EDAC'90, Glasgow, 12-15 Mars 1990.

[Wa77] T.J.WAGNER : "Verification of Hardware Designs Through Symbolic Manipulation" Int'l Symp. Design Automation and Microprocessors, Fep. 1977, pp.50-53.

Table des matières

Chapitre 4

Définition Formelle de P-VHDL

Chapitre 5

Sémantique Formelle des Primitives Temporelles du Langage VHDL

Résumé

L'objet de cette thèse est la vérification formelle des circuits digitaux décrits en VHDL. Nous avons, en premier lieu restreint VHDL pour le rendre utilisable par les outils de preuve existants, en proposant un sous-ensemble, appelé P-VHDL, afin de décrire les circuits combinatoires et les circuits séquentiels synchrones. Un tel sous ensemble a bien évidemment une sémantique beaucoup plus simple que celle de VHDL complet. En fait, le retard unitaire delta a été remplacé par une simple fonction de séquencement. Et l'échelle de temps devient la période de l'horloge. Ainsi, la machine d'état fini a pu être utilisée comme modèle formel pour le sous-ensemble. L'équivalence entre ce modèle et la sémantique de VHDL a été montrée sous les restrictions syntaxiques et sémantiques imposées par P-VHDL. Ce modèle est à la base de l'écriture d'un compilateur en vue de la preuve qui constitue le coeur de l'environnement de vérification formelle PREVAIL. Puis nous avons défini une sémantique dénotationnelle pour P-VHDL. Pour cela, nous avons proposé trois domaines différents pour les trois objets porteurs des valeurs: les variables, les signaux et les registres. Ensuite, nous avons proposé une sémantique formelle pour les primitives temporelles de VHDL, et nous avons prouvé, partiellement, l'équivalence entre cette sémantique et la sémantique opérationnelle informelle de VHDL. Enfin, nous avons montré comment notre sémantique peut constituer une base de construction d'un système de vérification temporelle.

Mots-Clés

Langages de description de matériel, sémantique dénotationnelle, vérification formelle de circuits digitaux, modélisation de circuits, vérification temporelle.

Abstract

The subject of this thesis is the formal verification of digital circuits described in VHDL. First, we restricted VHDL to make it processable by existing proof tools, and defined a subset, called P-VHDL, dedicated to the description of combinational and synchronous sequential circuits. The semantics of this subset is much simpler than the complete VHDL. In fact, we replaced the delta delay by a simple serialisation function, and the time scale has been chosen equal to the clock period. Thus, the use of the finite state machine as a formal model for the subset became possible. The finite state machine semantics has been shown to represent the P-VHDL semantics. Based on this formal model, we wrote a proof oriented compiler for P-VHDL. Then, we defined a complete denotational semantic for P-VHDL. We proposed three different domains for the three values holders in the language: the variables, the signals and the registers. Finally, we gave formal semantics for the VHDL timing constructs. We partially proved the equivalence between these semantics and the VHDL informal operational semantics. And we showed how our semantics can form a basis for building a formal timing verifier.

Keywords

Hardware description languages, denotational semantics, formal verification of digital circuits, circuits modelling, timing verification

Résumé

L'objet de cette thèse est la vérification formelle des circuits digitaux décrits en VHDL. Nous avons, en premier lieu restreint VHDL pour le rendre utilisable par les outils de preuve existants, en proposant un sous-ensemble, appelé P-VHDL, afin de décrire les circuits combinatoires et les circuits séquentiels synchrones. Un tel sous ensemble a bien évidemment une sémantique beaucoup plus simple que celle de VHDL complet. En fait, le retard unitaire delta a été remplacé par une simple fonction de séquencement. Et l'échelle de temps devient la période de l'horloge. Ainsi, la machine d'état fini a pu être utilisée comme modèle formel pour le sous-ensemble. L'équivalence entre ce modèle et la sémantique de VHDL a été montrée sous les restrictions syntaxiques et sémantiques imposées par P-VHDL. Ce modèle est à la base de l'écriture d'un compilateur en vue de la preuve qui constitue le coeur de l'environnement de vérification formelle PREVAIL. Puis nous avons défini une sémantique dénotationnelle pour P-VHDL. Pour cela, nous avons proposé trois domaines différents pour les trois objets porteurs des valeurs: les variables, les signaux et les registres. Ensuite, nous avons proposé une sémantique formelle pour les primitives temporelles de VHDL, et nous avons prouvé, partiellement, l'équivalence entre cette sémantique et la sémantique opérationnelle informelle de VHDL. Enfin, nous avons montré comment notre sémantique peut constituer une base de construction d'un système de vérification temporelle.

Mots-Clés

Langages de description de matériel, sémantique dénotationnelle, vérification formelle de circuits digitaux, modélisation de circuits, vérification temporelle.

Abstract

The subject of this thesis is the formal verification of digital circuits described in VHDL. First, we restricted VHDL to make it processable by existing proof tools, and defined a subset, called P-VHDL, dedicated to the description of combinational and synchronous sequential circuits. The semantics of this subset is much simpler than the complete VHDL. In fact, we replaced the delta delay by a simple serialisation function, and the time scale has been chosen equal to the clock period. Thus, the use of the finite state machine as a formal model for the subset became possible. The finite state machine semantics has been shown to represent the P-VHDL semantics. Based on this formal model, we wrote a proof oriented compiler for P-VHDL. Then, we defined a complete denotational semantic for P-VHDL. We proposed three different domains for the three values holders in the language: the variables, the signals and the registers. Finally, we gave formal semantics for the VHDL timing constructs. We partially proved the equivalence between these semantics and the VHDL informal operational semantics. And we showed how our semantics can form a basis for building a formal timing verifier.

Keywords

Hardware description languages, denotational semantics, formal verification of digital circuits, circuits modelling, timing verification