THESE DE DOCTORAT

de

L'UNIVERSITE Pierre et Marie CURIE

Spécialité:

Informatique

Présentée par:

Jean Bruce Guignet

Pour obtenir le grade de:

DOCTEUR de l'UNIVERSITE Pierre et Marie CURIE

Sujet de la thèse:

Abstraction Fonctionnelle des Composants VLSI

Application de techniques de reconnaissance de formes
à l'Abstraction Fonctionnelle des composants VLSI.

Soutenue le:

31 Mars 1998

Devant le Jury composé de:

Mr Jacques Pitrat, Président

Mme Dominique Borrione, Rapporteur

Mr Dominique Lavenier, Rapporteur

Mr Alain Greiner

Mr Philippe Oddo

UNIVERSITE PIERRE ET MARIE CURIE - 4, PLACE JUSSIEU - 75252 PARIS CEDEX 05

Laboratoire de Thèse:

LIP6

Laboratoire d'Informatique de Paris 6

CNRS - UPMC (ERS 587)

Table des matières

1. Introduction

1.1 Sujet

L’industrie électronique et informatique conçoit, réalise et utilise des composants intégrés à haute densité, appelés VLSI et constitués de plusieurs millions de transistors.

Un composant VLSI étant très complexe, il est décomposé en unités de moindre complexité pour des raisons à la fois techniques et humaines. D’un point de vue structurel, chaque composant VLSI se présente donc dès le début de sa conception comme un assemblage hiérarchique de plusieurs sous ensembles appelés blocs dont chacun réalise individuellement une fonction logique particulière. Ces blocs sont alors conçus indépendamment par des équipes spécialisées à l’aide de logiciels de Conception Assistée par Ordinateur. Les descriptions au niveau masque de chaque bloc seront finalement assemblées, après **vérification**, pour constituer la description complète du composant.

C’est dans le cadre de cette phase de vérification que s’intègre l’abstraction fonctionnelle. Deux processus entrent alors en oeuvre, lesquels permettent de déduire le fonctionnement d’un bloc en partant de la description informatique des masques. Ce sont:

- un processus dit d’**extraction**.
  Il permet de déduire la description en terme de connectique de transistors à partir de la seule description du dessin des masques.
- un processus dit d’**abstraction** fonctionnelle.
  Il exploite la connectique extraite pour en déduire une description fonctionnelle dite abstraite qui utilise un langage de description comportemental. C’est ce deuxième processus qui nous concerne dans cette étude.

L’objectif essentiel d’une telle opération de vérification est à priori de s’assurer que le comportement du composant qui doit être réalisé corresponde bien à sa spécification initiale. Cette vérification pourra consister par exemple en une comparaison formelle du la description fonctionnelle abstraite avec la spécification initiale du composant.

A partir de là, nous voulons obtenir un logiciel qui soit utilisable dans différents environnements de CAO des composants électroniques.

L’approche que nous proposons ici est basée sur des mécanismes de reconnaissance de formes de façon à ne pas être limitée à une technologie particulière. Concernant ce dernier point, nous montrerons, à partir de circuits BiCMos, comment la méthode peut s’appliquer aisément à des technologies autres que le CMos.

Nous verrons également quelles utilisations de l’abstraction autres que la vérification peuvent être envisagées et comment ce procédé peut aussi faciliter la migration technologique.

1.2 Plan

Le **chapitre 2** intitulé **"Les enjeux et les objectifs"** décrit le contexte d'exploitation de l'abstraction fonctionnelle et les besoins industriels qui orientent l'étude. Il fixe par là les objectifs à atteindre tout en répondant aux exigences des standards de l'industrie. On analyse l'impact de ces contraintes applicatives sur le choix des solutions.

Le **chapitre 3** intitulé **"L'état de l'art"** donne l'état présent des techniques d'abstraction. Un bref historique est suivi d'un aperçu des principes généraux qui sont applicables pour mener à bien une telle activité. Il permet de mieux situer cette étude par rapport à l'existant et plus particulièrement de justifier l'intérêt de la méthode proposée.

Dans le **chapitre 4** intitulé **"Présentation de la méthode"**, on décrit l'ensemble des concepts qui fondent la méthode proposée. Les éléments théoriques sont fournis dans ce chapitre. Les principes sont illustrés par des exemples qui permettent de se rapprocher des cas concrets de l'analyse des circuits. Le langage de règles est également décrit dans ce chapitre.

Le **chapitre 5** intitulé **"Réalisation logicielle"** a pour objectif de décrire l'architecture logicielle de l'outil Grog qui a été développé pour valider la méthode, et de présenter plus en détail l'algorithmique mise en oeuvre et les objets manipulés. Nous voyons comment des techniques issues de la programmation orientée objet et de l'intelligence artificielle se trouvent ici judicieusement associées pour résoudre le problème très spécifique de l'abstraction des circuits électroniques.

Le **chapitre 6** intitulé **"Tests et résultats"** traite des résultats obtenus et décrit l'ensemble des tests qui permettent d'évaluer le logiciel. Sont ainsi mis en évidence des éléments tels que la facilité d'utilisation, la facilité de prise en compte de nouveaux circuits, l'aptitude à traiter plusieurs technologies, et les performances obtenues.

Enfin, le **chapitre 7** intitulé **"Conclusions"** rappelle en conclusion les principaux résultats obtenus tant sur le plan méthodologique que sur le plan expérimental. On introduit également quelques perspectives de développement futur.

2. Les enjeux et les objectifs

2.1 Les besoins applicatifs

Nous estimons que la méthode d'abstraction doit prendre en considération les besoins suivants:

1. **être le moins possible dédié à un procédé de fabrication**

   Le contexte technologique et industriel dans lequel les composants sont conçus puis fabriqués est en constante évolution. Alors que la conception des composants VLSI passe par la réalisation d'investissements importants en savoir faire et en logiciels, ceux ci se voient régulièrement remis en cause du fait de l'évolution des techniques de fabrication: Bipolaires, Nmos, Cmos, AsGa, BiCmos. Nous voulons donc que la méthode soit applicable à plusieurs technologies.

2. **prendre en compte différents types de circuiterie**

   Nous savons que chaque technologie employée autorise une grande variété de styles de réalisation. Les circuits électroniques qui sont réalisés dans une technologie donnée font appel à des techniques optimisées et originales spécifiques à chaque entreprise, voire à chaque équipe de concepteurs. Nous voulons aussi pouvoir prendre en compte facilement tous les types de circuiterie [IntroVlsi80], [Pcvd85].

3. **exploiter toute la richesse des langages de description fonctionnelle**

   Les spécifications des divers langages de description fonctionnelle qui sont utilisés pour modéliser les composants évoluent. Nous voulons que notre logiciel d'abstraction soit suffisamment souple pour pouvoir exprimer les comportements des circuits en mettant à profit toute la richesse sémantique qui est permise par les langages de description fonctionnelle, et qu'il puisse aussi s'adapter aisément à ces évolutions.

2.2 Les utilisations visées

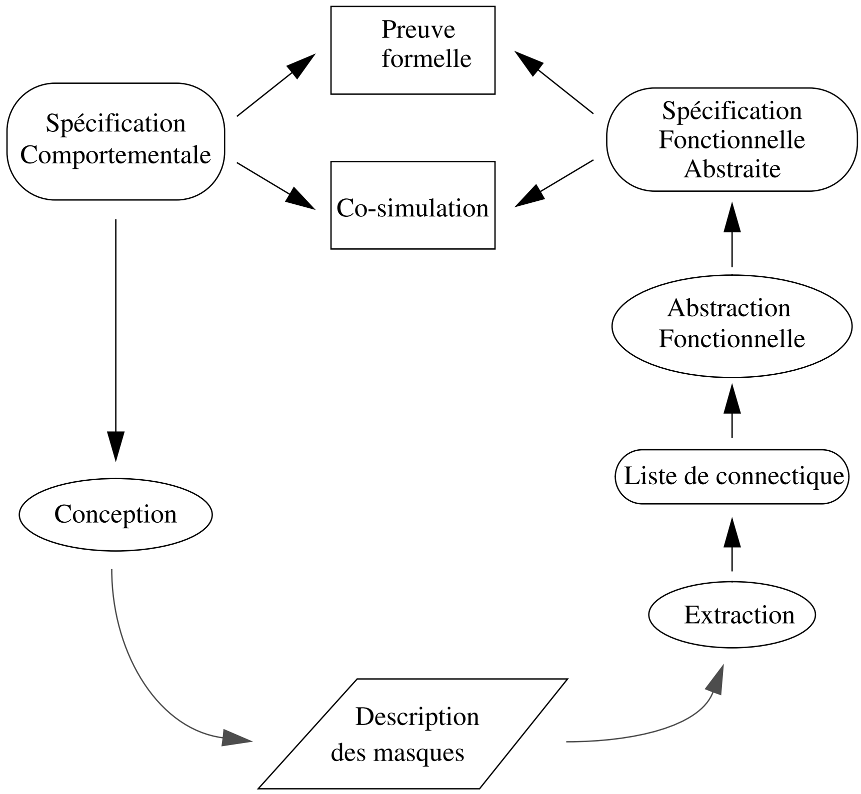
Nous proposons deux utilisations à l'abstraction fonctionnelle qui sont la vérification d'une part, et la migration technologique d'autre part. C'est ce que nous allons maintenant décrire.

2.2.1 Abstraction et vérification

L'abstraction doit s'intégrer dans les processus de vérification fonctionnelle déjà connus qui permettent de comparer fonctionnellement par **co-simulation** ou par **preuve formelle**

[Priam88], [Paris91], [Vformal] une description obtenue par abstraction avec une spécification comportementale initiale. C'est ce qu'illustre la Figure 2-1.

Figure 2-1 utilisations de l'abstraction fonctionnelle en vérification



Les procédés de preuve formelle ou de simulation envisagés doivent pouvoir exploiter directement les résultats issus de l'abstracteur. Le fait de partir du niveau de description le plus proche possible de la réalisation a pour principal avantage celui de se garantir vis à vis de tous les aléas, que ceux ci proviennent des logiciels ou bien des moyens humains qui sont mis en oeuvre dans la chaîne de conception prise dans son ensemble.

Dans un tel contexte de vérification, la méthode sera utilisable aussi bien dans le cadre de la conception des composants ou des blocs dédiés, que dans celui de la conception des bibliothèques de cellules. C'est là en effet que le processus de conception fait le plus appel à l'intervention humaine.

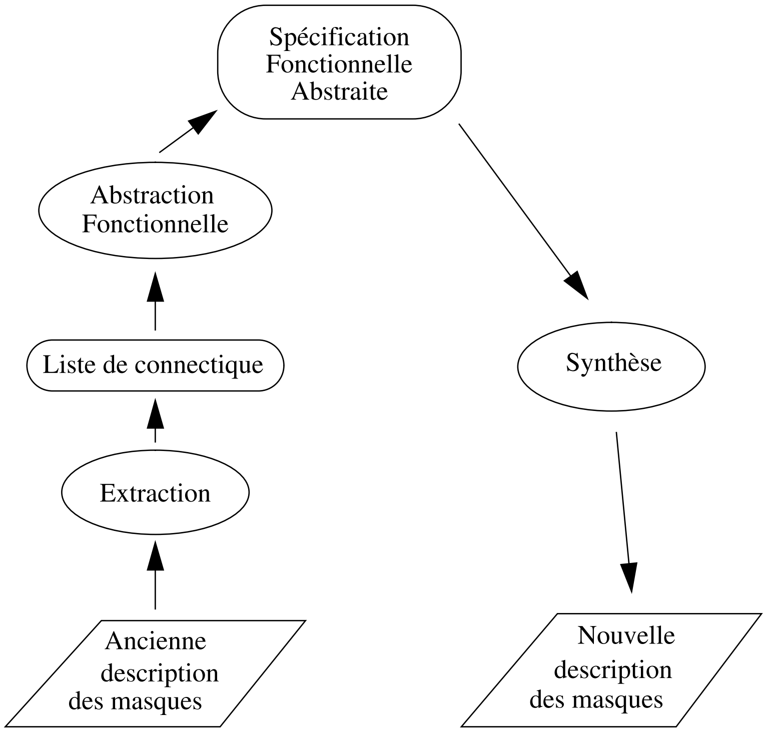
2.2.2 Abstraction et migration technologique

L'abstraction doit pouvoir servir de base à un processus de migration technologique en permettant de ré-utiliser un travail de conception effectué auparavant dans une ancienne technologie.

Nous avons vu que l'abstraction fonctionnelle permettait d'obtenir une description fonctionnelle à partir de n'importe quelle description physique de composant. De plus, la plupart des outils de synthèse [CompSynth],[SynopComp] utilisent comme donnée d'entrée un descriptif du comportement décrit dans un langage tel que Vhdl [VhdlIeee87], [VhdlMod90], [VhdlCook90] ou Verilog[VerilogIeee95].

Il devient alors possible d'obtenir très rapidement grâce à ce processus d'abstraction suivi de synthèse un composant plus performant que le composant d'origine. C'est ce processus qu'illustre la Figure 2-2.

Figure 2-2 utilisation de l'abstraction fonctionnelle en migration technologique



Notons en complément que comme l'abstraction fonctionnelle produit une description textuelle de type flot de données en langage Vhdl ou Verilog à partir d'une réalisation physique ou schématique, cette description peut aussi contribuer à documenter le composant lorsque la spécification initiale est manquante ou bien établie dans un autre langage.

2.3 Les objectifs

Nous venons de voir que la conception des composants micro-électroniques est un domaine complexe, en constante évolution, et permettant une grande variété de styles de circuiterie dont il est impensable de vouloir établir un inventaire exact a priori.

C'est pourquoi, dans un tel cadre, nous nous proposons de définir une méthode d'abstraction qui:

- prenne en entrée une description de type *net-list* de transistors extraite du dessin des masques,
- produise une description fonctionnelle dans un ou plusieurs langages qui soient des standards commerciaux ; ceux ci seront en priorité les langages Vhdl et Verilog,
- permette au niveau applicatif de couvrir simultanément les deux domaines de la vérification fonctionnelle et de la migration technologique des composants.

Nous voulons aussi plus particulièrement que cette méthode:

- soit assez souple pour pouvoir être configurée afin de prendre en compte n'importe quel style de conception dans chacune des technologies micro-électroniques qui sont exploitées à ce jour,
- puisse être complètement configurée en fonction de la cible technologique, qu'il s'agisse d'une technologie Mos, Bipolaire ou BiCMos,
- accepte tous les schémas de circuit possibles,
- permette d'exprimer avec toute la précision requise les comportements individuels des circuits reconnus, que ce soit en logique synchrone ou asynchrone, et ceci à l'aide des deux langages Vhdl et Verilog,
- produise en fin d'abstraction, à partir des comportements individuels des circuits reconnus, des comportements Vhdl et Verilog résultants qui soient organisés et compacts, et prêts à être lus par des compilateurs commerciaux.

3. L’état de l’art

Nous décrivons dans ce chapitre quelques méthodes d’abstraction prises parmi celles qui ont eu une application industrielle significative. Nous examinons les particularités de chaque méthode et nous analysons leurs limites au travers des programmes qui les ont mises en oeuvre.

Nous verrons que ces programmes sont généralement spécialisés dans la technologie CMos, laquelle est aujourd’hui la technologie la plus fréquemment employée pour la réalisation des composants VLSI. Il se trouve aussi que cette technologie Mos se prête particulièrement bien à des approches de type algorithmique d’analyse des réseaux de transistors [AlgoSwi87], [Cosmos87], [SigFlow87], [FMetho87], [Logex88], [Mosi80], [NetwPart87], [Survey87].

Les programmes présentés sont représentatifs des deux méthodes d’abstraction qui sont l’**analyse de réseaux de transistors** et la **reconnaissance de formes**:

- les méthodes d’analyse de réseaux de transistors sont généralement utilisées dans le cadre de la technologie CMos. Ces méthodes sont complexes car il faut prendre en compte non seulement les aspects électriques tels que la bidirectionnalité des transistors Mos, mais aussi le stockage dynamique des charges électriques.
- le principe des méthodes utilisant la reconnaissance de formes consiste à identifier dans un circuit toutes les structures qui sont connues en référence à des modèles prédéfinis. Il devient ensuite possible d’associer à chaque structure reconnue le comportement logique qui lui correspond. De telles techniques exigent de définir à priori un catalogue des modèles de structures à reconnaître.

3.1 Les logiciels utilisant une méthode d’analyse de réseaux

3.1.1 ANAMOS

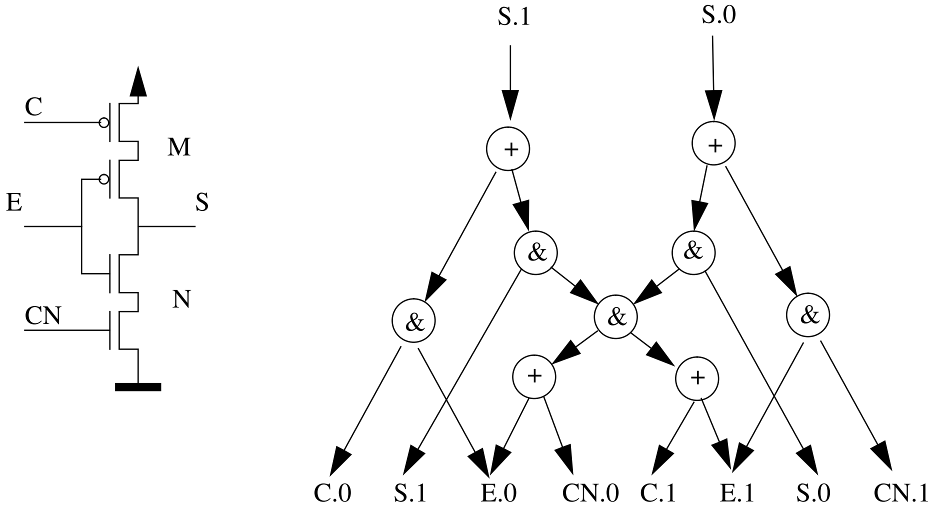
Anamos [Cosmos87] est le pré-processeur du simulateur *Cosmos* (COmpiled Simulator for Mos circuits). En tant qu’analyseur symbolique, il permet de compiler une description au niveau transistor en un ensemble de formules booléennes.

Anamos partitionne un circuit Mos en un ensemble de sous réseaux, et ce sont ces sous réseaux, appelés aussi *CCC* (*Connected Chanel Components*) qui attaquent les grilles des transistors d’autres *CCC*. Le graphe formé au niveau de l’assemblage des sous réseaux est donc orienté, toute entrée de grille étant commandée par ce qui est supposé être une sortie d’un autre *CCC*.

A chaque noeud de sortie d’un *CCC* est ensuite affectée une paire de formules qui évalue l’état du noeud correspondant en fonction des états d’entrée grâce à des fonctions de traitement symbolique.

Le comportement de tout *CCC* est alors exprimé lors de l'analyse du circuit par un graphe orienté acyclique dont les feuilles sont les variables. C'est ainsi qu'est représentée à titre d'exemple en Figure 3-1 la fonction booléenne du *CCC* d'une porte trois états:

Figure 3-1 représentation booléenne par *Anamos* d'une porte trois états



Les feuilles de cet arbre représentent l'encodage booléen des entrées. *S* est le seul noeud observable de l'extérieur du *CCC*. Lorsque la porte se trouve en état haute impédance, l'état de *S* dépend de la charge qui est présente sur cette même sortie *S*, ce qui est exprimé ici par l'introduction de *S.0* et *S.1* en tant que terminaux du graphe. Ce système produit une description hiérarchique simulable, les simplifications ne sont effectuées qu'au niveau de chaque *CCC*. On voit sur cet exemple que le partitionnement ne permet pas de prendre en compte localement la complémentarité entre *C* et *CN*, ce qui permettrait d'éviter l'expression du faux conflit pour lequel on aurait simultanément *S.0* et *S.1*.

L'avantage de cette approche est d'être très générale et de prendre en compte systématiquement les éléments mémorisants dynamiques. Son inconvénient réside dans la possible explosion combinatoire lorsque le nombre d'entrées devient trop élevé, et pour ce qui nous concerne, sa principale limitation est d'être réservée à la technologie CMos.

3.1.2 LOGEX

Logex (LOGical EXtractor) [Logex88] a été développé chez Siemens pour obtenir à partir d'une netlist de transistors CMos une description en terme de porte en vue de la simulation. La reconnaissance des portes n'est pas limitée à la circuiterie de type dual et prend aussi en compte les logiques de type dynamique.

Pour cela *Logex* va d'abord classifier les transistors en unidirectionnels ou bidirectionnels, et reconnaître ensuite des branches de type série et parallèle à partir des règles suivantes:

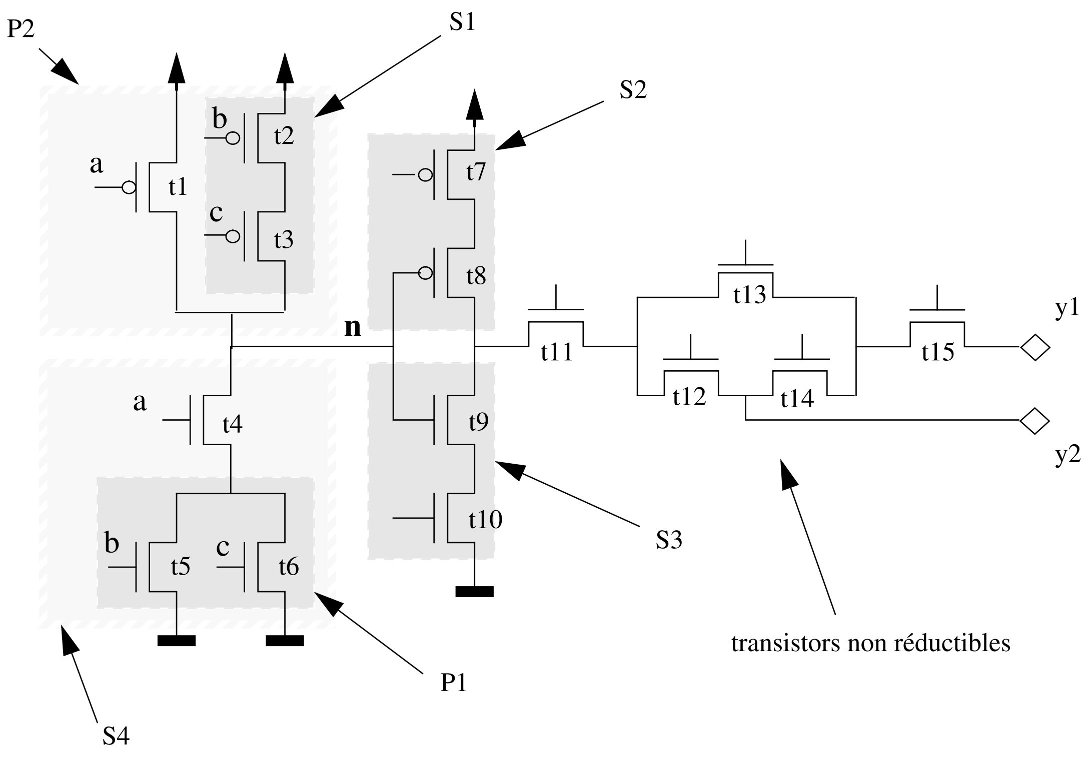
- ***règle 1***: une branche est un chemin de conduction entre 2 terminaux,
- ***règle 2***: un transistor constitue la plus petite branche possible, et ses deux connection de drain et de source sont les deux terminaux utilisés pour effectuer les regroupements,
- ***règle 3***: des branches sont dites parallèles dès lors qu'elles partagent les deux mêmes terminaux,

- ***règle 4***: une branche dite série est constituée par 2 branches connectées à une même noeud de façon à ce que ce noeud n'ait pas d'autre accès (ni port, ni alimentation ni autre branche).

Des regroupements de branches de type série et parallèle sont alors ainsi constitués, et les branches obtenues sont réparties selon 3 catégories: les branches unidirectionnelles connectées à VSS, les branches unidirectionnelles connectées à VDD, et les branches bidirectionnelles placées entre des noeuds autres que VSS et VDD. D'après les types de construction obtenus, les équations de conduction des branches sont déduites automatiquement de la façon suivante: le NON logique d'inversion est obtenu à partir des signaux de grille des transistors mos P, la transmission est obtenue à partir des signaux de grille des transistors mos N, le OU logique est appliqué sur les équations de plusieurs branches en parallèle, et le ET logique sur les équations des branches en série.

C'est ce que montre l'exemple de la Figure 3-2 suivante qui est extrait de [Logex88], et dans lequel on peut identifier les regroupements de type série (*S1, S2, S3* et *S4*) et parallèle (*P1* et *P2*), ainsi que l'orientation des branches *S4* et *S3* vers *vss,* et *P2* et *S2* vers *vdd* qui en résulte. On voit aussi que les transistors *t11* à *t15* ne sont pas réductibles d'après les règles de regroupement qui viennent d'être définies, et ceci à cause de l'accès au noeud intermédiaire *y2*.

Figure 3-2 regroupement des transistors en branches par *Logex*

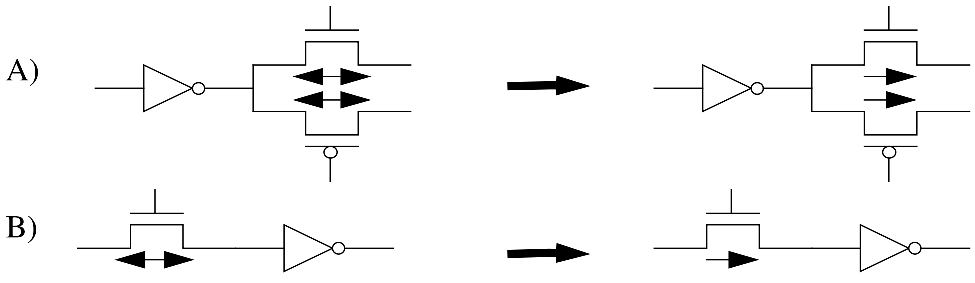


Il reste ensuite à orienter les branches bidirectionnelles, mais auparavant, *Logex* va examiner les branches de type N préalablement orientées vers *vss,* puis les branches de type P préalablement orientées vers *vdd,* et ceci afin de trouver des cellules telles que des réseaux CMos complémentaires, comme le sont dans notre exemple les réseaux *P2* et *S4*. Pour chaque ensemble de branches complémentaires qui est trouvé, une équation unique vient alors enrichir un catalogue, tandis que la branche P du réseau est supprimée, ce qui toujours dans notre exemple donnera l'équation du noeud *n* d'après *S4* en supprimant *P2* pour avoir *n = NON (a ET (b OU c)).* Sont également traités lors de cette étape des cas particuliers de connectique comme par exemple les transistors de type *bleeder* qui sont supprimés.

Ensuite, dès lors qu'il reste des branches bidirectionnelles, *Logex* va tenter de les orienter en utilisant tout d'abord deux heuristiques repérées par A et B qui sont illustrées en Figure 3-3:

- lorsque l'extrémité d'une branche est reliée à ce qui est identifié comme étant une sortie de cellule de type inverseur ou porte logique, alors il s'agit d'une entrée (A).
- lorsque l'extrémité d'une branche est reliée à ce qui est identifié comme étant une entrée signal ou un contrôle de cellule de type inverseur ou porte logique, alors il s'agit d'une sortie, mais seulement si cette branche est unique (B).

Figure 3-3 heuristiques d'orientation des branches dans *Logex*



Pour orienter les branches restantes, d'autre règles sont appliquées mettant en jeu une notion de force, cette force étant affectée aux extrémités de chaque branche.

L'intérêt de l'approche de *Logex* est de pouvoir traiter de façon générale bon nombre de cas. Mais comme nous venons de le voir, le fait d'appliquer tout d'abord des réductions de type série parallèle, puis de mettre en oeuvre des règles d'orientation, ne permet pas toujours d'identifier correctement tous les circuits, certains cas particuliers pouvant mettre la méthode en défaut. De plus, cette méthode ne s'applique qu'à la circuiterie CMos.

3.1.3 DESB

Le programme *Desb* [Desb92], [Desb95] a été développé à l'université de *Paris6*. Il est adapté à la technologie CMos et il est construit autour d'une méthode d'analyse de réseaux de transistors originale puisque basée sur une décomposition du réseau analysé en cônes logiques non nécessairement disjoints.

Desb part du principe que chaque grille de transistor est toujours à un état déterminé par un chemin vers le '0' ou vers le '1' logique (alimentations *vss* et *vdd*). Ces chemins constituent des branches de type '*vss*' ou '*vdd*' selon qu'ils mènent à l'une ou l'autre de ces alimentations. L'état logique de chaque signal est alors déterminé par un état logique des branches '*vss*' et *vdd*' auquelles il est relié et qui forment un cône logique.

La transformation de la connectique en branches préalable à celle de ces branches en cônes est conditionnée par trois règles qui sont:

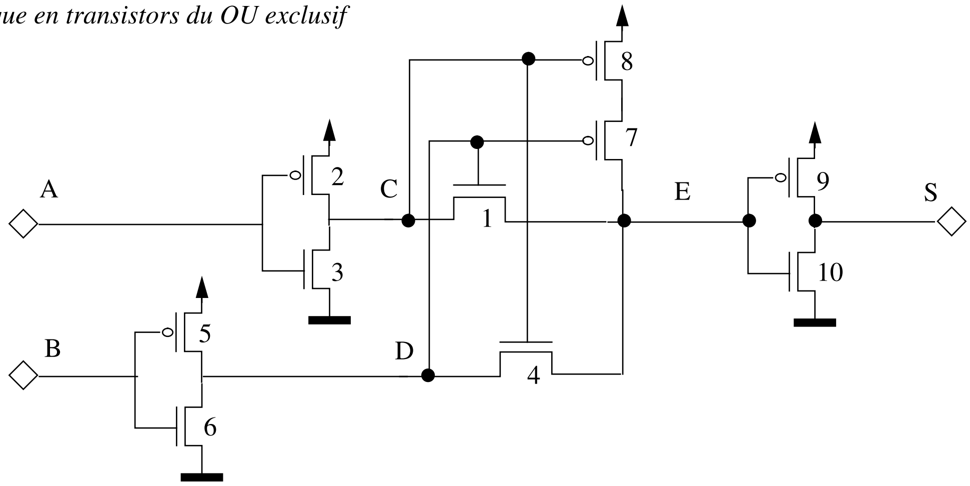
- ***règle 1***: une branche ne passe pas 2 fois par le même signal
- ***règle 2***: une branche ne contient jamais 2 transistors de type différent contrôlés par le même signal.

- ***règle 3***: en parcourant une branche dans un sens quelconque, il ne doit pas être possible de rencontrer sur un drain (ou sur une source) un signal qui attaque la grille d'un transistor préalablement parcouru.

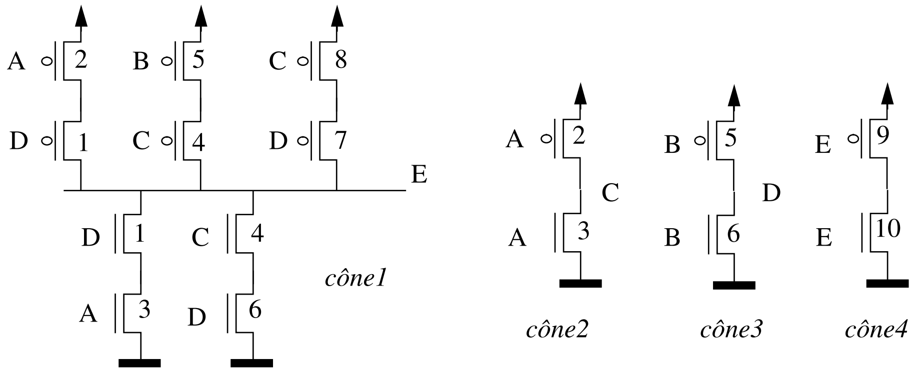
La transformation par *Desb* de la connectique en cônes de logique est illustrée par l'exemple de la Figure 3-4. On voit que dans cette méthode, un même transistor peut appartenir à plusieurs cônes. On n'a donc pas affaire ici à une partition stricte du circuit puisque les cônes peuvent être recouvrants.

Figure 3-4 exemple de transformation en cônes de logique d'un OU exclusif

a) connectique en transistors du OU exclusif

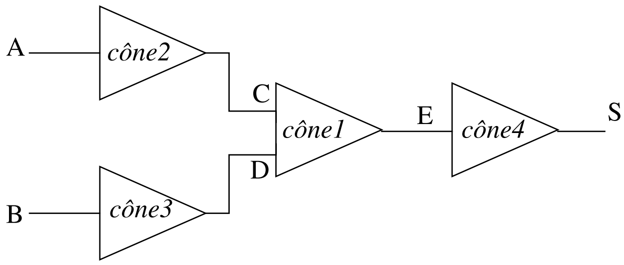


c) décomposition en cônes



REMARQUE: *par application de la règle 3, lors de la construction des branches associées au signal C (qui contrôle les grilles des transistors 4 et 8), on ne prend pas en compte les chemins 1/7/8, 1/4/5, et 1/4/6.*

c) orientation du réseau de cônes



On obtient ainsi un réseau de cônes orientés assimilables à des portes et dont on peut exprimer les équations de sortie en fonction des entrées. Ceci permet de connaître le comportement global du circuit analysé.

Les cas particuliers d'identification tels que les éléments séquentiels (latches ou bascules) sont résolus par l'application de techniques de reconnaissance de forme sur les structures concernées, et ceci chaque fois qu'une boucle est détectée dans le réseau de cônes. Les modèles des registres à reconnaître sont ainsi explicitement décrits dans l'application elle même.

La méthode est rapide et a prouvé son efficacité sur de nombreux circuits industriels chez Bull. Les limites du procédé sont relatives au nombre important de représentations possibles des latches et registres qui doivent être décrits explicitement dans l'application elle même pour être reconnus, mais aussi à sa limitation à la seule technologie CMos, et aussi au fait de ne pas pouvoir traiter de façon systématique les circuits qui ont un comportement de type analogique, comme par exemple les amplificateurs différentiels des mémoires RAM.

3.2 Les logiciels utilisant la reconnaissance de formes

3.2.1 VERA

Vera qui a été développé par la société *Philips* [Vera91] est un programme de vérification qui procède par reconstruction et qui pour cela met en oeuvre une approche de type reconnaissance de forme. Cet outil a pour rôle de reconnaître des structures prédéfinies qui peuvent être elles mêmes agencées selon un mode hiérarchique. *Vera* vérifie donc la conformité d'une réalisation par rapport à une spécification de structure. Ce procédé permet de détecter et de localiser les erreurs de conception.

Vera travaille à partir de modèles de structures de base. *Vera* reconnaît certains modes d'assemblage de ces structures de base qui sont décrits par des règles de type règles de production (*Si condition Alors action*). Une primitive '*Recognize*' présente dans la partie *Condition* permet de faire référence à un modèle instancié. Une primitive *Abstract* présente dans la partie *Action* de la règle permet de regrouper plusieurs modèles instanciés reconnus en une description de niveau hiérarchique supérieur.

Vera distingue quatre classes (0 à 3) de reconnaissance des blocs fonctionnels:

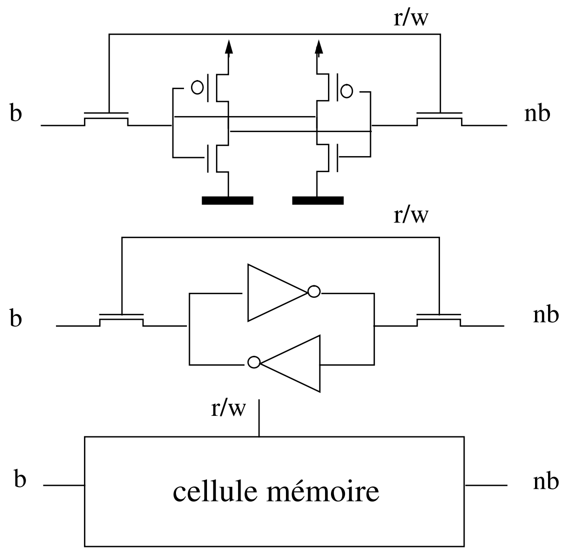
La classe 0 est celle des modules non paramétrés. Ce sont par exemple les composants des bibliothèques de cellules précaractérisées. Le mode d'abstraction de la classe 0 met en oeuvre un mécanisme d'abstraction hiérarchique qui substitue progressivement aux configurations reconnues des éléments de plus haut niveau. Un exemple de circuit qui peut être ainsi reconnu est celui qui est ici décrit dans la Figure 3-5.

La classe 1 exprime la reconnaissance de structures répétitive de type série ou parallèle et elle s'effectue à l'aide de deux primitives qui sont respectivement *Chain* et *Fork*.

La classe 2 consiste à combiner ces deux formes de répétitivité pour reconnaître des structures matricielles à fonctionnalité constante. C'est ce qui permet par exemple d'exprimer l'organisation en lignes de mots des registres d'une mémoire RAM. Cette classe ne met pas en oeuvre d'autre opérateur que ceux de la classe 1.

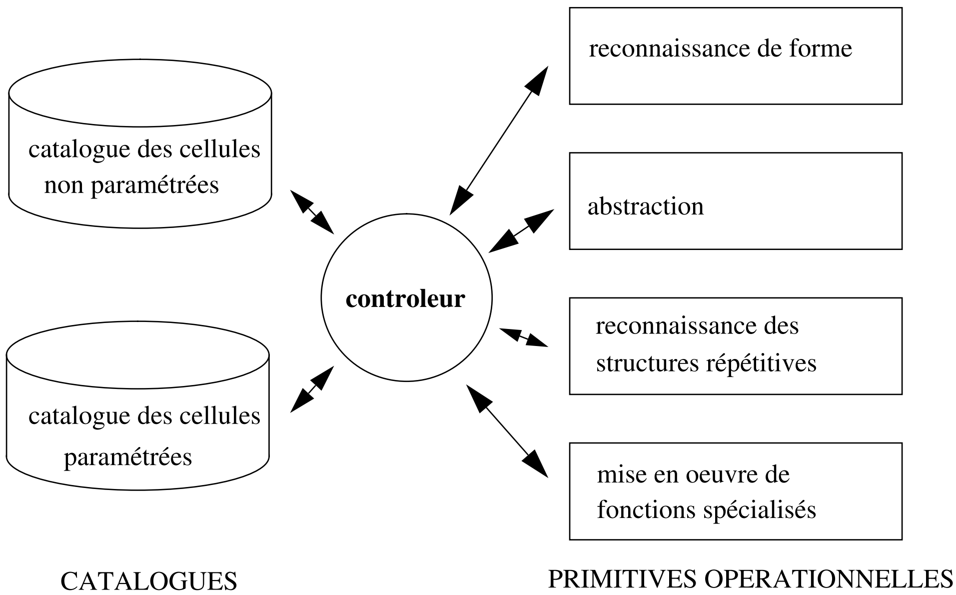
La classe 3 permet de reconnaître des structures paramétrées ayant une fonctionnalité variable telles que les ROM et les PLA. Cette dernière classe fait appel à des fonctions spécialisées.

Figure 3-5 exemple de reconnaissance de cellule mémoire non paramétrée



Les opérations associées à ces classes s'articulent au travers d'un modèle opérationnel qui est donné dans la Figure 3-6 suivante:

Figure 3-6 modèle opérationnel du reconstructeur *Vera*



Un contrôleur est créé pour chaque générateur de module, et c'est ce contrôleur qui décrit comment le module doit être reconstruit. *Vera* est donc associé à des compilateurs de silicium tels que *Cathedral* ou des environnements de conception intégrés tels que *Piramid*.

Le rôle de *Vera* n'est pas de générer un comportement à partir de n'importe quelle *net-list* de transistors. Vera reconstitue seulement la hiérarchie précisement et préalablement décrite d'un composant déterminé pour vérifier que la structure hiérarchique qui est remontée par abstraction est bien identique à celle qui était définie dans la phase de conception. *Vera* est donc essentiellement un outil de vérification par reconstruction de l'assemblage de macrocellules produites par un générateur paramétrable (ROM,RAM,...), la reconstruction se faisant à partir de l'ensemble de directives de reconstruction qui est spécifique au seul module traité. *Vera* ne sait pas traiter une description qui serait extérieure aux environnements de conception *Cathedral* et *Piramid*.

Un aspect intéressant du système est celui qui consiste à utiliser des catalogues externes pour décrire les modèles et l'agencement hiérarchique de ces modèles, bien que lors de la reconnaissance de structures répétitives, il soit nécessaire de connaître à priori le nombre exact d'éléments qui décrivent le composant. Cette limitation interdit par exemple de décrire précisement de façon générique un multiplexeur dont le nombre d'entrées serait variable selon les implémentations. Les alternatives de reconstruction (opérateur *ou*) ne sont pas non plus permises.

On retiendra l'intérêt de l'approche hiérarchique, celui de son contrôle par un langage de règles externes, et la possibilité de reconnaître aussi bien des circuits CMos que des circuits bipolaires et analogiques. On reprochera essentiellement à cette application son manque de souplesse puisqu'il ne peut pas s'appliquer à des composants dont le contenu est à priori inconnu. Ce reconstructeur doit s'intégrer obligatoirement dans une méthodologie de conception spécifique (Cathedral et Piramid). On lui reprochera aussi de ne pas avoir su exploiter les possibilités qu'offre l'abstraction structurelle hiérarchique pour générer une description comportementale pour faire aussi de l'abstraction fonctionnelle. Enfin, la mise en oeuvre des mécanismes de reconstruction est assez lourde et demanderait à être unifiée, puisqu'elle nécessite de mettre en oeuvre explicitement pas moins de 4 classes de fonctions distinctes pour pouvoir reconnaître certains composants tels que ceux de type mémoire.

3.2.2 DESA

Le programme *Desa* [Desa90] est le premier outil d'abstraction développé chez Bull. Il a été utilisé avec succès pendant plusieurs années.

Desa met en oeuvre des techniques de reconnaissance de formes associées à l'exploitation de modèles de composants élémentaires. Ces modèles de composants sont rassemblés dans une bibliothèque qui est compilée avec le programme.

Le mode de travail de *Desa* consiste à reconnaître des sous ensembles connexes de transistors qui satisfont un modèle présenté jusqu'à ce qu'il ne reste plus de transistors pouvant être filtrées par le modèle présenté. Le programme traite ensuite le modèle suivant, les modèles étant présentés dans un ordre pré-défini.

Les modèles simples sont donnés en priorité, car cela permet de délester la base de donnée et d'optimiser ainsi le temps de traitement. Les circuits plus complexes sont traités ensuite, en ré-utilisant le cas échéant les résultats précédents. Par exemple, un latch utilisera des inverseurs précédemment reconnus.

Le principal avantage de la méthode est sa puissance, car *Desa* sait reconnaître n'importe quelle structure de circuit électronique, comme par exemple les portes logiques élémentaires de type *nand, nor, xor*, mais aussi les circuits à nombre variable d'éléments tels que les multiplexeurs et les circuits à précharge. *Desa* reconnaît également les amplificateurs différentiels utilisés dans les mémoires RAM.

Comme *Desa* réalise une partition stricte du circuit à traiter, un transistor ne peut appartenir qu'à un seul composant à la fois. Tous les transistors donnés en entrée de *Desa* doivent être reconnus comme appartenant à un circuit modélisé, et s'il reste des transistors non associés à des composants reconnus en fin de traitement, alors ces transistors sont significatifs d'erreurs dans la connectique analysée.

Cependant, *Desa* n'est pas un outil d'abstraction fonctionnelle, et son travail consiste seulement à produire une *net-list* de *portes* interconnectées à partir de la *net-list* de transistors initiale.

Desa est écrit en langage *Prolog*, et les règles qui décrivent les modèles sont décrites sous la forme de prédicats. Le formalisme de codage des règles étant relativement complexe, ceci constitue un fort handicap si l'on souhaite pouvoir rendre accessible à des non informaticiens la construction et la gestion des modèles de circuits. L'imbrication étroite entre le programme et les descriptions des modèles ont rendu là aussi cet outil non exportable en dehors de son contexte industriel d'origine.

3.2.3 DIALOG

Dialog [Dialog89] est un Système Expert qui a été développé à l'IMEC (Centre Interuniversitaire de Micro-Electronique de Louvain) pour d'effectuer la vérification électrique des composants VLSI en technologie Mos.

Le rôle de *Dialog* est d'assister les concepteurs pour les aider à évaluer les conditions de bon fonctionnement électrique des circuits logiques synchrones. Les techniques du type système expert permettent de définir et vérifier des règles topologiques et électriques.

Le système expert est bâti au dessus du langage *Lisp* et il a la particularité d'être mixte, car il combine deux procédés qui sont d'une part l'analyse du circuit pilotée par un système de règles, et d'autre part l'exécution de fonctions programmées en langage *Lisp*.

Le langage de règles a pour nom *lextoc* (Language EXpressing TOpology Constraints). Les règles de *lextoc* sont du type règles de production, ce qui s'exprime par:

```
if P1 and ... Pn then do A1 and ... Am
```

où la séquence P1 à Pn représente une conjonction de conditions. Notons ici que dans la partie condition, il n'y a pas de disjonction possible, et que l'ordre des conditions n'importe pas. La séquence A1 à Am représente une suite d'actions à réaliser dès lors que l'ensemble de conditions s'évalue à vrai. Notons ici aussi que dans la partie action, chaque action retourne toujours vrai, et que l'opérateur *and* sert tout simplement à enchaîner les actions.

Les règles de *Dialog* incorporent des variables dites de filtrage. De telles règles peuvent être hiérarchiques, et elles peuvent aussi activer des fonctions Lisp. C'est ainsi que:

- les règles servent essentiellement à décrire des structures à reconnaître, et à identifier des chemins de connection. Elles permettent aussi d'extraire des informations spécifiques à chaque style de circuit pour identifier par exemple les horloges et les noeuds mémorisants. Ces mêmes règles permettent encore de dériver l'orientation des signaux et d'analyser les boucles de contre réaction asynchrones.
- les fonctions Lisp peuvent être utilisées dans les règles aussi bien pour créer des conditions évaluables, afin d'effectuer des calculs et afin de parcourir des structures, que pour réaliser les actions.

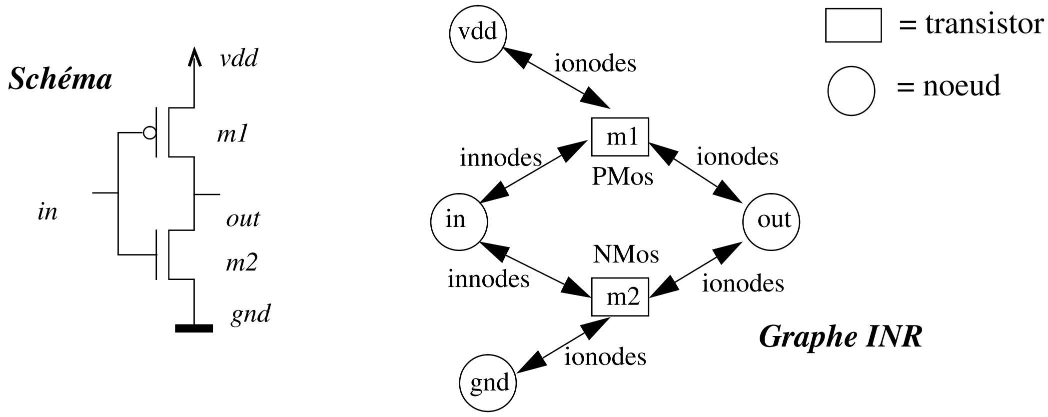
Dialog opère sur une représentation des données appelée "Représentation Invariante du *Réseau*" (INR). Cette représentation définit des objets avec leurs *attributs* (association nom et valeur), leurs *propriétés* (attribut avec valeur booléenne), et leurs *relations* (dépendances uni-directionnelles entre objets). A partir de là, le *Réseau* est vu comme étant constitué de blocs de base appelés *éléments*. Les transistors, résistances et capacités constituent des éléments *terminaux*, les équipotentielles des *noeuds*, et les *connecteurs* réalisent l'interface de chaque élément.

La connection d'un *noeud* au drain ou à la source d'un transistor crée une relation bidirectionnelle de type *ionodes* entre le *noeud* et le transistor. Un *noeud* connecté à la grille d'un transistor crée une relation *innodes* entre le noeud et le transistor.

Les règles travaillent à partir de modèles des structures de données de la représentation invariante précitée.

C'est ainsi par exemple que la Figure 3-7 montre comment une règle inverseur modélisée à partir d'une représentation en format INR, ici représentée par son graphe, permet de reconnaître un inverseur à partir de la conjonction (opérateur *and* implicite) des conditions de type *PROPERTY* (pour obtenir des valeurs d'arguments) et des relations de type *RELATION*. Cette règle crée par l'action *ABSTRACT* une représentation abstraite de plus haut niveau de l'inverseur.

Figure 3-7 modélisation d'un inverseur dans une règle DIALOG



Règle

```
(DEFRULE define-invertor ()
(*IF
   ((PROPERTY pmos m1)
   (RELATION ionodes m1 vdd)
   (RELATION ionodes out e1)
   (PROPERTY nmos m2)
   (RELATION ionodes m2 out)
   (RELATION ionodes gnd m2)
   (RELATION innodes in m1)
   (RELATION innodes in m2))
*THEN
   ((ABSTRACT (invert m1 m2) enew)
   (ASSIGN-INFO invertor enew))))
```

De façon non exhaustive, nous pouvons citer quelques propriétés des règles de *Dialog* qui: peuvent être hiérarchiques, c'est à dire s'activer les unes et les autres en échangeant des paramètres; peuvent appeler des fonctions *Lisp*; peuvent réaliser des contrôles de tautologie sur des expressions fonctionnelles; supportent l'opérateur *not* qui peut être appliqué à une condition ou à une valeur d'argument, et l'opérateur *or bien que* limité aux seules valeurs d'arguments; peuvent déclarer des relations entre objets; peuvent construire et détruire les objets; peuvent chercher l'appartenance d'expressions dans des listes; peuvent émettre des messages d'erreur et d'avertissement; peuvent exécuter des simulations locales,...

Les règles sont activées automatiquement par les mécanismes d'inférence en chaînage arrière quand c'est nécessaire. Cependant, un tel mécanisme de contrôle des règles peut être difficile à appréhender, et de plus, l'ordre d'exécution des règles influe sur le résultat final, ce qui peut rendre le système indéterministe. C'est pourquoi, lors de l'activation d'une règle, la stratégie employée par *Dialog* consiste à chercher d'abord toutes les instances qui sont filtrées par la partie condition de la règle et à les enregistrer. Ce n'est qu'après que toutes les instances ont été trouvées que les actions correspondantes sont exécutées pour chacune d'entre elle (l'autre alternative aurait été de les exécuter immédiatement). Il est à noter ici cependant que malgré cela, la stratégie qui a été retenue laisse subsister des cas de non déterminisme difficilement contrôlables.

La méthode de vérification de *Dialog* consiste à exploiter les règles de construction et de comportement de circuits dits *Intentionnellement* corrects pour contrôler que les structures et les comportements de ces circuits sont effectivement conformes à ces règles prédéfinies. Toute erreur de conception sera détectée lors de la vérification parce que invalidant l'une de ces règles.

Dialog permet aussi d'associer des techniques d'analyse logique aux vérifications locales des structures. Pour cela, le circuit analysé est préalablement décomposé en structures de moindre complexité, lesquelles sont ensuite modélisées en unités fonctionnelles qui doivent être obligatoirement synchrones. C'est à partir de ces unités fonctionnelles que peuvent être identifiées les mémoires de type latch ou précharge et les horloges qui leur sont associées. Il est à noter ici que l'analyse logique est uniquement locale, et que *Dialog* ne remonte pas le comportement global du bloc analysé.

Enfin, ce logiciel permet de traiter des circuits de l'ordre de 10000 transistors.

On retiendra que *Dialog* met en oeuvre une méthode originale de vérification qui applique des règles formelles de vérification en associant des techniques informatiques de type Système Expert à de la programmation fonctionnelle. On peut regretter là aussi, que les potentiels de la méthode n'aient pas été suffisamment exploités pour effectuer une abstraction fonctionnelle complète des blocs de données analysés. Le principal atout de cette approche mixte est cependant sa souplesse d'adaptation. Le principal inconvénient est que le contrôle de l'application est complexe, et qu'il reste réservé à des informaticiens à la fois spécialistes de la programmation en langage *Lisp* et initiés aux techniques des Systèmes Experts.

3.3 Conclusion

Nous venons de voir deux approches différentes pour réaliser l'abstraction des circuits électroniques: les méthodes d'analyse de réseaux de transistors et les méthodes de reconnaissance de formes.

Les méthodes d'analyse de réseaux sont en pratique limitées à la technologie CMos, et le plus souvent spécialisées pour traiter certains styles de circuits.

Les méthodes dites de reconnaissance de formes ont jusqu'à présent été handicapées par leur manque de souplesse. Les outils existants mettant en oeuvre des techniques dites de type *Intelligence Artificielle* sont jusqu'à maintenant complexes à configurer, et pratiquement dédiés à un seul environnement industriel.

Cependant, si l'on veut à la fois généraliser l'abstraction à des technologies multiples, et en même temps la rendre paramétrable en fonction du contexte industriel, la méthode de reconnaissance de formes nous semble être la voie potentiellement la plus prometteuse. En effet, une telle méthode, qui est par définition configurable, est la seule qui soit à même de pouvoir prendre en compte toutes les variantes et évolutions technologiques actuelles et à venir.

Le problème qui reste à résoudre consiste à définir un langage de description des modèles de circuits à reconnaître qui soit à la fois souple et simple à appréhender par des programmeurs non experts.

A partir de là, le fait de rendre externes au programme les descriptions de modèles permettra non seulement à l'outil de cibler précisement n'importe quelle technologie ou méthodologie de conception, mais cela permettra aussi d'exporter l'outil dans d'autres environnements industriels, et aussi de le maintenir et de le faire évoluer facilement.

Il faudra enfin que ce langage soit à même de produire directement une description fonctionnelle. Nous avons pu voir en effet que les applications de l'abstraction basées sur des techniques de reconnaissance de formes étaient surtout orientées vers la reconnaissance de structures de circuits et vers l'analyse électrique de ces circuits. A ce jour aucun développement particulier n'a été effectué à partir des techniques de reconnaissance de formes pour traiter de façon efficace l'abstraction fonctionnelle. C'est donc aussi cette voie que nous allons nous proposer d'explorer dans cette étude.

4. Présentation de la méthode

Nous avons été amenés pour satisfaire les objectifs que nous venons de définir dans le précédent chapitre à retenir une technique de **reconnaissance de formes**, laquelle nécessite donc la définition d'une bibliothèque de schémas prédéfinis.

Pour que l'outil puisse être configuré facilement par les concepteurs de circuits électroniques, nous avons choisi de rendre externes au programme ces descriptions de formes à reconnaître. Les **modèles** de circuits seront ainsi décrits à l'aide d'un langage dédié appelé langage GRL.

Ce chapitre décrit ainsi les objets sur lesquels travaille l'abstraction, la méthode employée et le langage GRL.

4.1 Les données d'entrée

4.1.1 La *net-list* de transistors

Pour réaliser l'abstraction nous partons d'une *net-list* décrivant le circuit à abstraire. C'est à partir de cette *net-list* qu'est construite la structure de données qui est utilisée par le programme d'abstraction.

La *net-list* exprime une connectique en terme:

- **de composants élémentaires:** ce sont les transistors, les diodes, les résistances.
- **de connecteurs:** ce sont les éléments qui décrivent l'interface du circuit (connecteurs externes) et l'interface des composants (connecteurs internes).
- **de signaux:** ce sont les connections entre les connecteurs des composants qui véhiculent les signaux électriques. Ici, on assimilera la connection et le signal qu'elle véhicule en désignant la connection sous le vocable de ***signal*** et en lui donnant le nom de ce signal.

Un circuit VLSI complexe est habituellement construit selon une approche hiérarchique qui résulte en un assemblage de blocs. La *net-list* à traiter représente généralement un de ces blocs, et peut contenir plusieurs dizaines de milliers de transistors.

La *net-list* est extraite au moyen d'un extracteur à partir d'une description de masques par exemple dans un format *CIF* ou *GDS2*. Les techniques d'extraction des *net-list* sont aujourd'hui bien connues et nous ne les détaillerons pas, de nombreux logiciels commerciaux permettant d'extraire une *net-list* de composants élémentaires à partir du dessin des masques d'un bloc.

Chaque format de *net-list* externe doit être converti en un fichier intermédiaire de format *ascii* appelé fichier ***GND*** qui est directement interprété par l'abstracteur. Ce format est spécifié en “ANNEXE A“.

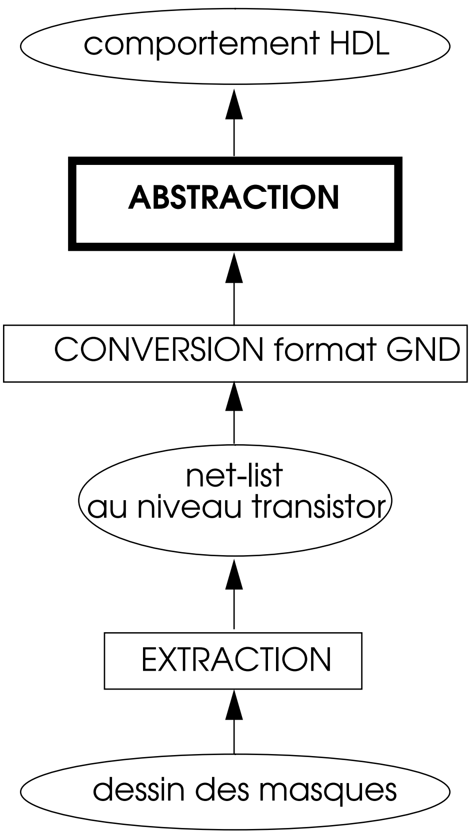
Nous pouvons en particulier convertir en format *GND* deux formats de *net-list* très répandus qui sont le format ***fne*** de la société Compass [CompNetl92], [Vti], et le format ***cdl*** de la société *Cadence* [CadRm91].

4.1.2 Le flot de données

Le flot de données part de la description physique des masques, pour aboutir après les étapes successives d'extraction, de conversion de format, et d'abstraction au descriptif comportemental de type **flot de données** qui est décrit dans des langages dits HDL (*Hardware Description Langage*). Dans notre application, les langages HDL qui sont produits sont d'une part le langage **Vhdl** [VhdlIeee87] et d'autre part le langage **Verilog** [VerilogIeee95].

C'est ce flot de données qui est décrit dans la Figure 4-1 suivante.

Figure 4-1 le flot de données



4.2 La modélisation des objets de l’abstraction

Nous allons d’abord décrire les **objets** qui sont nécessaires pour modéliser la *net-list*, puis donner un exemple de l’organisation de ces objets dans la *net-list*.

4.2.1 Les composants élémentaires

Les composants élémentaires de la *net-list* sont appelés les **primitives**. Le terme générique de primitive décrit ainsi des objets élémentaires qui ne résultent pas de l’assemblage d’autres objets.

Les primitives sont des **instances** des composants élémentaires suivants:

- les transistors Mos N et P à enrichissement, et Mos N à déplétion.
- les transistors bipolaires NPN et PNP.
- les diodes
- les résistances

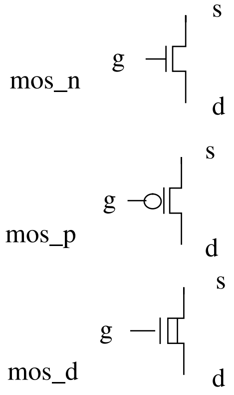
Nous allons maintenant décrire la sémantique de ces primitives comme suit:

Transistors Mos

Les transistors Mos représentés en Figure 4-2 sont caractérisée par les champs suivants:

- *identificateur* définit le type de transistor mos, à savoir *mos_n, mos_p*, *mos_d*.
- *largeur* et *longueur* contiennent les largeur et longueur de grille.
- *abscisse* et *ordonnée* contiennent les coordonnées du transistor.
- *grille, drain* et *source* contiennent les références aux signaux correspondants.

Figure 4-2 sémantique des transistors Mos



transistor mos	identificateur
	largeur de grille
	longueur de grille
	abscisse
	ordonnée
	grille
	drain
	source

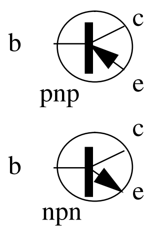
Transistors bipolaires

Les transistors bipolaires représentés en Figure 4-3 sont caractérisés par les champs suivants:

- *identificateur* contient un nom qui donne le type du transistor bipolaire, à savoir *npn* ou *pnp*.
- *abscisse* et *ordonnée* contiennent les coordonnées du transistor lors de son placement.

- *base*, *émetteur*, *collecteur* contiennent les références aux signaux correspondants.
- *surface_emetteur* contient la surface de l'émetteur

Figure 4-3 sémantique des transistors Bipolaires



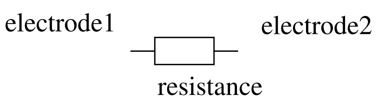
transistor bipolaire	identificateur
	abscisse
	ordonnée
	base
	émetteur
	collecteur
	surface_émetteur

Dipôles résistance

Les dipôles résistance qui sont décrits Figure 4-4 sont caractérisés par les champs suivants:

- *électrode1* et *électrode2* contiennent les références aux signaux d'accès à la résistance
- *abscisse* et *ordonnée* contiennent les coordonnées de la résistance lors de son placement.

Figure 4-4 sémantique des résistances



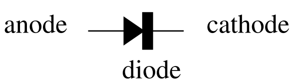
résistance	valeur_résistive
	abscisse
	ordonnée
	électrode1
	electrode2

Dipôles diode

Les dipôles diodes qui sont décrits en Figure 4-5 sont caractérisés par les champs suivants:

- *anode* et *cathode* contiennent les références aux signaux d'accès à la diode.
- *abscisse* et *ordonnée* contiennent les coordonnées de la diode lors de son placement.

Figure 4-5 sémantique des diodes



diode	abscisse
	ordonnée
	anode
	cathode

Remarque: On voit que chaque primitive possède deux types d’attributs qui sont:

- des connecteurs (3 pour les transistors, 2 pour les résistances et les diodes) qui permettent de décrire la connectique.
- des paramètres physiques qui caractérisent chaque instance d’un composant élémentaire.

4.2.2 Les Unités Abstraites

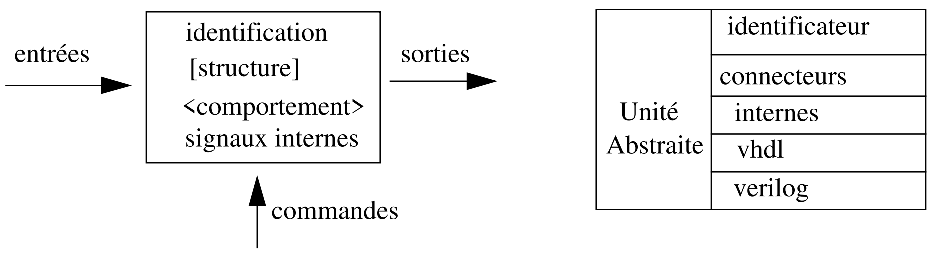
Les Unités Abstraites sont des instances de modèles reconnus par le programme d’abstraction. Les Unités Abstraites résultent donc de l’assemblage de primitives et/ou d’autres Unités Abstraites. Il est à noter que les Unités Abstraites, à la différence des primitives, ne possèdent pas de paramètres physiques intrinsèques.

Tout modèle doit impérativement correspondre à un schéma connexe. Il peut posséder des signaux internes, c’est à dire non connectés à un connecteur de l’interface du modèle.

L’Unité Abstraite est modélisée dans la Figure 4-6, et ses champs sont les suivants:

- *identificateur* contient un nom qui est le nom du modèle que l’on instancie.
- *connecteurs* contient la liste des références aux signaux qui réalisent l’interface de l’Unité Abstraite.
- *interne* contient la liste des références aux signaux internes à l’Unité Abstraite.
- *vhdl* et *verilog* reçoivent respectivement les descriptifs des comportements Vhdl et Verilog.

Figure 4-6 sémantique de l’objet Unité Abstraite



Remarque:

1. Tandis qu’une primitive a un nombre fixe de connecteurs, une Unité Abstraite en a un nombre variable. Une Unité Abstraite n’a donc pas de connecteur prédéfini comme cela était le cas par exemple pour les transistors Mos avec les connecteurs de grille, drain et source, et c’est donc la position de chaque signal dans la liste *connecteurs* qui sert à spécifier à quel connecteur de l’Unité Abstraite concernée ce signal est connecté.
2. On sera amené par la suite pour des raisons techniques et d’ergonomie à distinguer plus précisement les types de connecteurs de l’Unité Abstraite en tant que connecteurs d’entrée, de commande et de sortie; une telle distinction permettra essentiellement d’obtenir des informations sur l’orientation des signaux, informations utiles pour produire les comportements Vhdl et Verilog.

4.2.3 Les composants

On désignera le plus souvent les primitives et les Unités Abstraites sous le terme générique de **composants**.

Remarque: Pour alléger les descriptions nous n'avons pas fait figurer le champ booléen *utilisé* qui est commun à tous les composants, et dont le rôle est de marquer le composant lorsque celui ci à été utilisé pour construire une Unité Abstraite. Initialement toutes les primitives ont leur champ *utilisé* marqué à *Faux*.

4.2.4 Les signaux

On distinguera:

- les **signaux internes** qui n'ont pas de visibilité externe au bloc.
- les **signaux externes** qui ont une visibilité externe, parce qu'ils sont connectés à un connecteur externe.

Un objet de type signal est modélisé comme en Figure 4-7 et possède les champs suivants:

- *type* dont la valeur est *interne* ou *externe* (ce qui décrit la visibilité dans le bloc).
- *nom* qui contient le nom complet du signal.
- *racine* qui est le nom sans l'index.
- *index* qui est le rang du signal dans le vecteur de signaux.
- *composants* qui contient la liste des composants auxquels ce signal est connecté.
- *tag* qui est appelé à contenir un des indicateurs *latch*, *register*, *tristate*, *precharge, null* qui servent aux éditions HDL et aux contrôles de cohérence sur les données.
- *vh_type* qui permet de positionner le type vhdl du signal concerné à une valeur différente du type par défaut. Le type par défaut est le type *std_logic* du paquetage IEEE Std 1164, à moins qu'il ne soit redéfini sur la ligne de commande du programme. C'est ainsi par exemple que le champ *vh_type* d'un signal peut être affecté au besoin à *std_logic* si le type par défaut est *std_ulogic*.

Figure 4-7 modèle de l'objet signal

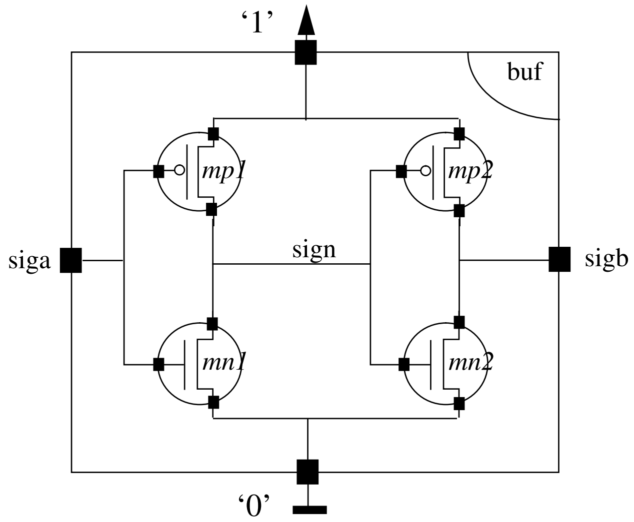
	type
	nom
	racine
signal	index
	composants
	tag
	vh_type

4.2.5 Exemple

Nous allons voir maintenant par un exemple simple comment les objets que nous avons spécifiés peuvent être agencés pour décrire un circuit.

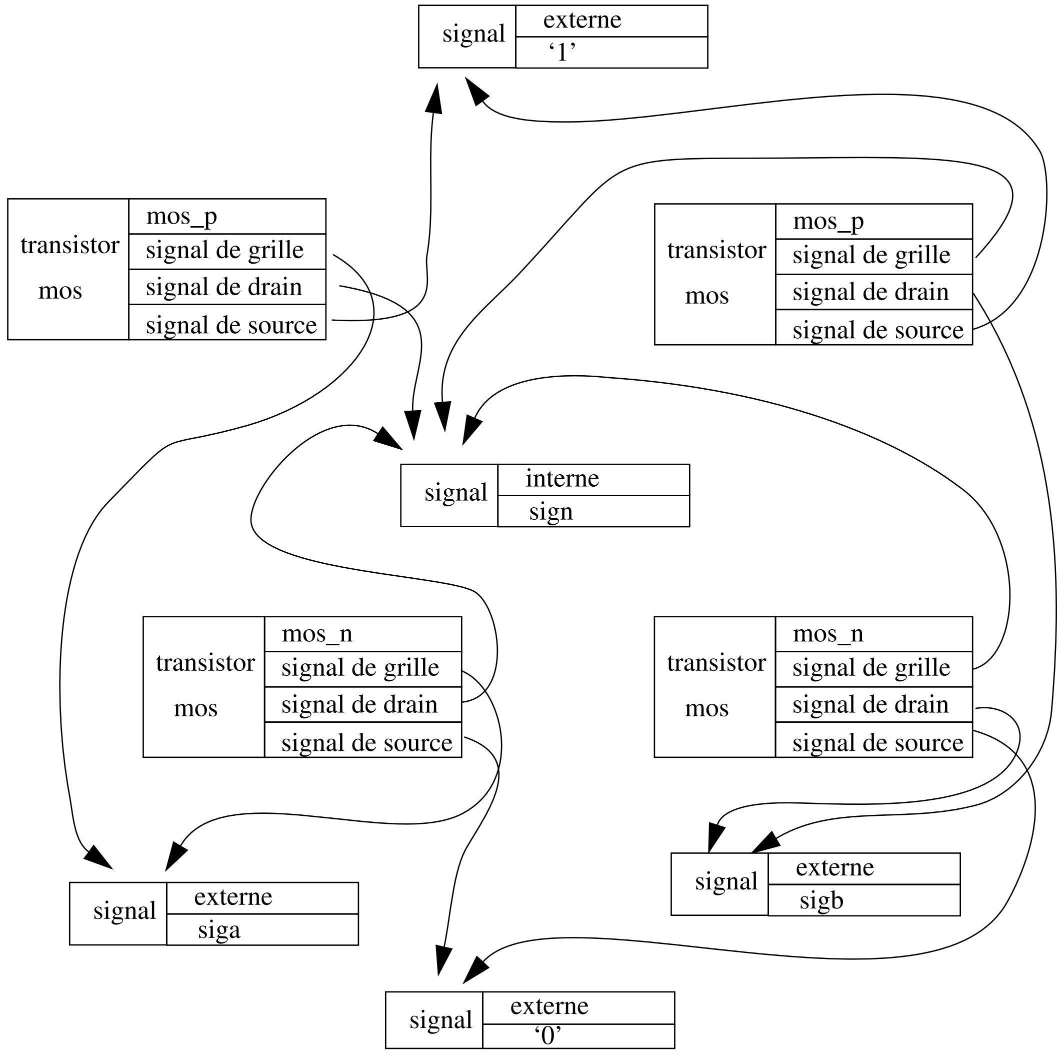
L'exemple de la Figure 4-8 décrit au niveau transistor la *net-list* d'un circuit dit *buffer* contenant 4 transistors. Dans cet exemple, *siga* est le signal d'entrée, *sigb* est le signal de sortie et *sign* un signal interne, *'0'* et *'1'* sont les signaux d'alimentation. Les signaux *siga*, *sigb* sont associés à des connecteurs externes du circuit *buffer*. Chaque transistor possède des connecteurs de *grille*, *drain* et *source.*

Figure 4-8 schéma de la *net-list* d'un circuit *buffer* constitué de 4 transistors



Nous pouvons voir en Figure 4-9 l'organisation des objets qui est issue d'une telle description. Les descriptions d'objets sont ici simplifiées dans un soucis de concision.

Figure 4-9 exemple simplifié d'organisation des objets de la *net-list* du circuit *buffer*



4.3 L'abstraction

4.3.1 Principes

On a vu que l'on dispose initialement d'une *net-list* de primitives non structurées. Le procédé d'abstraction consistera à:

- reconnaître dans la *net-list* des modèles constitués par un assemblage particulier de primitives ou d'Unités Abstraites.
- créer une Unité Abstraite pour chaque modèle reconnu, qui se substitue aux primitives et Unités Abstraites qui composent ce modèle.
- associer à chaque Unité Abstraite nouvellement créée un descriptif comportemental qui traduise le fonctionnement du modèle reconnu.

En fin de traitement, l'ensemble des descriptifs comportementaux des Unités Abstraites construites constitue le comportement global du circuit traité.

Il est à noter que l'abstraction est un processus **ré-entrant** puisqu'elle consiste à construire des Unités Abstraites non seulement à partir de primitives, mais aussi à partir d'autres Unités Abstraites. C'est la ré-entrance du processus d'abstraction qui permet:

- de décrire des modèles complexes de façon hiérarchique.
- de traiter simplement tous les modèles qui se composent d'un nombre variable de composants identiques puisque ceux ci peuvent être construits en mettant oeuvre un mode de composition cumulatif.

Ces deux points particuliers seront détaillés dans la suite de ce chapitre.

Il ne doit plus rester de primitives non intégrées dans une Unité Abstraite en fin d'abstraction. Les primitives qui peuvent éventuellement subsister en fin de traitement sont significatives de cas d'erreur.

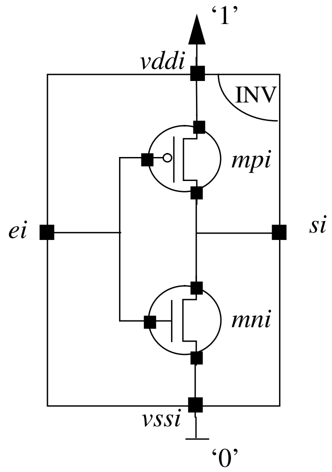
Remarque: La construction d'une Unité Abstraite est un mécanisme d'abstraction. Ceci signifie qu'une fois l'Unité Abstraite construite, on ne garde pas la trace des composants qui la constituent.

4.3.2 Méthode de reconnaissance des circuits

Nous allons illustrer la méthode sur l'exemple précédent de la Figure 4-8 qui décrivait la schéma de la net-list d'un circuit buffer constitué de 4 transistors. A partir de là nous allons décrire comment ce circuit va être structuré par les applications successives d'un modèle d'inverseur.

Le modèle d'inverseur est décrit dans la Figure 4-10 où *mpi* et *mni* sont 2 transistors Mos N et P, *ei, si, vssi, vddi* sont les connecteurs du modèle, et *INV* est le nom du modèle.

Figure 4-10 schéma du modèle d'inverseur



La reconnaissance va s'effectuer en deux temps:

1. Dans un premier temps, le modèle de l'inverseur *INV* de la Figure 4-10 est appliqué sur la *net-list* de la Figure 4-8.

   En appliquant ce modèle sur la *net-list*, on cherche à vérifier les contraintes suivantes:

- le transistor Mos N noté *mni* du modèle doit correspondre avec un transistor de même type de la *net-list* à traiter -> on choisit *mn1*,
- le transistor Mos P noté *mpi* du modèle doit correspondre avec un transistor de même type -> on choisit *mp1*,
- le connecteur de drain du transistor N doit être connecté au même signal que le connecteur de drain du transistor P -> le signal *sign* respecte la contrainte,
- le connecteur de grille du transistor N doit être connecté au même signal que le connecteur de grille du transistor P -> le signal *siga* respecte la contrainte,
- le connecteur de source du transistor N doit être connecté au signal de valeur imposée ‘0’ -> c’est le cas,
- le connecteur de source du transistor P doit être connecté au signal de valeur imposée ‘1’ -> c’est le cas.

Toutes les contraintes sont satisfaites. On peut donc remplacer les 2 transistors *mn1* et *mp1* par l’Unité Abstraite *inv* en connectant le signal *siga* au connecteur *ei* de l’inverseur et le signal *sign* au connecteur *si* de l’inverseur.

Le nouveau schéma du circuit devient celui de la Figure 4-11. L’état de la *net-list* est modifié en conséquence, et cette *net-list* qui est décrite en Figure 4-12 est maintenant composée de la nouvelle Unité Abstraite *inv1* qui vient d’être créée et des deux transistors Mos N et P *mn2* et *mp2* restant.

Figure 4-11 schéma du circuit après la reconnaissance du premier inverseur

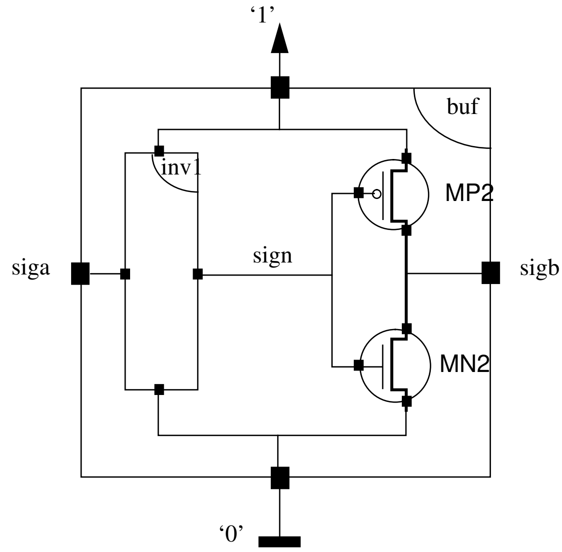
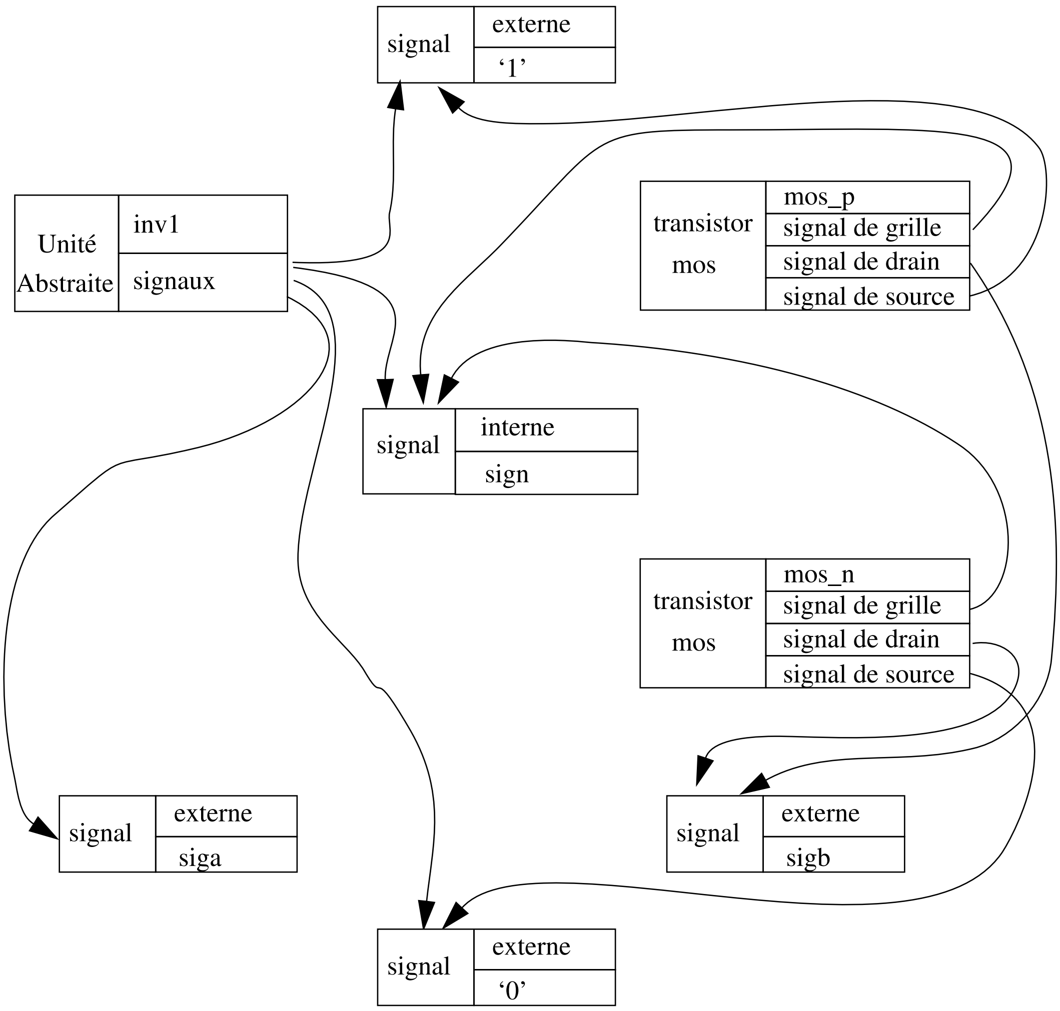


Figure 4-12 organisation de la *net-list* après la reconnaissance du premier inverseur



2. Dans un deuxième temps, on applique à nouveau le modèle de l'inverseur *INV* de la Figure 4-10 dont le schéma est décrit en Figure 4-11 et la 'net-list' est décrite en Figure 4-12 de façon à ce que le deuxième inverseur *inv2* puisse être reconnu.

Le nouveau schéma du circuit est maintenant celui de la Figure 4-13, et l'état correspondant de la *net-list* est représenté en Figure 4-14.

Figure 4-13 schéma de la *net-list* après la reconnaissance du deuxième inverseur

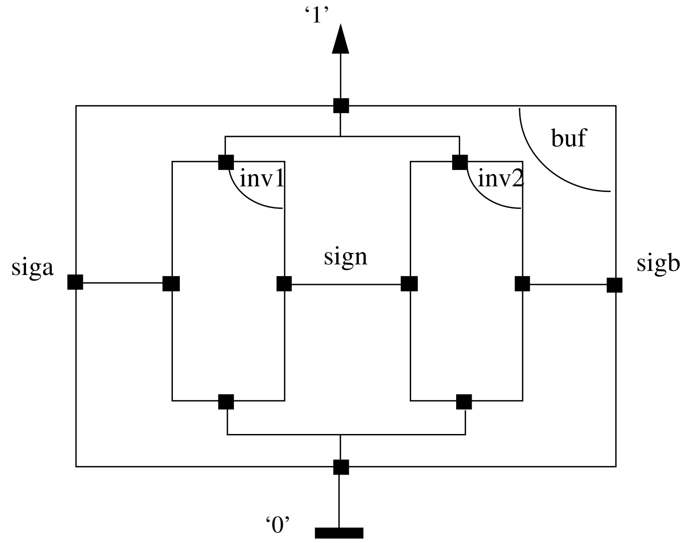
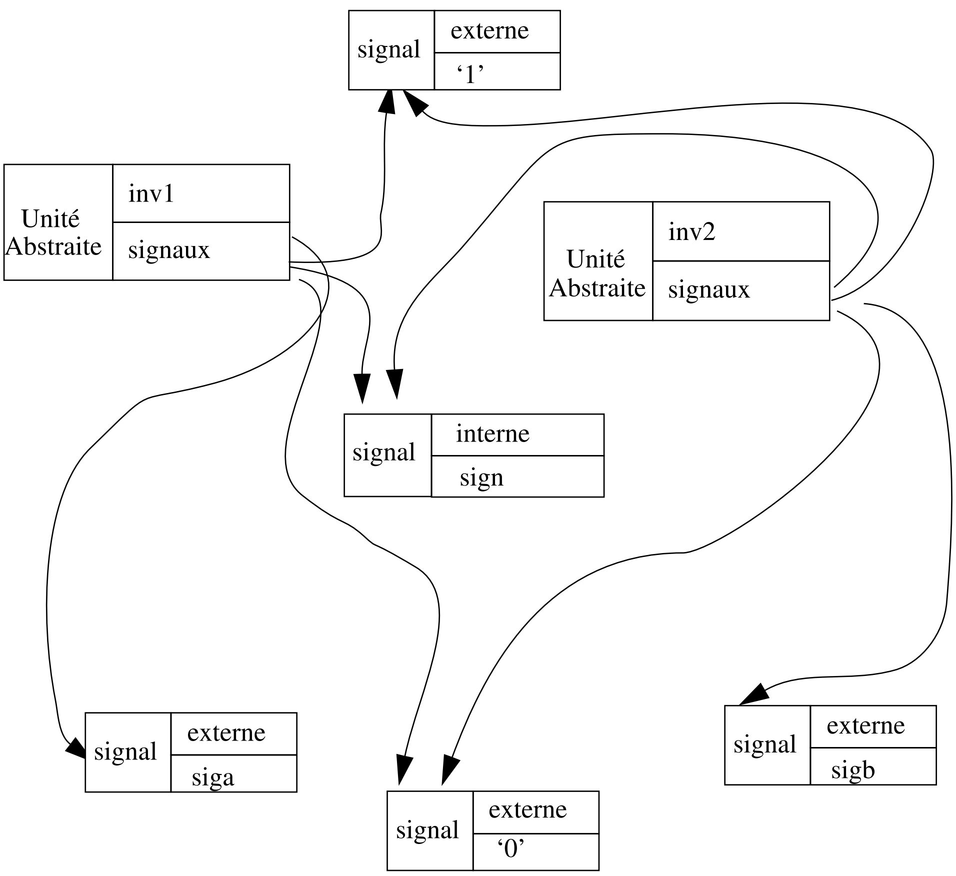


Figure 4-14 organisation de la *net-list* après la reconnaissance du deuxième inverseur



4.4 La description des modèles

Nous venons de voir que décrire un modèle consistait à exprimer un certain nombre de contraintes et de relations entre ses composants. Nous souhaitons maintenant exprimer la structure des modèles à reconnaître, et trouver pour cela un formalisme adéquat. Nous allons montrer comment ce qui vient d'être décrit dans un langage de *net-list* va pouvoir l'être aussi dans le langage des prédicats.

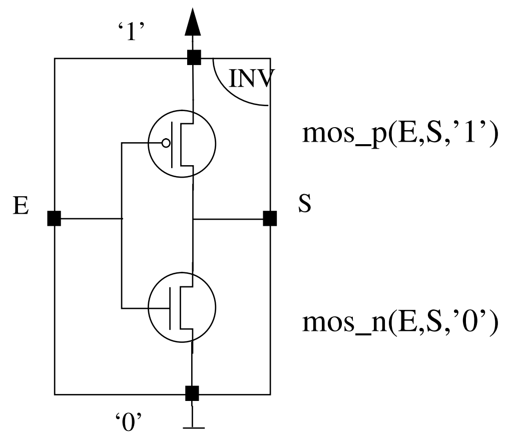
Il aurait été possible de décrire les modèles en utilisant un langage de description de *net-list* existant tel que le format EDIF [Edif87] ou bien le Vhdl structurel [VhdlIeee87], [VhdlMod90]. Nous avons préféré cependant spécifier un langage particulier pour pouvoir introduire tous les opérateurs spécifiques que nous estimons nécessaires dans le cadre de cette technique de reconnaissance.

Nous allons pour cela utiliser le formalisme des prédicats. Nous rappelons qu'un prédicat fait référence à des variables, et qu'il retourne ***Vrai*** ou ***Faux*** lors de son activation selon que celle ci réussit ou échoue. A partir de là, deux cas peuvent se présenter lors de l'activation d'un prédicat:

- si ses variables n'ont pas de valeur initiales, elles pourront se voir affecter une valeur si l'activation du prédicat réussit,
- si ses variables ont déjà reçu une valeur, alors cette valeur doit coïncider avec la valeur imposée par l'activation du prédicat, sinon l'activation échoue.

La Figure 4-15 décrit le modèle d'un inverseur et montre quels sont les prédicats qui sont associés aux transistors.

Figure 4-15 forme prédicative d'un modèle d'inverseur



Le prédicat décrivant un inverseur doit exprimer les 5 contraintes suivantes notées 1 à 5:

1. l'inverseur est composé de 2 primitives mos_n et mos_p,
2. les 2 primitives mos_n et mos_p doivent avoir le même signal noté E sur leur connecteurs de grille. La valeur de ce signal n'est pas contrainte, c'est donc une variable,
3. les 2 primitives mos_n et mos_p doivent avoir le même signal S sur leur connecteurs de drain. La valeur de ce signal n'est pas contrainte, c'est donc une variable,
4. la primitive mos_n doit avoir un signal de valeur constante *‘0’* sur son connecteur de source,

5. la primitive mos_p doit avoir un signal de valeur constante *'1'* sur son connecteur de source. La description d'un tel inverseur à l'aide de prédicats aura donc la forme suivante:

mos_n(E,S,'0') et mos_p(E,S,'1')

où *E* et *S* sont des variables, et *'0'* et *'1'* sont des constantes qui servent à représenter les signaux particuliers que sont les alimentations.

Dans cette dernière forme d'écriture, on voit que la position de chaque variable ou constante dans la liste des attributs de la primitive indique de quel connecteur il s'agit (ici un connecteur de grille, de source ou de drain), tandis que le nom de la variable identifie le signal concerné du modèle.

Cette écriture des prédicats est dans un premier temps simplifiée puisqu'elle ne fait apparaître que les attributs connecteurs et les contraintes de connectique dans les prédicats. Il est possible de complexifier les prédicats en introduisant des contraintes portant sur la valeur des paramètres physiques associés aux primitives. Par exemple, si on cherche à décrire le modèle d'un inverseur particulier dont les deux transistors possèdent les mêmes dimensions électriques, il faut ajouter 2 contraintes supplémentaires:

6. la largeur électrique de la primitive mos_n doit être égale à la largeur électrique de la primitive mos_p.
7. la longueur électrique de la primitive mos_n doit être égale à la longueur électrique de la primitive mos_p.

Si les 2 derniers paramètres des primitives mos_n et mos_p correspondent à la largeur et à la longueur électrique des transistors, le prédicat s'écrira donc:

mos_n(E,S,'0',W,L) et mos_p(E,S,'1',W,L)

ce qui exprime alors l'ensemble des 7 contraintes de 1 à 7 que nous venons de citer.

4.5 L'interprétation des modèles

4.5.1 Généralités

Les modèles que nous venons de décrire sous leur forme prédicative doivent être confrontés à la *net-list* de composants. Ceci afin d'obtenir un isomorphisme entre chacun des modèles et un ou plusieurs sous ensemble des composants qui constituent le circuit à reconnaître.

L'application d'un modèle est une opération complexe parce qu'elle nécessite d'effectuer de multiples essais avant d'arriver à une activation réussie. Nous allons voir que ceci se traduit par la gestion de retours arrières lors de l'application du modèle. Nous verrons aussi comment le modèle est initialisé grâce à une technique dite de point d'ancrage.

4.5.2 Les points d'ancrage

On ne saurait appliquer un mécanisme de filtrage systématiquement sur la totalité des composants présents dans la *net-list*, car une telle opération serait coûteuse en temps d'exécution. C'est pourquoi les sous ensembles de composants qui doivent être sélectionnés pour être confrontés au modèle seront initialement choisis d'après une relation particulière de voisinage autour d'un composant remarquable. C'est ce composant remarquable qui est appelé le **point d'ancrage,** et c'est lui qui permettra d'initialiser efficacement la recherche des éléments qui sont susceptibles de satisfaire le modèle en ne considérant que les sous ensembles plausibles. Cette technique de point d'ancrage est par ailleurs couramment utilisée dans le cadre de la reconnaissance structurelle de formes [RFormes85].

En ce qui nous concerne, c'est le **premier** des prédicats qui est cité dans le modèle qui définit le point d'ancrage, et il doit donc être choisi en connaissance de cause lors de l'écriture du modèle.

Il pourra être ainsi judicieux de commencer la recherche d'un circuit donné en s'ancrant sur un composant connecté à un signal d'alimentation '0' ou '1' puisque le nombre de transistors connectés à une alimentation représente un sous ensemble de moindre taille dans l'ensemble des transistors. L'organisation du modèle est laissée au choix de la personne qui écrit les modèles et qui devra faire en sorte que le premier objet cité dans chaque modèle constitue un bon point d'ancrage.

Les heuristiques qui conduisent au choix d'un bon point d'ancrage sont détaillées en ANNEXE C, dans le sous chapitre C.2.

4.5.3 Construction et évaluation de l'arbre des prédicats

Les prédicats constituant un modèle peuvent être structurés suivant un arbre binaire dont les noeuds sont les opérateurs *{et,ou},* et dont les feuilles sont les prédicats élémentaires. Cet arbre permet d'exprimer toutes les variations morphologiques permises du modèle recherché. Considérons par exemple en Figure 4-16 deux descriptions d'un élément de multiplexeur.

Nous voyons que ces deux modèles partagent une même description commune qui est constituée par les deux transistors N et P qui réalisent la porte de transmission, et que l'on peut factoriser cette porte au niveau du schéma comme cela est décrit en Figure 4-17.

La description de ces deux modèles peut s'exprimer sous forme prédicative par l'écriture suivante:

```
mos_n(XCN,XE,XS)
et mos_p(XCP,XE,XS)
et (inv(XCP,XCN,'0','1') ou inv(XCN,XCP,'0','1')).
```

Nous avons ainsi représenté en Figure 4-18 l'arbre binaire *{et,ou}* correspondant.

Figure 4-16 deux réalisations possibles d'un élément de multiplexeur

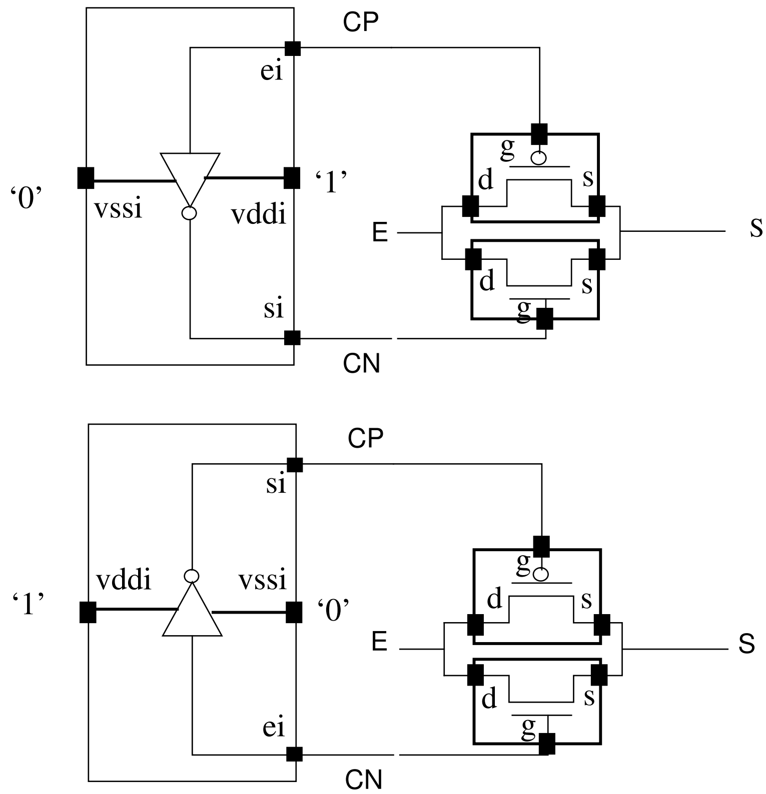


Figure 4-17 modèle décrivant 2 configurations possibles de l'élément de multiplexeur

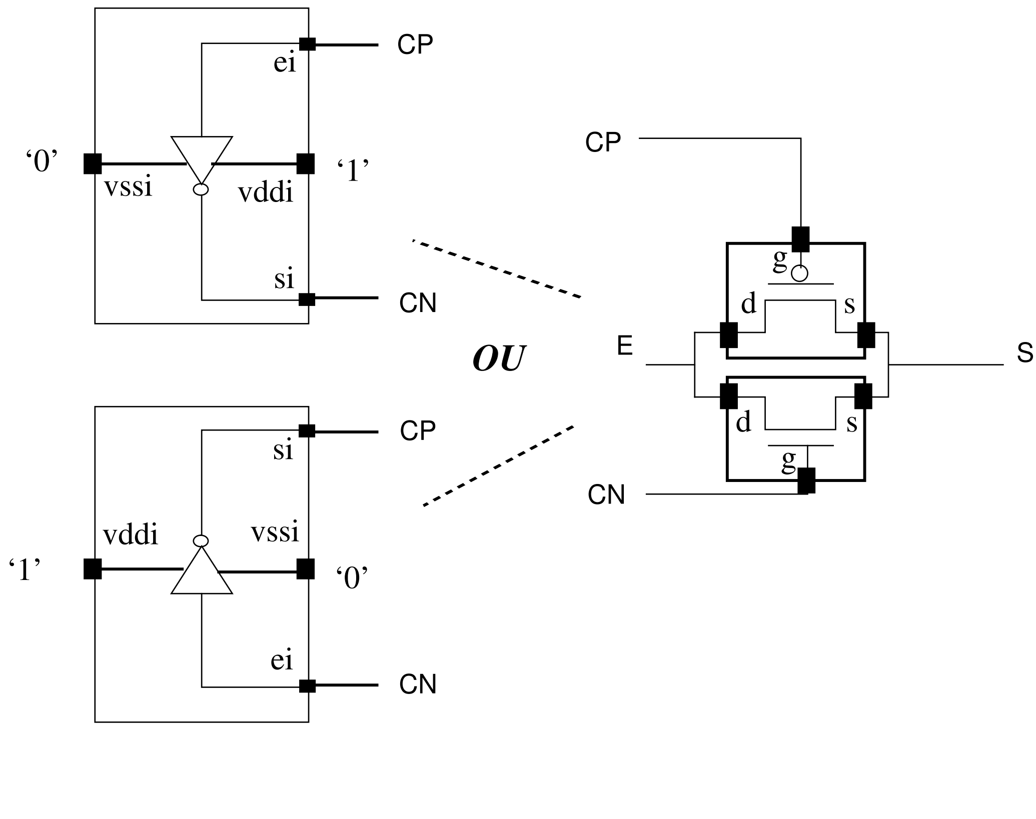
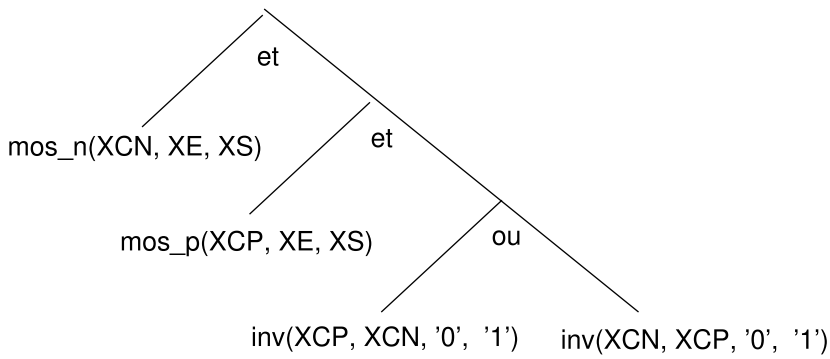


Figure 4-18 représentation de l'arbre {et,ou} du modèle d'élément de multiplexeur



Chaque modèle peut ainsi être décrit sous la forme d'une structure d'arbre ***{et, ou}*** de telle sorte qu'il reflète non seulement la structure du composant à reconnaître mais qu'il permette aussi d'en organiser la reconnaissance, car l'évaluation de l'arbre de prédicats s'effectue sur un mode dit profondeur-gauche qui consiste à explorer l'arbre en profondeur d'abord, et de la gauche vers la droite. Nous voyons alors sur l'exemple de la Figure 4-18 qu'aura d'abord lieu la recherche de la configuration

{ mos_n(XCN,XE,XS) et mos_p(XCP,XE,XS) }

et ensuite celle des configurations

{ inv(XCP,XCN,'0','1') ou inv(XCN,XCP,'0','1') }

Lorsque le premier prédicat *mos_n(XCN,XE,XS)* est activé, les variables *XCN*, *XE* et *XS* n'ont pas encore été affectées (elles sont dites libres), et elles seront affectées dès que ce prédicat aura filtré un composant inverseur dans la *net-list* de composants. Lorsque le deuxième prédicat *mos_p(XCP,XE,XS)* est activé à son tour, et comme la variable *XS* vient d'être affectée, le sous ensemble constitué par les transistors *mos_n* connectés à *XS* sera précisement connu. L'abstracteur exploite donc les propriétés de localité du schéma pour éviter de parcourir toute la *net-list* de composants pour retrouver un composant recherché.

Remarque: Il est à noter que lorsqu'une variable est affectée, elle garde sa valeur, et elle ne pourra pas être affectée à nouveau, à moins qu'elle ne retrouve son état antérieur dit libre lors des opérations de backtrack. Lorsqu'une variable est affectée, elle ne peut se voir proposer qu'une valeur égale à celle qu'elle a déjà reçu, sans quoi l'activation du prédicat correspondant échoue. On met ainsi en oeuvre un mécanisme dit d'unification.

Les échecs dans l'évaluation des prédicats sont pris en compte grâce à des **retours arrière** dans le parcours de l'arbre *{et,ou}* du modèle. Ces retours arrière permettent d'essayer d'autres solutions lorsqu'une hypothèse n'a pas pu être validée. Ceci permet donc de rechercher les nouveaux objets qui sont susceptibles de satisfaire l'hypothèse exprimée par le prédicat en cours d'évaluation, et cela après avoir remis les variables qui étaient précédemment valuées à leur état libre.

Le retour arrière opère tant qu'un ensemble d'objets n'a pas pu satisfaire le modèle autour du point d'ancrage sélectionné, et tant qu'il restera encore des composants susceptibles de satisfaire le modèle.

Un composant ne peut appartenir à deux Unités Abstraites. C'est pour cela que dès qu'un composant est filtré par un prédicat, il est immédiatement **marqué** pour ne plus être accessible dans la *net-list* de composants. Un tel marquage consiste pratiquement à positionner le champ *utilisé* (voir remarque du chapitre 4.2.3) du composant concerné. Le marquage d'un composant peut bien sûr être défait au moment du retour arrière.

Dés que le modèle est satisfait, la partie action de la règle est activée, et les composants constitutifs du modèle reconnu restent marqués et sont remplacés par l'Unité Abstraite correspondante. L'exploitation du modèle ne cesse tout naturellement que lorsqu'il n'y a plus de point d'ancrage dans la *net-list* de composants.

4.6 L'exploitation des modèles par les règles d'abstraction

4.6.1 Principes

Lorsqu'un modèle est reconnu, il faut remplacer les composants qui constituent le modèle reconnu par l'Unité Abstraite correspondante. Pour cela, nous associons la description du modèle et la description de l'action correspondante dans ce que nous appellerons désormais une **règle d'abstraction**.

L'exemple de la Figure 4-19 décrit ainsi une règle d'abstraction permettant de reconnaître un inverseur.

Figure 4-19 première forme de règle d'abstraction

```
if
        mos_n(E,S,'0')
and
        mos_p(E,S,'1')
then
        build inv(E,S,'0','1')
end
```

Nous dirons que la partie de la règle qui suit le mot clé ***if*** est la partie **modèle**, et que la partie qui suit le mot clé ***then*** est la partie **action** de la règle. Les mots ***and*** et ***or*** servent à traduire les opérateurs *et* et *ou* précédemment décrits.

Dans l'exemple de la Figure 4-19, l'Unité Abstraite inverseur qui doit être produite à chaque fois que le modèle d'inverseur est reconnu est ici décrite par le prédicat *inv(E,S,'0','1')* où *E* et *S* sont ici des variables. Les variables servent à filtrer et recueillir des valeurs, et la visibilité des variables est locale à chaque règle.

Il est à noter que l'interprétation de telles structures de règles au travers d'un programme informatique réalise un système dit **système de production**. En effet, une règle d'abstraction est analogue à une règle de production [IaRp86],[AiPrincip82], qui permet de faire passer l'état interne d'un résolveur d'un état initial E vers un état E' en enclenchant la partie action de la règle quand sa partie condition est satisfaite. De plus, dans les règles d'abstraction, nous avons combiné le formalisme des règles de production à celui des prédicats.

Ici les composants sont les objets manipulés par les règles d'abstraction, et ils se trouvent à ce titre tous rassemblés dans une **base d'objets**.

4.6.2 Mécanismes de déclenchement

Le déclenchement de la partie action de la règle est l'opération par laquelle est enregistrée dans la base d'objets un objet Unité Abstraite qui remplace les objets constituant le modèle reconnu grâce à l'ordre ***build***. Lors de l'enregistrement de l'objet Unité Abstraite, l'expression qui décrit l'Unité Abstraite à construire doit être **close**, ce qui signifie que toutes les variables référencées par cette expression sont valuées.

Il est à noter ici que dans notre système, l'enregistrement des résultats est immédiat et non pas différé, ce qui le distingue là aussi d'autres systèmes d'abstraction à base de règles tel que [Dialog89]. Un tel comportement aura pour effet, comme nous le verrons par la suite, de permettre à une règle d'exploiter à nouveau ses propres résultats.

Un tel comportement qui est dit non monotone, pourrait avoir comme conséquence de rendre le système difficilement contrôlable si la création d'une Unité Abstraite conduisait au déclenchement d'autre règles. C'est pourquoi nous avons adopté la stratégie suivante d'activation des règles: quand une règle est utilisée, le moteur d'inférence cherche à appliquer cette règle jusqu'à ce que celle ci ne puisse plus donner de solution.

Il est important de préciser que cette règle une fois utilisée ne sera par la suite plus jamais utilisée. C'est ainsi par exemple, que l'utilisation de la règle *inverseur* conduit à la reconnaissance de toutes les Unités Abstraites qui sont des instances du modèle de l'inverseur. Après quoi, on passe à la règle suivante, et ainsi de suite, jusqu'à ce qu'il ne reste plus du tout de règle à activer. Il ne peut donc pas y avoir de récursivité entre les règles, c'est pourquoi un tel mode de fonctionnement a l'avantage d'être à la fois sûr d'utilisation et simple à appréhender.

4.6.3 Conséquences

Une conséquence importante du formalisme de type règle de production associé aux mécanismes particuliers de déclenchement que nous venons de voir, est que chaque modèle peut être décrit individuellement dans une règle distincte, et donc indépendamment des autres modèles.

La construction de chaque Unité Abstraite peut donc être précisement associée à un ensemble de une ou plusieurs règles contiguës. Ceci permet d'effectuer aisément toutes les opérations élémentaires de maintenance des modèles que sont l'ajout, la suppression et la modification, sans avoir à connaître ni à agir sur l'ensemble des règles. Il suffit alors de connaître la spécification d'une Unité Abstraite pour l'utiliser dans un ou plusieurs modèles. Ceci rend notre système de règles modulaire, et comme nous allons le voir, nous pouvons aussi ordonnancer ces règles de façon statique.

4.7 L'ordonnancement des règles d'abstraction

Nous venons de définir précédemment le processus d'abstraction comme un processus ré-entrant consistant à substituer des Unités Abstraites à des primitives et à d'autres Unités Abstraites. Les Unités Abstraites construites dans la partie action de la règle se substituent en effet aux primitives et Unités Abstraites référencées dans la partie modèle de cette même règle.

Toute Unité Abstraite construite dans la partie action d'une règle est donc mise en relation de dépendance avec les Unités Abstraites référencées dans la partie modèle de cette règle.

Comme une règle est bijectivement associée dans sa partie action avec une Unité Abstraite construite, il existe des relations de dépendance entre les règles.

Ces relations de dépendance définissent un graphe orienté acyclique entre les règles, et l'on peut déduire d'un tel graphe un ordre de classement qui éliminera les références en avant. Un tel ordre est un ordre partiel parce qu'il peut arriver que plusieurs règles concluent sur une même Unité Abstraite, auquel cas c'est l'ordre d'écriture des règles qui est pris en compte afin d'arriver à classer les règles et obtenir finalement un ordre total.

Remarque: Il est important de noter ici que les relations de dépendance entre les règles s'établissent uniquement par l'intermédiaire des Unités Abstraites référencées dans les parties modèle et action de ces règles. Il n'est donc pas question ici d'avoir des références directes d'une règle à une autre avec échange d'arguments comme cela peut se trouver dans d'autres systèmes à base de règles tel que par exemple là aussi dans [Dialog89].

L'exemple suivant de la Figure 4-20 montre l'ordonnancement d'un système de règles dans lequel on ne fait figurer que les prédicats qui décrivent des Unités Abstraites symbolisées par *Ei*, et des primitives symbolisées par *Pi*.

Figure 4-20 exemple de système de règle à ordonnancer

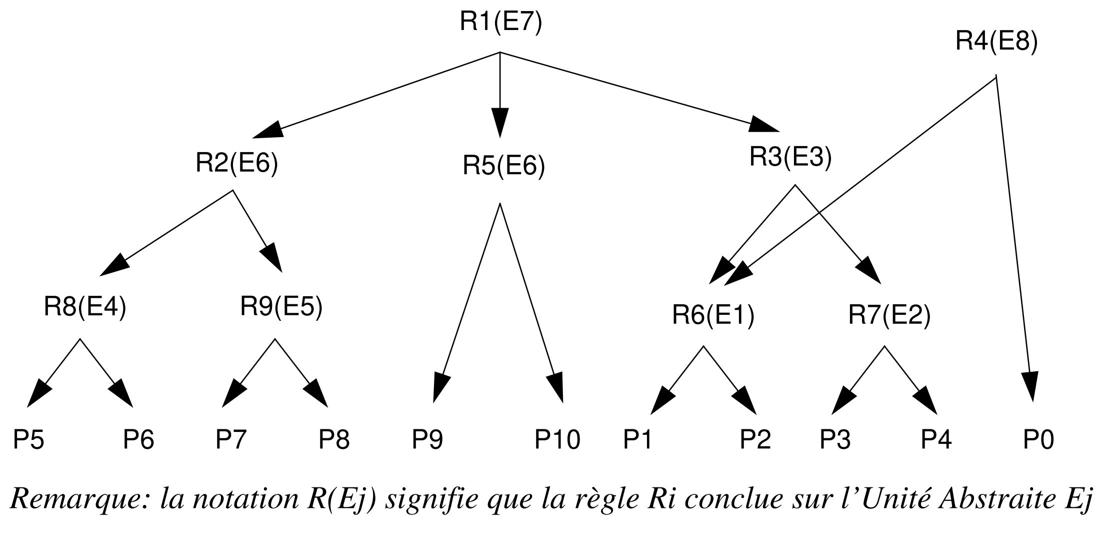
```
-> "règle R1" if E6 and E3 then build E7 end
-> "règle R2" if E4 and E5 then build E6 end
-> "règle R3" if E1 and E2 then build E3 end
-> "règle R4" if E1 and P0 then build E8 end
-> "règle R5" if P9 and P10 then build E6 end

-> "règle R6" if P1 and P2 then build E1 end
-> "règle R7" if P3 and P4 then build E2 end
-> "règle R8" if P5 and P6 then build E4 end
-> "règle R9" if P7 and P8 then build E5 end
```

Dans cet exemple, on voit que les règles *R2* et *R5* concluent sur la même Unité Abstraite *E6*, mais comme *R2* précède *R5*, *R2* sera résolue avant *R5*. Les règles *R3* et *R4* utilisent la même Unité Abstraite *E1*, mais comme *R3* précéde *R4*, *R3* sera activée avant *R4*.

Le graphe de dépendance des règles de la Figure 4-20 est alors le suivant:

Figure 4-21 exemple de graphe de dépendance entre des règles



Remarque: la notation R(Ej) signifie que la règle Ri conclue sur l'Unité Abstraite Ej

Dans la Figure 4-22 ci-dessous, on peut voir l'ordre d'activation des règles de la Figure 4-20 tel qu'il est déduit à la fois de l'organisation préalable des règles et de leur contenu.

Figure 4-22 exemple d'ordonnancement des règles

```
(1) regle R8
(2) regle R9
(3) regle R2
(4) regle R6
(5) regle R7
(6) regle R3
(7) regle R5
(8) regle R1
(9) regle R4
```

Nous allons exprimer maintenant l'algorithme de calcul de l'ordonnancement pour obtenir un ordre total sur les règles. Cet algorithme qui est décrit en Figure 4-23 consiste à prendre en compte successivement chacune de règles d'une liste non ordonnée, pour aller rechercher récursivement les règles qui servent à construire les Unités Abstraites instanciées dans la règle traitée pour les traiter à leur tour.

Cette procédure d'ordonnancement vérifie également en Figure 4-24 que le graphe de dépendance est effectivement sans cycle.

On utilise pour cela:

- une liste *LREGI* qui est la liste initiale des règles à ordonner.
- une liste *LREGO* qui est initialement vide et qui contient la liste des règles ordonnées.
- une fonction *ENTITES_MODELE(règle)* qui retourne la liste des Unités Abstraites qui sont instanciées dans le modèle de la règle notée *règle*.
- une fonction *ENTITE_CONSTRUITE(règle)* qui retourne l'Unité Abstraite sur laquelle conclue une règle notée *règle*.
- une fonction *CONCLUE_SUR(UA)* qui retourne la liste de toutes les règles qui concluent sur une même Unité Abstraite notée *UA*. S'il n'y a pas de règle qui conclue sur l'Unité Abstraite désignée, cela constitue un cas d'erreur.

Figure 4-23 description de la procédure recursive d'ordonnancement

```
Définition_Procedure ordonner_regles(LREGI, LREGO)
   Si (VIDE(LREGI)) Alors
        Retour ;
   Fin de Si
   R1 = PREMIER(LREGI) ;
   Si ( Non MARQUEE(R1) ) Alors
        MARQUER(R1) ;
        LENTM = ENTITES_MODELE(R1) ;
        Pour_Chaque EM de LENTM Faire
             LREGC = CONCLUE_SUR(EM) ;
             controler_règle_non_cyclique(R1, LREGC, LREGO) ;
             ordonner_règles(LREGC, LREGO) ;
        Fin de Pour_Chaque
        AJOUTER(R1, LREGO) ; % ajoute R1 à la liste LREG0
   Fin de Si
   LREGI = SUIVANT(LREGI) ; % accés au suivant de la liste LREGI
   ordonner_regles(LREGI, LREGO) ;
Fin de Définition_Procedure
```

Figure 4-24 description de la procédure qui contrôle l'absence de cycle dans les règles

```
Définition_Procedure controler_règle_non_cyclique(R1, LREGC, LREGO)
   Pour_Chaque RX de LREGC Faire
        Si (      MARQUEE(RX)
                  Et Non APPARTIENT(RX,LREGO)
                  Et RX Different_De R1 )
        Alors
             ERREUR(" La règle est dans un cycle : ", RX) ;
        Fin de Si
   Fin de Pour_Chaque
Fin de Définition_Procedure
```

4.8 Le format de description du langage de règles

Nous voulions que le langage GRL qui décrit les règles d'abstraction puisse être mis en oeuvre par les concepteurs de VLSI eux mêmes, et qu'il n'exige pas de ceux ci des connaissance préalables en programmation.

C'est pourquoi nous avons fait en sorte que ce langage:

1. soit suffisamment simple à appréhender,
2. soit directement transposable à partir des descriptions schématiques des circuits.

De façon générale, une règle d'abstraction est annoncée par un symbole noté '**->**' suivi d'une chaîne de caractères, cette dernière servant à décrire brièvement la règle et à l'identifier lors de son activation.

La forme d'écriture compacte de l'exemple de la Figure 4-19 obligeait à citer tous les champs de l'objet même s'ils n'étaient pas utilisés. Pour les primitives, nous avons préféré une forme d'écriture plus explicite dans laquelle on puisse mettre en correspondance un nom de connecteur du composant avec un nom de signal.

Dans le cas des Unités Abstraites, le nombre des connecteurs et leur nommage diffèrent d'un modèle à un autre, c'est pourquoi nous avons alors choisi une méthode d'identification de nature positionnelle pour identifier les connecteurs de chaque Unité Abstraite. Nous avons ainsi défini 3 attributs notés ***input***, ***command*** et ***output*** pour répartir les connecteurs de chaque Unité Abstraite selon 3 listes distinctes qui correspondent respectivement aux ports d'entrée, de contrôle et de sortie de l'Unité Abstraite.

Tandis que l'exemple de la Figure 4-19 décrivait une première forme de règle d'inverseur, nous pouvons l'exprimer maintenant grâce au langage de règles de la façon suivante en Figure 4-25.

Figure 4-25 forme définitive de règle d'inverseur

```
-> "regle de reconnaissance d'inverseur"
if
   mos_n with g=E ; s='0' ; d=S
and
   mos_p with g=E ; s='1' ; d=S
then
   build inverseur with input := E ; output := S
end
```

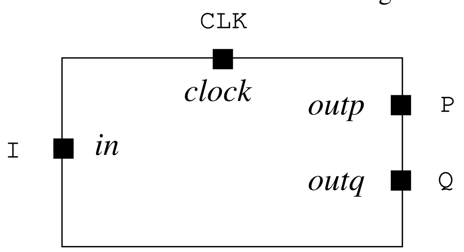
On peut voir, toujours dans cet exemple, que l'opérateur '=' signifie dans ce contexte que la variable écrite en caractères majuscules et située à droite s'unifie avec l'attribut situé à gauche. Par exemple, lorsqu'on cite un transistor *mos_n* dans un modèle, on peut le décrire ainsi si on veut citer tous ses attributs:

mos_n with g=G ; d=D ; s=S ; x=X ; y=Y ; w=W ; l=L

Il est à noter à propos de cet exemple que l'abstracteur considère les connections de drain et source des transistors Mos comme équivalentes, cela étant par ailleurs spécifié en 4.10.2. Le symbole '**:=**' dans la partie action exprime alors l'affectation des valeurs référencées par les variables aux attributs de l'Unité Abstraite construite. Le mot clé ***with*** sert de séparateur entre le type de l'objet et la liste des contraintes sur ses attributs. Le mot clé ***build*** sert à spécifier l'action de construction d'Unité Abstraite.

La Figure 4-26 montre maintenant comment l'ordre spécifique de construction d'une Unité Abstraite plus complexe, telle que par exemple une Unité Abstraite de type registre peut s'écrire. Le langage GRL est par ailleurs complètement décrit et spécifié en ANNEXE B.

Figure 4-26 ordre de construction d'une Unité Abstraite avec un signal de commande




```
...
   build register with
        input := I ; command := CLK ; output := P,Q
end
```

4.9 La production des comportements HDL

Nous voulons pouvoir produire les descriptions des comportements Vhdl et Verilog de type flot de données pour chaque circuit reconnu. C'est pourquoi nous avons associé systématiquement à chaque Unité Abstraite les descriptifs fonctionnels qui traduisent son comportement dans chacun de ces langages grâce à deux attributs nommés ***vhdl*** et ***verilog***.

Si nous reprenons l'exemple précédent de la Figure 4-25 dans lequel est décrite une règle qui permet de reconnaître un inverseur, nous voyons en Figure 4-27 comment peuvent être ajoutés à la fois un comportement Vhdl et un comportement Verilog à l'Unité Abstraite inverseur, et ceci grâce aux deux attributs ***vhdl*** et ***verilog***.

Nous avons fait en sorte d'associer aux deux attributs ***vhdl*** et ***verilog*** précités des listes constituées d'éléments de texte et de Variables, et nous avons cirsconscrit ces listes au moyen des caractères '**{**' et '**}**'. La seule opération réalisée par l'abstracteur sur le descriptif HDL placé entre les caractères '**{**' et '**}**' consiste alors à retranscrire son contenu lors de la phase d'édition des comportements HDL du bloc abstrait.

C'est ainsi que le travail réalisé en fin d'abstraction sur les descriptifs contenus dans les attributs *vhdl* et *verilog* se limite à:

- éditer les éléments de texte tels qu'ils se présentent dans le descriptif HDL,
- éditer les noms des signaux, sachant que lors de l'abstraction, on a remplacé les variables présentes dans le descriptif HDL par les adresses des signaux filtrés par ces variables.

Cette façon de procéder présente l'avantage d'autoriser toutes les formes de construction d'expressions logiques, et elle est complètement indépendante des niveaux de spécification des langages Vhdl et Verilog, puisqu'il n'y a ici aucun traitement sémantique de ces langages. Elle permet aussi d'en traiter les extensions dans l'analogique, et d'introduire si besoin des éléments temporels comme par exemple l'instruction *after* en Vhdl. Elle permet encore de documenter les descriptifs HDL en leur associant des commentaires.

Figure 4-27 association de comportements HDL à une Unité Abstraite

```
-> "regle de reconnaissance des inverseurs"
if
   mos_n with g=E ; d='0' ; s=S
and
   mos_p with g=E ; d='1' ; s=S
then
   build inverseur with
        input := E ;
        output := S
        vhdl := { S <= not E ; // commentaire possible
             } ;
        verilog := { assign S = ~ E ;
             }
end
```

Il est à noter aussi que le traitement effectué est ici d'autant plus simple que les deux langages ciblés supportent les affectations concurrentes. Le principe du procédé reste cependant valide avec des langages qui ne supporteraient pas les affectations concurrentes, puisque chaque Unité Abstraite contient grâce à ses attributs *input*, *command* et *output* toutes les informations de connectique nécessaires pour ordonner les équations.

On notera aussi que dans la cas du Vhdl, on travaille le plus souvent avec un type par défaut positionné à *std_logic (*paquetage IEEE Std 1164*)*, c'est à dire toujours résolu. On peut cependant à des fins d'optimisation travailler avec le type par défaut *std_ulogic* (alors positionné sur la ligne de commande), et ne déclarer précisément comme résolus que les signaux qui doivent l'être. Pour réaliser cela, on vient positionner le champ *vh_type* des signaux concernés, comme dans l'exemple de la Figure 4-28, par une action ***update*** appliquée au signal et installée dans la partie action de la règle.

Figure 4-28 affectation du type Vhdl d'un signal

```
-> "regle de reconnaissance d'un element de multiplexeur"
if
   ....
and
       mos_n with g=C1 ; s=I1 ; d=S
and
       mos_n with g=C2 ; s=I2 ; d=S
then
       build mult with
            input := I1,I2; command := C1,C2; output := S;
            vhdl := { S <= I1 when (C1 ='1') else 'Z';
                      S <= I2 when (C2 ='1') else 'Z';
                      }
   and
        update signal S with vh_type := std_logic
end
```

4.10 Les opérateurs du langage de règles

4.10.1 Opérateurs d'exclusion et de présence

Pour qu'un modèle puisse s'appliquer, on peut souhaiter imposer des contraintes supplémentaires sur la définition des circuits à reconnaître. Des opérateurs spécialisés du langage de règles vont ainsi permettre d'exprimer les deux contraintes suivantes:

- **interdire** (exclure) explicitement des composants d'un modèle à reconnaître.
- **requérir** la présence de composants dans le voisinage d'un modèle à reconnaître.

Pour exprimer ces deux types de contraintes, nous avons défini des opérateurs que nous allons décrire.

opérateur *not*

Une première forme d'exclusion est réalisée à l'aide de l'opérateur ***not*** qui s'applique à un composant. Cet opérateur indique que la présence d'un composant du type spécifié connecté d'une certaine façon entraîne l'échec de l'unification du modèle avec le circuit recherché.

L'exemple de règle de la Figure 4-30 qui décrit le schéma de la Figure 4-29 exprime une utilisation de cet opérateur. On voit sur cet exemple que s'il y avait un troisième transistor sur le schéma, alors la règle échouerait.

Figure 4-29 schéma d'un multiplexeur 2 voies

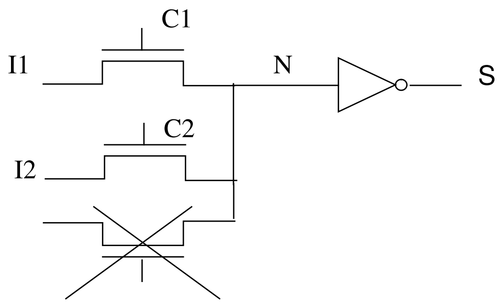


Figure 4-30 règle décrivant le multiplexeur 2 voies

```
-> "regle de reconnaissance d'un multiplexeur 2 voies"
if
   inverter with input=N ; output=S
and
   mos_n with g=C1 ; s=I1 ; d=N
and
   mos_n with g=C2 ; s=I2 ; d=N
and
   not mos_n with d=N
then
   build switch2 with
        input := I1,I2 ; command := C1,C2 ; output := S ;
        vhdl := { S <= not I1 when (C1='1' and C2='0')
                  else not I2 when (C1='0' and C2='1') else 'X';
             }
 end
```

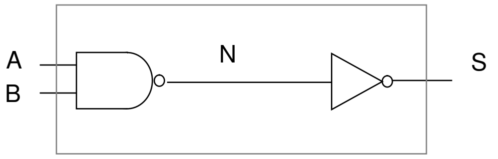
opérateur *reduced*

On pourra aussi exprimer qu'en dehors des composants cités dans la règle, on ne veut pas qu'un autre composant, quel qu'il soit, puisse être connecté en un noeud précis du modèle à reconnaître, ce qui signifie que ce noeud constitue un signal interne au modèle. On écrira ***reduced*** *N* dans la règle de la Figure 4-32 pour dire que le noeud électrique associé à la variable N n'est pas connecté à des composants autres que ceux qui sont cités explicitement dans le modèle.

C'est ainsi qu'en appliquant la règle décrite Figure 4-32, on reconnaîtra le cas de circuit *no 1* de la Figure 4-31, mais on ne reconnaîtra pas le cas *no 2*.

Figure 4-31 exemple d'utilisation de l'opérateur '*reduced*'

cas no 1: Porte Et 2 entrées: N est réduit (reduced)



cas no 2: Ce n'est pas une porte ET 2 entrées, N n'est pas réduit

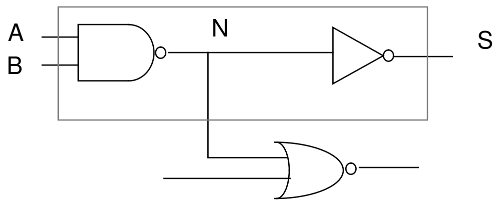


Figure 4-32 règle utilisant l'opérateur *reduced*

```
-> "regle de reconnaissance des portes ET"
if
   nand with input = A,B ; output=N
and
   inverter with input=N ; output=S
and
   reduced N
then
   build porte_et with
        input := A,B ; output := S ;
        vhdl := { S <= A and B;
             } ;
        verilog := { assign S = A&B ;
             }
 end
```

Remarque: Puisqu'il contrôle qu'il n'y a plus de composant connecté à un signal, cet opérateur *reduced* devra apparaître dans le modèle obligatoirement après la dernière occurrence de composant faisant référence à la variable citée.

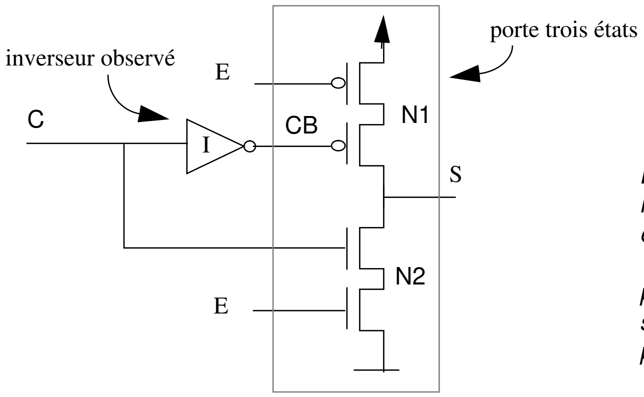
opérateur *observed*

On cherche maintenant à exprimer qu'une règle n'est applicable que s'il existe un composant qui ne fait pas partie du modèle, mais qui est connecté d'une façon particulière aux composants du modèle. L'absence de ce composant entraîne l'échec de l'unification du modèle avec le circuit.

Pour exprimer qu'un composant doit faire partie du voisinage d'un circuit à reconnaître, on utilise l'opérateur noté ***observed*** qui s'applique à un composant. C'est ce que montre l'exemple de règle de la Figure 4-34 qui décrit le circuit de la Figure 4-33.

Il est à noter que le composant observé par ce prédicat ne fait pas partie de la porte trois états, il n'est donc pas retiré de la *net-list* (il n'est pas marqué).

Figure 4-33 exemple d'utilisation de l'opérateur '*observed*'



L'inverseur I est observé, il sert à reconnaître que la porte 3 états est correctement commandée (CB = non C) mais il ne fait pas partie de sa description, il pourra servir à commander à lui seul plusieurs portes de ce type.

Figure 4-34 règle de reconnaissance d'une porte trois états

```
-> "règle de reconnaissance d'une porte trois états"
if
   mos_n with g=E ; s='0' ; d=N2
and
   mos_n with g=C ; s=N2 ; d=S
and
   reduced N2
and
   mos_p with g=CB ; s=N1 ; d=S
and
   mos_p with g=E ; s='1' ; d=N1
and
   reduced N1
and
   observed inverter with input=C ; output=CB
then
   build porte_trois_etats with
        input := E ;
        command := C ;
        output := S ;
        vhdl := {S <= not E when (C='1') else 'Z';
             }
 end
```

4.10.2 Equivalence des signaux

L'équivalence entre les signaux drain et source des transistors CMos qui est due à l'équivalence électrique des diffusions de drain et de source est automatiquement prise en compte par le programme.

Il est donc équivalent d'écrire par exemple:

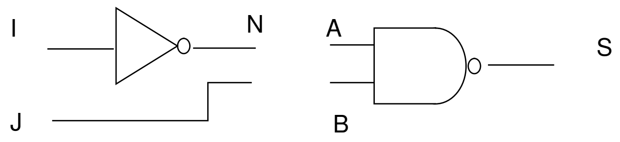
mos_n with g=G ; d=X ; s=Y

ou bien

mos_n with g=G ; d=Y ; s=X

On peut aussi chercher à exprimer explicitement des équivalences fonctionnelles entre plusieurs schémas. A cet effet, nous avons introduit un opérateur dit d'équivalence '**?=**' qui appliqué à deux listes permet de réaliser toutes les permutations possibles entre les listes situées de part et d'autre de cet opérateur. C'est ce que montre le schéma de l'exemple de la Figure 4-35.

Figure 4-35 . Exemple de permutations sur des listes



on cherche à résoudre l'équivalence: (N=A et J=B) ou (N=B et J=A).

A partir de cet exemple, on voit comment la forme d'écriture *A,B ?= N,J*
permet de traduire directement l'équivalence *(N=A et J=B) ou (N=B et J=A).*

On retrouve cette expression insérée dans la règle de la Figure 4-36. Cette règle permet de reconnaître le schéma de la Figure 4-35.

Figure 4-36 règle utilisant l'opérateur de permutation

```
-> "regle de reconnaissance des portes AN1"
if
   nand with input = A,B ; output=S
and
   A,B ?= N,J
and
   inverter with input=I ; output=N
and
   reduced N
then
   build porte_an1 with
        input := I,J ; output := S ;
        vhdl := { S <= I or not J;
             } ;
        verilog := { assign S = I | ~J;
             }
 end
```

4.10.3 Concaténation de listes

Nous avons voulu simplifier la manipulation des listes. L'opérateur "," effectue donc une concaténation systématique de plusieurs listes lors d'une affectation. C'est ce que montre l'exemple suivant.

Figure 4-37 exemple de concaténation automatique des listes

```
-> "exemple"
if
...
   L1=A,B,C
and
   L2=D,E,F
then
   build ua_ex with input := L1,L2 ; ...
end
```

Dans cet exemple où on a L1 = A,B,C et L2 = D,E,F, le fait d'affecter *L1, L2* à l'attribut *input* de *ua_ex* permet d'enregistrer la liste résultante constituée par *A,B,C,D,E,F* dans cet attribut.

Remarque: Il appartient à l'utilisateur de vérifier qu'il n'unifie pas une variable avec une liste qui fasse référence à cette même variable. Par exemple, l'opération *B = A,B,C* est illégale, puisque la variable *B* est présente dans les termes de droite et de gauche, et cela sera signalé comme cas d'erreur par le programme.

4.11 La hiérarchisation de l'abstraction

Grâce au caractère ré-entrant de la méthode d'abstraction, nous pouvons **hiérarchiser** l'abstraction. Pour cela, le modèle d'un composant complexe fera référence aux modèles d'Unités Abstraites qui devront être préalablement reconnues.

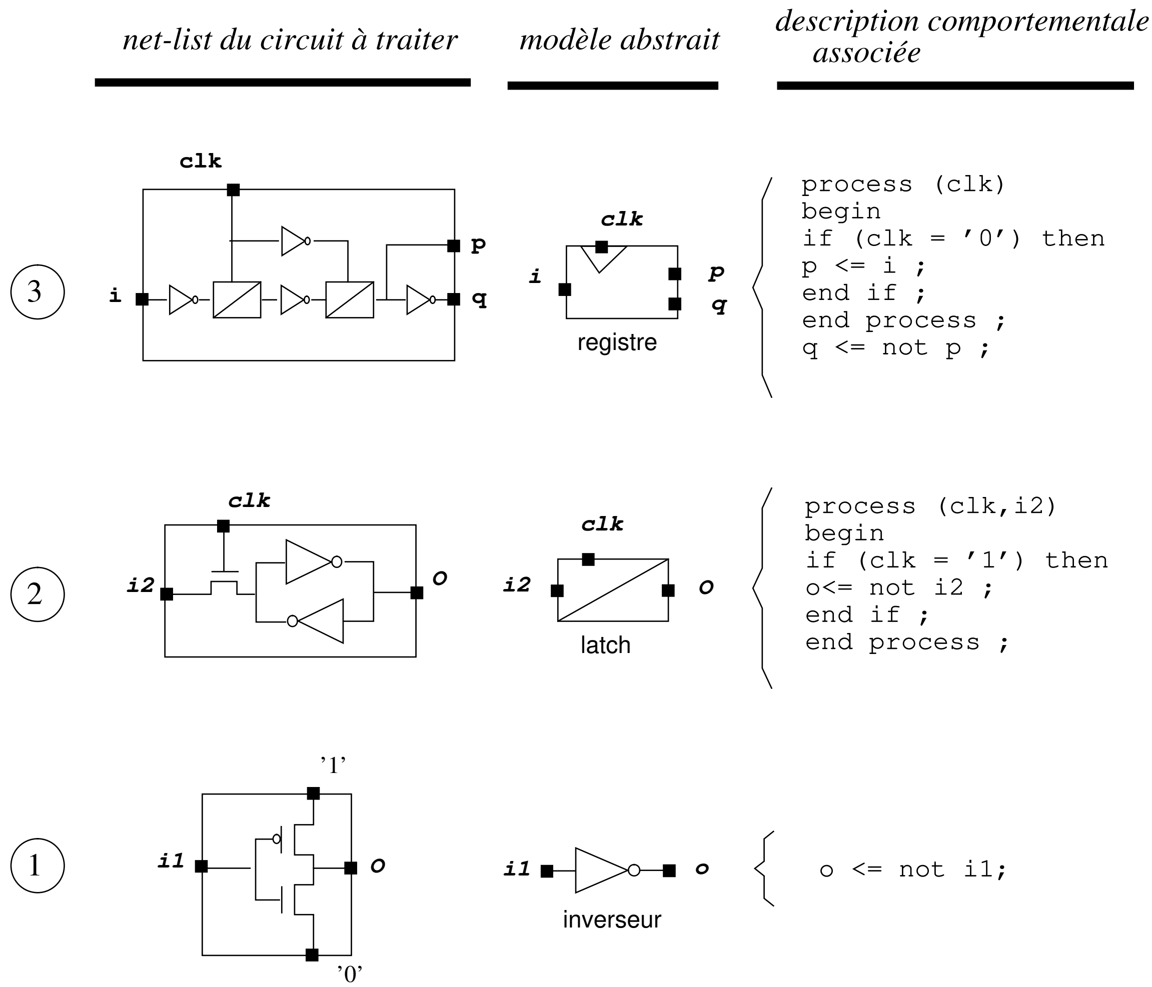
La base d'objets est de cette façon utilisée à la manière d'un espace de travail en permettant à toute Unité Abstraite d'être réutilisée pour construire une Unité Abstraite de plus haut niveau.

L'abstraction hiérarchique présente plusieurs avantages:

- elle facilite la gestion des modèles par la personne qui écrit les règles.
- elle permet d'optimiser le processus de reconnaissance qui gagne en efficacité puisque l'arbre de recherche que constitue le modèle hiérarchisé devient moins complexe, et que les opérations de retour arrière effectuées lors de l'activation du modèle sont moins nombreuses.
- le résultat fonctionnel global que l'on obtient est plus concis. En effet, il est plus efficace au niveau simulation et aussi plus précis de décrire les comportements des éléments mémorisants au niveau registre qu'au niveau latch.

Nous illustrons cet aspect hiérarchique par l'exemple de la Figure 4-38 qui décrit la reconnaissance d'un registre. Nous voyons que les inverseurs sont d'abord reconnus en (1), puis les latches en (2) à partir des inverseurs et ensuite les registres en (3) à partir de ces mêmes latches. A chaque niveau d'abstraction, une description comportementale peut être associée. Aussi, quel que soit le niveau d'abstraction effectivement réalisé, (1) ou (2), ou (3) on obtiendra toujours une description fonctionnelle cohérente et complète.

Figure 4-38 exemple d'abstraction hiérarchique



Remarque: il est à noter sur cet exemple que la description du comportement du registre sous la forme d'un process validé par un front d'horloge constitue en elle même une abstraction comportementale de plus niveau, car on fait ici l'hypothèse que le circuit est observable sur le front descendant de l'horloge *clk* uniquement.

4.12 Le traitement des structures génériques

Les règles cumulatives ont essentiellement pour but de reconnaître les circuits formés d'un nombre variable de composants de même type. Pour mettre en oeuvre un tel mode, nous mettons à profit le fait qu'une règle puisse modifier l'état de la base d'objets en créant des Unités Abstraites, et ré-utiliser ensuite les nouvelles Unités Abstraites ainsi ajoutées. Il suffit alors, pour obtenir un mode de fonctionnement cumulatif, qu'une règle qui crée une Unité Abstraite d'un certain type dans sa partie action, fasse référence à une Unité Abstraite de même type dans sa partie modèle.

C'est ce mécanisme qui permet de reconnaître des circuits composés d'un nombre variable et indéterminé de composants comme nous le montrons dans l'exemple de la Figure 4-39, et ceci sans qu'il soit nécessaire de faire appel à des procédures récursives supplémentaires. Cet exemple décrit ici un multiplexeur à 5 entrées. Le mécanisme cumulatif permet de décrire simplement dans une règle cette configuration de circuit, et de la reconnaître. De façon générale, il sera possible de reconnaître n'importe quel multiplexeur dont on ne connaît pas à priori le nombre d'entrées et donc le nombre de transistors qui le composent.

A partir du schéma de la Figure 4-39 qui décrit un multiplexeur (avec ici 5 entrées), nous voyons en Figure 4-40 comment avec 2 règles il est possible de reconnaître une telle structure.

Figure 4-39 exemple de *net-list* de multiplexeur à reconnaître

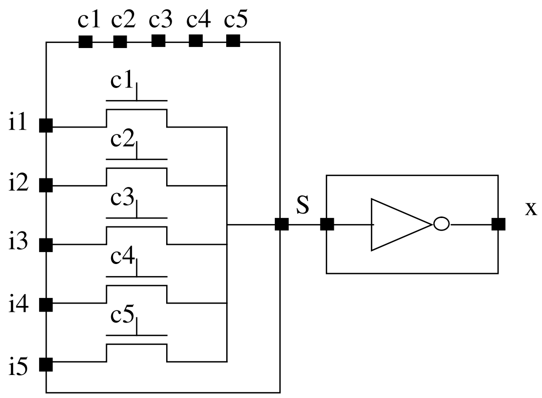


Figure 4-40 ensemble de 2 règles pour reconnaître un multiplexeur

```
-> "Multiplexeur: premiere regle"
if
   observed inverter with input=S
and
   mos_n with gate=C ; source=I ; drain=S
then
   build mult with
            input := I ;
            command := C ;
            output := S ;
            vhdl := { S <= I when (C='1') else 'Z';
                         }
end

-> "Multiplexeur: deuxieme regle"
if
   mult with input=I1 ; command=C1 ; output=S ; vhdl = V1
and
   mult with input=I2 ; command=C2 ; output=S ; vhdl = V2
then
   build mult with
            input := I1,I2 ;
            command := C1,C2 ;
            output : = S ;
            vhdl := V1,V2;
end
```

La première règle, qui est activée tout d'abord lors de l'abstraction, consiste à transformer chaque transistor Mos du multiplexeur en une Unité Abstraite *mult,* tandis que la deuxième règle qui est activée ensuite, permet regrouper ces Unités Abstraites *mult*.

On peut voir dans la deuxième règle, que les listes des entrées et commandes de l'Unité Abstraite alors construite sont obtenues en concaténant les listes d'entrée *I1* et *I2* du modèle (*input := I1,I2*) d'une part, et en concaténant les listes de commandes *C1* et *C2* du modèle (*command := C1,C2*) d'autre part.

4.13 La vectorisation du HDL

La vectorisation est une étape de l'abstraction qui permet de compacter le comportement HDL en exprimant les comportements au niveau vecteur de bits au lieu de les exprimer au niveau bit. Ceci s'effectue en analysant préalablement les noms de signaux indexés présents dans la *net-list*, et en propageant les informations d'indexation obtenues dans la logique abstraite, comme nous allons le décrire ci après. Après vectorisation, le HDL qui est produit est plus compact, plus lisible, et aussi beaucoup plus efficace en simulation.

Nous donnons ainsi en exemple dans la Figure 4-41 un ensemble de portes trois états allant de *t0* à *t5,* qui partagent le même signal de commande *k* et qui affectent un même bus de données *busa*. Nous avons représenté en Figure 4-42 le comportement non vectorisé de cet ensemble de portes, et en Figure 4-43 le même comportement après vectorisation.

Figure 4-41 ensemble vectorisable de portes 3 états

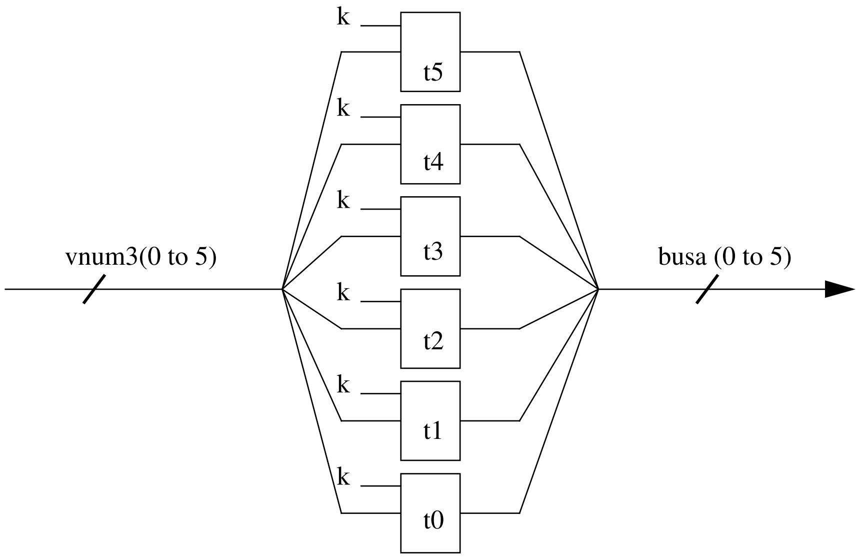


Figure 4-42 exemple de comportement non vectorisé de 6 portes 3 états sur un bus.

```
busa(5) <= not vnum3(5) when (k = '1') else 'Z';
busa(4) <= not vnum3(4) when (k = '1') else 'Z';
busa(3) <= not vnum3(3) when (k = '1') else 'Z';
busa(2) <= not vnum3(2) when (k = '1') else 'Z';
busa(1) <= not vnum3(1) when (k = '1') else 'Z';
busa(0) <= not vnum3(0) when (k = '1') else 'Z';
```

Figure 4-43 vectorisation du comportement de 6 portes 3 états sur un même bus

```
busa(5 downto 0) <= not vnum3(5 downto 0) when (k = '1')
else "ZZZZZZ";
```

On voit dans cet exemple qu'un ensemble initial de 6 équations de transfert entre 2 vecteurs de 6 bits se trouve ramené à une seule équation.

Lors de l'abstraction, la vectorisation est effectuée quand toutes les règles ont été activées, et donc quand tous les circuits ont été reconnus.

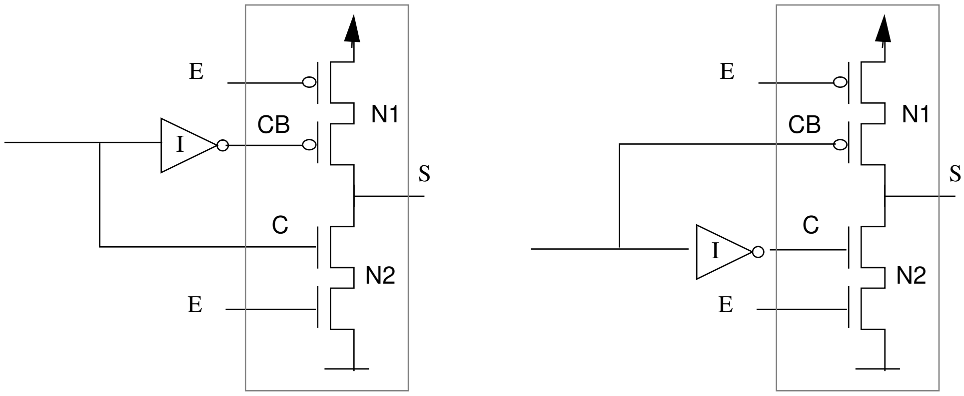
La méthode que nous avons employée pour effectuer la vectorisation se décompose en 6 étapes que nous décrivons comme suit:

1. on identifie tout d'abord les vecteurs dans la *net-list* à partir de noms de signaux. Il peut s'agir ici de noms de signaux d'interface ou bien de noms signaux internes au bloc. Pour identifier les vecteurs, on regarde si les noms de signaux sont constitués d'une racine suivi d'un indice numérique. On retiendra par exemple, selon les formats de *net-list* exploités des noms tels que `sa(3), sb[5]` ou `sc_8.`
2. On classe ensuite ces signaux en les triant d'après les deux critères successifs de nom et d'indice. Ceci permet de regrouper et ordonner les signaux de même racine et d'indices successifs dans une même structure de type vecteur.
3. A partir de là, on parcourt toutes les structures de type vecteur qui viennent d'être crées, et on recherche pour les signaux de chaque vecteur quelles sont les Unités Abstraites qui leur sont associées. A ce niveau, on distingue les Unités Abstraites entre celles qui sont affectées en entrée par ces signaux, et celles dont les sorties affectent ces signaux.
4. On cherche ensuite a créer des partitions de vecteurs homogènes. Pour cela, on recherche pour chaque vecteur quels sont les groupes ou sous groupes d'Unités Abstraites vectorisables qui respectent chacun des 4 critères d'homogénéité que nous listons ci après:
    - les Unités Abstraites doivent avoir la même désignation (par exemple *inverseur*, *latch*, *registre*, ...),
    - les Unités Abstraites doivent avoir des modèles de comportement HDL identiques,
    - si elles sont contrôlées (signaux de l'attribut *commande*), les Unités Abstraites doivent partager des signaux de contrôle identiques,
    - les signaux vectorisables qui servent au regroupement des Unités Abstraites doivent avoir des indices continus, c'est à dire qui évoluent de 1 en 1.
5. Toutes les Unités Abstraites qui appartiennent à une même partition homogènes sont regroupées dans les structures de type vecteur-ua. Ce sont ces structures vecteur-ua qui peuvent être éditées sous une forme vectorisée.
6. On essaye maintenant de propager la vectorisation, et pour cela, on cherche à identifier à partir de chacune des structures dites vecteur-ua de nouveaux vecteurs de signaux. C'est ainsi que certains signaux qui initialement n'avaient pas de nom dans la *net-list* originelle peuvent se voir attribuer automatiquement, lors de cette étape, un nom composé d'une racine suivie d'un indice. Ces nouveaux noms sont construits en accolant la racine *vnum* à un incrément *i* représentatif du vecteur, et en lui associant un indice *j* qui représente l'occurrence du signal dans le vecteur pour obtenir un nom résultant du type *vnumi(j)*. On obtiendra par exemples des noms tels que *vnum3(12), vnum5(1), vnum5(2),*.... Dans l'exemple précédent de la Figure 4-41, on voit que le bus désigné par *busa* sert à construire après vectorisation le vecteur désigné par *vnum3*.
7. Les signaux qui viennent d'être nouvellement nommés sont utilisés pour propager la vectorisation en ré-itérant le procédé à partir de l'étape numéro 1. Le procédé de vectorisation est ainsi enclenché selon un algorithme de relaxation, et il ne s'arrête que lorsque plus aucune vectorisation n'est possible.

Il est à noter que la vectorisation ne peut s'effectuer dans certains cas que si les constantes qui décrivent les états logiques tels que '0', '1', 'Z', 'X', 'H', 'L' en Vhdl et leurs équivalents Verilog peuvent être dimensionnés d'après la taille des vecteurs de signaux auxquels ils sont affectés. On voit en effet dans l'exemple de la Figure 4-43, que l'affectation du bus *busa* à l'état haute impédance se fait par l'intermédiaire du vecteur d'état haute impédance "ZZZZZZ" de même taille que celle du bus. Pour réaliser cela facilement, nous avons été amenés à introduire dans les textes HDL des variables désignées par $0, $1, $Z, $X, $H, $L. Ces variables recevront chacune respectivement un état logique de type ‘0’, ‘1’, ‘Z’, ‘X’, ‘H’, ‘L’ dont la taille sera rendue identique à celle du vecteur. C’est ce que montre l’exemple de règle de la Figure 4-44 qui permet de produire le comportement de l’exemple de la Figure 4-43 dans lequel l’état logique ‘Z’ est automatiquement calibré sur la largeur du vecteur.

Figure 4-44 règle d’abstraction d’une porte 3 états avec comportement vectorisable

Schéma des 2 configurations de portes 3 états à reconnaître



Règle d’abstraction exprimant les 2 configurations

```
-> "trst_1 R2"
if
        mos_p with d='1' ; g=IN  ; s=N1
   and
        mos_p with g=CB  ; s=N1  ; d=OUT
   and
        reduced N1
   and
        mos_n with d='0' ; g=IN  ; s=N2
   and
        mos_n with g=C   ; s=OUT ; d=N2
   and
        reduced N2
   and
            observed inverter with input=C ; output=CB
       or
            observed inverter with input=CB ; output=C
then
       update signal OUT with tag := tristate
   and
       build trst_1 with
            output   :=  OUT ;
            input    :=  IN ;
            command  :=  C,CB ;
            internal:=  N1,N2 ;
            vhdl     := { OUT <= not IN  when C = '1' else $Z;
                          };
            verilog := { assign OUT := (C=1)? ~IN: $Z;
                          }
end
```

4.14 Les contrôles sur l'affectation des signaux

Nous avons introduit en fin d'abstraction une procédure qui permet de détecter certains cas d'incohérence en vue d'éditer des messages d'erreur ou de simple avertissement comme suit:

- signaux référencés non affectés => erreur
- signaux affectés non référencés => avertissement
- signaux non référencés et non affectés => avertissement

Pour cela nous définissons les signaux suivants:

1. les signaux référencés sont ceux qui:
   - appartiennent à l'attribut *command* d'au moins une Unité Abstraite
   - appartiennent à l'attribut *input* d'au moins une Unité Abstraite
   - sont associés à un connecteur de sortie ou d'entrée-sortie du bloc abstrait
2. les signaux affectés sont ceux qui:
   - appartiennent à l'attribut *output* d'au moins une Unité Abstraite
   - sont associés à un connecteur d'entrée ou d'entrée-sortie du bloc abstrait

Pour réaliser ces vérifications, nous avons associé à chaque objet signal 2 indicateurs booléens *bAssigned* et *bReferenced* positionnés initialement à *faux*. Pour évaluer l'état des signaux, on parcourt en fin d'abstraction toutes les Unités Abstraites, et pour chaque Unité Abstraite, on parcourt tous les signaux associés aux attributs *command*, *input*, *output* de ces Unités Abstraites, afin de positionner l'un ou l'autre des ces 2 indicateurs selon l'attribut qui fait référence au signal en appliquant les définitions 1 et 2 précédentes.

A partir de là, en parcourant tous les signaux, on regarde l'état de ces indicateurs et on émet des messages d'avertissement pour tous les signaux qui sont non référencés et ceux qui sont non référencés et non affectés et des messages d'erreur pour tous les signaux qui sont non affectés.

4.15 Conclusion

Nous voulions un procédé d'abstraction qui puisse être facilement configuré en fonction d'une cible technologique, qui puisse accepter tous les schémas de circuit possibles, et surtout qui permette d'exprimer facilement sous une forme compacte et évolutive les comportements Vhdl et Verilog de ces circuits. Nous avons donc choisi de faire reposer notre application sur des principes de reconnaissance de forme.

Nous avons élaboré pour cela un langage simple, concis, et spécialement dédié à la description des circuits électroniques. Ce langage permet de configurer extérieurement l'abstracteur, et son principal mérite est de permettre une description structurelle des schémas à reconnaître très proche des *net-list* hiérarchiques habituellement manipulées par les concepteurs de circuits, tout en fournissant des opérateurs permettant une description paramétrée (opérateurs *or*, *not*,...).

Le langage GRL que nous avons spécifié permet donc:

- de décrire n'importe quel type de circuit électronique dans n'importe quelle technologie Mos ou Bipolaire,
- de décrire des structures génériques telles que par exemple les multiplexeurs à n entrées,
- de décrire de façon concise des structures équivalentes (utilisation de l'opérateur *or*),
- de décrire des règles qui sachent prendre en compte le contexte du circuit à reconnaître (opérateurs *reduced*, *not, observed*),
- d'associer des comportements Vhdl et Verilog à chaque circuit reconnu,
- de vectoriser les comportements Vhdl et Verilog obtenus pour les rendre encore plus compacts.

5. Réalisation logicielle

Les principes de l’abstracteur que nous venons de décrire ont donné naissance au logiciel **GROG (Generalized Recognition Of Gates)** chez Bull [GrogFsd91], [GrogHldd91], [GrogUg97].

Nous allons maintenant décrire dans ce chapitre les principales fonctions logicielles et structures de données qui sont à la base de la réalisation de Grog.

5.1 Organisation générale du programme

L’abstraction est réalisée grâce à deux programmes exécutables qui sont:

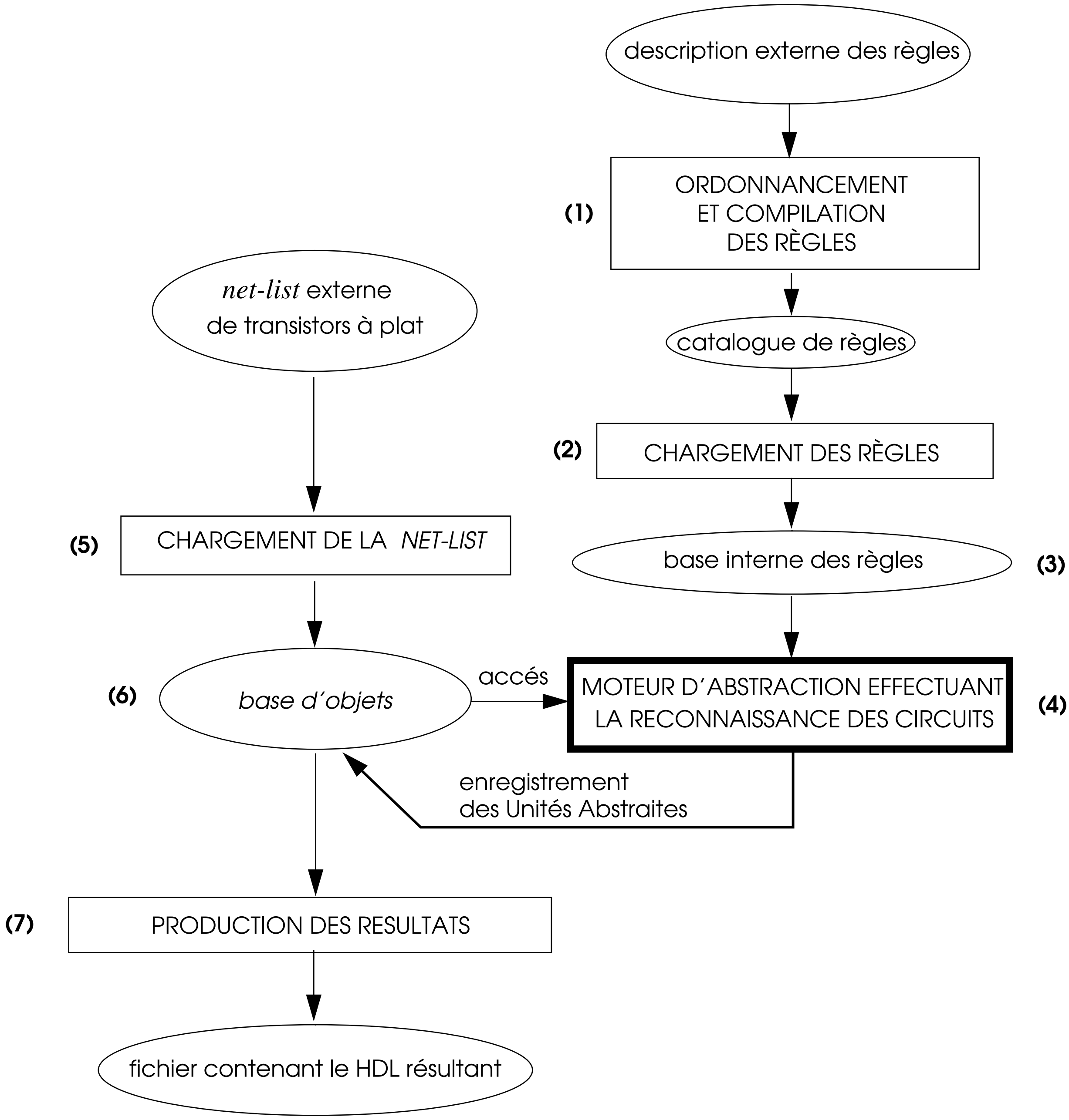
- le compilateur de règles
- le programme d’abstraction.

Ces programmes se décomposent eux mêmes en plusieurs unités appelées modules que nous allons décrire en faisant référence aux fonctions correspondantes de la Figure 5-1. Cette figure illustre l’agencement des principaux constituants de l’abstracteur, auxquels nous allons faire référence au travers des indices (1) à (7).

Pour avoir une bonne organisation du programme d’abstraction, et pour associer cette organisation à une gestion efficace des données, nous avons décidé de mettre en oeuvre une programmation de type **orientée objet**. Pour des raisons d’efficacité et de portage, c’est le langage **C++** [Gnuc++90] qui a été retenu pour réaliser ce programme.

C’est grâce à l’approche objet que nous avons pu définir facilement l’organisation des données en mettant en oeuvre un graphe d’héritage de **classes**. A partir de là, chaque objet informatique qui est traité par l’abstracteur appartient à une classe terminale dont il est l’**instance** (une classe terminale est une feuille de la hiérarchie des classes). Il devient ainsi possible de regrouper avec chaque objet les opérations qui lui sont associées, et grâce aux mécanismes d’héritage entre classes, de factoriser toutes les opérations qui sont communes à plusieurs type d’objets. Ceci nous permet de spécifier distinctement dans l’arbre d’héritage de classes tous les objets de l’abstraction avec leurs relations. Nous allons voir dans le reste de ce chapitre comment sont effectivement rangés et manipulés au travers de la même organisation hiérarchisée l’ensemble des objets signaux, primitives et Unités Abstraites.

Figure 5-1 schéma fonctionnel de l'abstraction.



5.2 Le module de compilation des règles

La compilation des règles est réalisée par un module logiciel qui se présente comme un exécutable distinct du reste de l'application.

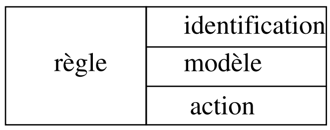
Ce module réalise la fonction (1) de la Figure 5-1. Le compilateur de règles construit un catalogue de règles sous une forme binaire à partir d'un fichier de règles décrites dans le langage GRL. Il a pour rôle de vérifier au préalable la syntaxe des règles, puis d'analyser ensuite les dépendances entre les règles pour les ordonner.

5.3 Le module de chargement des règles

Il réalise la fonction (2) de la Figure 5-1. Il permet de lire le catalogue de règles et de charger les structures de règles dans la base de règles. Les règles sont disposées sous la forme d'une liste d'instances d'un modèle *règle* et peuvent être directement interprétées par l'abstracteur. Nous donnons ici quelques indications sur la façon dont sont constituées les structures internes essentielles des règles.

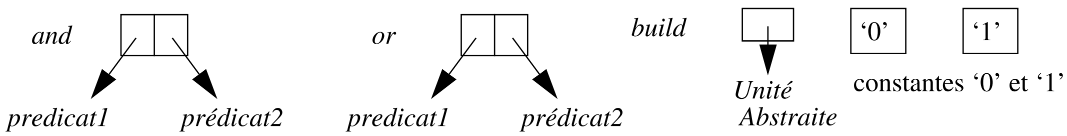
Un objet *règle* possède un champ *identification*, un champ *modèle* et un champ *action*, comme cela est décrit en Figure 5-2.

Figure 5-2 structure décrivant une règle



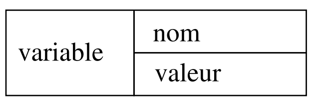
Les champs *modèle* et *action* pointent vers des structures qui sont respectivement *{and,or}* et *{and}* qui sont toutes des structures à deux opérandes, les feuilles de ces structures étant les prédicats. Il existe aussi des structures à un seul opérande comme *build*, et des éléments terminaux tels que des adresses d'objets signaux, les constantes '0' et '1', des chaînes de caractères constantes, ou encore des variables. C'est ce que montre la figure Figure 5-3 suivante:

Figure 5-3 structures utilisées par les règles



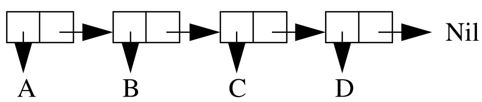
L'objet *variable* possède un champ *nom* qui est destiné à identifier la variable lors des éditions de trace et de mise au point, et il a un champ *valeur* qui est appelé à contenir la valeur (constante ou adresse) de la variable.

Figure 5-4 objet variable.



Une structure de liste est constituée par une cellule qui fait référence à une valeur ou à une variable et à une cellule suivante ou à Nil. Une structure de liste sert à regrouper plusieurs valeurs dans un champ d'objet. C'est ce que montre l'exemple de la Figure 5-5:

Figure 5-5 structure de liste

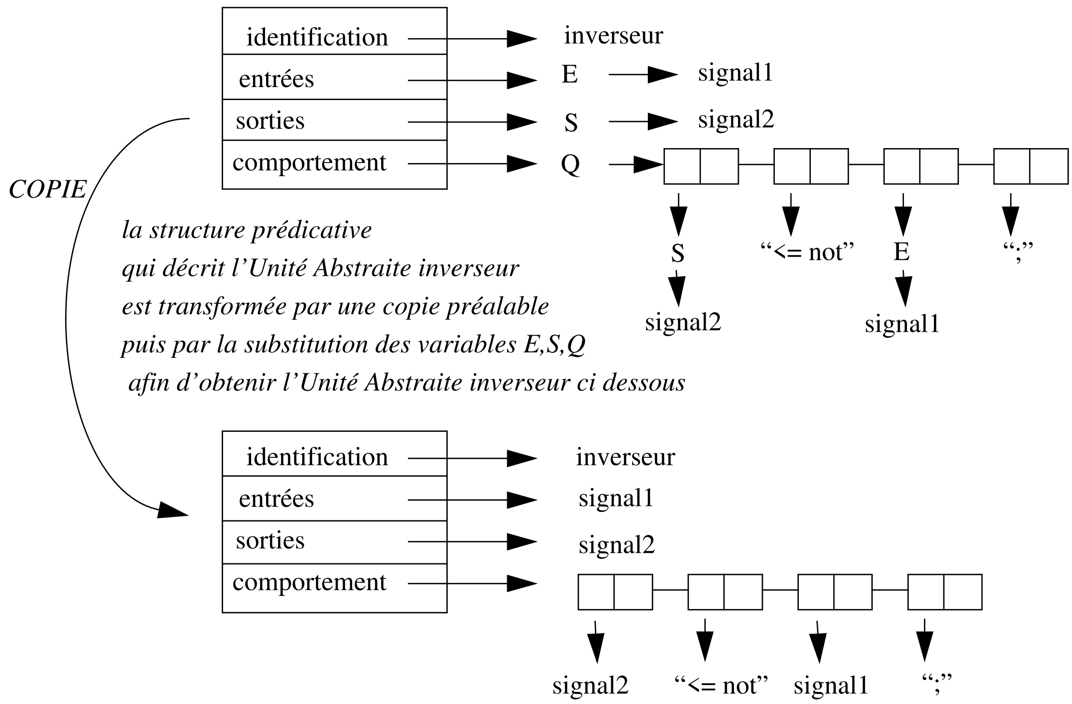


De la même façon que nous avons auparavant décrit la sémantique attachée à chaque objet de type composant ou signal (cf chapitre 4.2 page 29), nous pouvons à nouveau réutiliser cette même sémantique pour décrire les structures des prédicats qui vont servir à modéliser ces objets. Il suffit alors, pour réaliser le modèle d'un composant ou d'un signal, de prendre le même type de structure que celui qui décrit ces objet lorsqu'ils sont crées à partir de la *net-list*, mais cette fois en faisant référence à des variables à la place des adresses ou constantes dans les attributs du prédicat. Le mécanisme de filtrage peut alors opérer directement en tentant d'unifier deux objets de structures identiques, l'un de ces objets étant le prédicat dont les champs font référence à des variables, et l'autre étant l'objet signal ou composant dont les champs font référence à des adresses ou constantes.

A partir de là, toute construction d'Unité Abstraite consistera simplement lors du filtrage à recopier la structure prédicative de l'Unité Abstraite en remplaçant les variables du prédicat par les valeurs (constante ou adresse) reçues par ces variables dans l'objet Unité Abstraite nouvellement construit.

C'est ce que montre l'exemple suivant de la Figure 5-6 qui illustre la construction d'une Unité Abstraite inverseur à partir de la structure prédicative qui la modélise.

Figure 5-6 : construction d'une Unité Abstraite à partir d'une structure prédicative



- E,S,Q sont des variables
- signal1, signal2 sont les adresses des objets signaux de la net-list
- "<= not" et ";" sont des chaînes de caractères

5.4 Le module qui permet d'accéder à la base de règles

Il réalise la fonction (3) de la Figure 5-1. Il permet l'accès en séquence aux règles par le moteur d'abstraction, les règles étant tout simplement disposées sous forme d'une liste à parcourir. Nous ne le détaillerons donc pas davantage.

5.5 Le module qui réalise le moteur d'abstraction

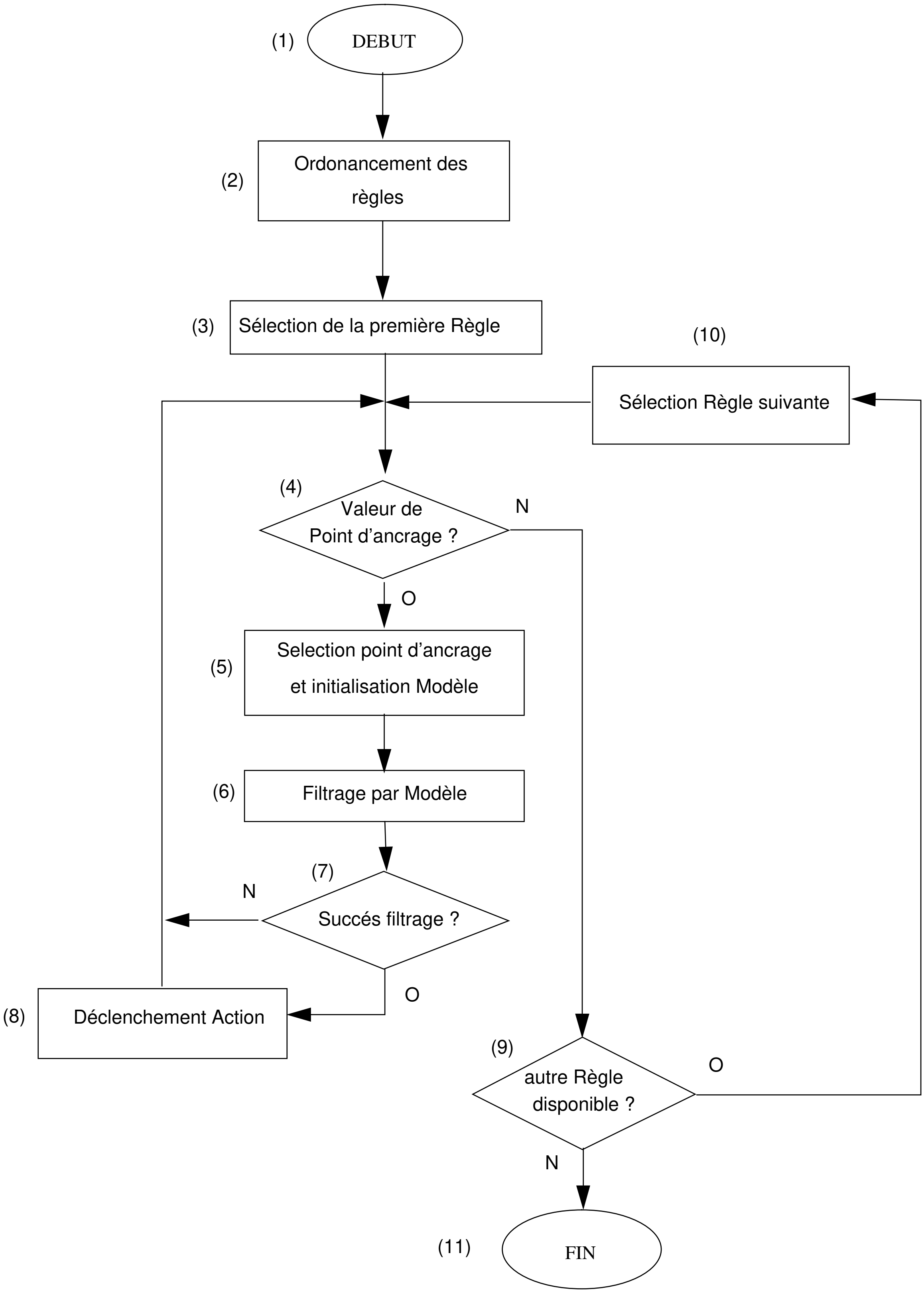
Il réalise la fonction (4) de la Figure 5-1. Pour cela, il lit les structures des règles et réalise la reconnaissance des circuits en confrontant les modèles des règles avec les objets. Il enregistre ensuite les Unités Abstraites dans la base d'objets.

Le mode de fonctionnement de l'abstraction fait que chaque modèle est confronté avec un ensemble d'objets de la base d'objets jusqu'à ce qu'il ne reste plus d'objet qui soit susceptible de satisfaire ce modèle.

Le fonctionnement du moteur d'abstraction complet est représenté par la Figure 5-7 et se décrit comme suit:

- l'abstraction débute en (1),
- on ordonne les règles en (2),
- on considère d'abord la première règle en (3), laquelle sert à initialiser le processus d'abstraction,
- on peut chercher à résoudre la règle à partir d'une première valeur de point d'ancrage s'il en existe une (4),
- dés qu'une valeur de point d'ancrage est trouvée (5), on initialise le modèle avec les valeurs initiales obtenues,
- on cherche ensuite à recueillir par filtrage un ensemble de valeurs qui satisfasse le modèle présenté en (6),
- si le filtrage réussit d'après le test (7), on exécute la partie Action en (8) avant de revenir chercher s'il y a une autre valeur de point d'ancrage disponible en (4) pour recommencer le cycle correspondant,
- si le filtrage échoue d'après le test (7), on revient directement chercher s'il y a une autre valeur disponible de point d'ancrage en (4),
- lorsqu'il n'y a plus de valeur de point d'ancrage disponible en (4), ce qui fait échouer le test correspondant, on regarde s'il y a une règle disponible (9),
- s'il y a une règle disponible dans la base de règles (9), on la sélectionne en (10), et on recommence en (4),
- sinon, s'il n'y a plus de règle disponible dans la base de règles (9), on arrête le processus d'abstraction (11).

Figure 5-7 fonctionnement du moteur d'abstraction



5.6 Le module de chargement de la ‘*net-list*’

Il réalise la fonction (5) de la Figure 5-1. Il permet de charger initialement la base d’objets à partir de la lecture d’une *net-list* de transistors et de signaux d’un bloc à abstraire.

Les formats de *net-list* peuvent être du type *fdn* format interne à *Bull*, du type *fne, fns,nle,nls* de *Compass* ou encore *cdl* de *Cadence*. Des contrôles syntaxiques sont réalisés à ce niveau pour vérifier la conformité de la description vis à vis de la spécification du format.

Quelques erreurs élémentaires sont détectées également, telles que par exemple celles dues à des grilles de transistor isolées. De simples avertissement signalent des particularités comme par exemple les transistors Mos qui ont un de leur connecteur de drain ou source isolé, ceci ne constituant pas une erreur dans le cas de circuits ROM.

Remarque: Il est à noter à cette occasion que l’utilisateur peut aussi réaliser certains contrôles supplémentaires sur le contenu de la *net-list* grâce aux règles d’abstraction. En effet, il est très facile de détecter dans la partie modèle une configuration anormale de connectique, d’émettre un message au moyen de l’opérateur *echo* dans la partie action, et même de retirer en même temps les éléments fautifs de la *net-list* en les transformant en Unités Abstraites de type *erreur*.

5.7 Le module qui réalise la base d’objets

Ce module réalise la fonction (6) de la Figure 5-1. La base d’objets met judicieusement à profit l’utilisation du langage C++, comme nous l’avons argumenté précédement dans le sous-chapitre 5.1, page 63. Cette base d’objets est comme nous allons le voir, conçue pour permettre un accès efficace et rapide du programme aux objets tout au long du processus d’abstraction.

Trois types de transactions avec la base d’objets sont possibles lors de l’abstraction qui sont:

- la lecture,
- la création des Unités Abstraites,
- la suppression par marquage des primitives ou Unités Abstraites qui composent une Unité Abstraite reconnue.

Les objets composants peuvent être accédés directement dans la base d’objets à partir d’informations sur leur connectique. A partir d’un objet signal, un mécanisme d’indexation permet en effet d’accéder à tous les composants qui partagent ce même signal.

Supposons par exemple que l’on veuille accéder à un composant Mos décrit dans le modèle par la structure prédicative suivante:

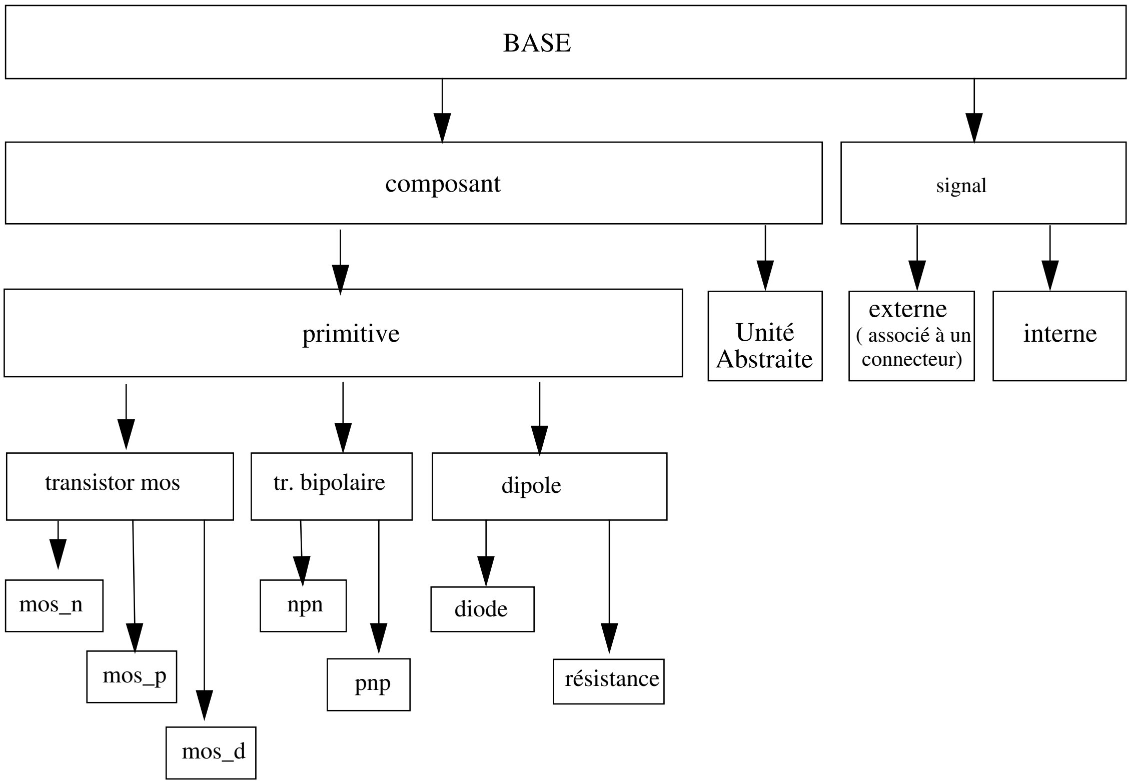
```
mos_n with g=G ; d=D ; s=S
```

Si au moins une des variables citées, prenons par exemple *D,* a été précédemment unifiée à un signal*,* alors l’abstracteur cherchera des *mos_n* seulement dans le sous ensemble de composants *Mos N* qui est associé au signal filtré par *D*. Comme il se produit souvent que plusieurs valeurs d’index soient candidates pour accéder à un ensemble de composants, c’est alors l’arité des signaux, c’est à dire le nombre de composants connectés à chaque signal qui sera mise à profit pour choisir le meilleur signal index possible. Le programme sélectionnera ainsi le signal qui aura l'index de plus faible arité pour choisir parmi les signaux d'indexation candidats celui qui est connecté au plus petit nombre de composants. L'abstracteur travaillera ensuite, pour la structure prédicative considérée, sur le seul sous ensemble de composants qui est associé à ce signal.

Il se peut aussi, lors de l'activation d'un prédicat, qu'aucune valeur d'index ne se présente, ce qui signifie que le prédicat doit obligatoirement être un point d'ancrage (premier prédicat d'accès à un composant référencé dans le modèle) comme cela a été spécifié (cf chapitre 4.5.2 page 40). Sans quoi cela constituerait un cas d'erreur, et ce serait signalé comme tel par le programme. Il est à noter que tandis que les primitives restent inchangées tout au long du programme, il n'en est pas de même des Unités Abstraites. C'est pourquoi les références d'indexation des signaux sont mises à jour tout au long de la phase d'abstraction, chaque fois que de nouvelles Unités Abstraites sont construites.

Chaque description de classe contient un nombre déterminé de champs nommés appelés attributs dont héritent les objets. Toute information de connectique, de topologie ou de comportement est ainsi rendue nominativement accessible. Chaque classe possède aussi un ensemble de fonctions dites méthodes qui permettent au programme d'abstraction de modifier et manipuler les objets qui en sont les instances. La base de données interne du programme est décrite sous une forme hiérarchique comme illustré en Figure 5-8, dans laquelle on voit que toutes les classes dérivent d'une même classe appelée *BASE.* Cette classe *BASE* permet de factoriser au plus haut niveau les attributs et méthodes communs à toutes les classes dérivées.

Figure 5-8 organisation générale des classes



Le *type* d'un objet est le nom de la classe terminale à laquelle il appartient. C'est ainsi, par exemple, que lors du chargement de la *net-list* dans la base d'objets, on crée l'objet transistor Mos N dont le type sera *mos_n*, qui appartient donc à la classe terminale *mos_n.*

L'objet *transistor* ainsi créé possède des attributs de *grille*, *source* et *drain* qui retiennent les informations de connectique correspondantes, des attributs *x* et *y* qui contiennent les coordonnées, et des attributs *w* et *l* qui contiennent les largeur et longueur de grille.

Les classes *transistor*, *dipôle* sont regroupées sous une classe nommée *primitive*. Les classes *primitive* et *Unité Abstraite* sont elles mêmes regroupées sous une même classe dite *composant*. Pour mieux montrer l'intérêt de cette organisation, on peut noter ici par exemple que les attributs x et y qui sont communs à tous les objets de type composant seront spécifiés au niveau de la classe *composant*.

La classe *composant* possède aussi en particulier un attribut *utilisé* dont nous avons déjà parlé (Remarque du chapitre 4.2.3, page 32), et qui sert à indiquer si le composant fait partie d'une Unité Abstraite. Les objets des classes *transistor* et *Unité Abstraite* héritent là aussi automatiquement de cet attribut.

5.8 Le module de vectorisation et production des résultats.

Il réalise essentiellement la fonction (7) de la Figure 5-1. Cette fonction permet d'obtenir les comportements HDL qui résultent de l'abstraction d'un bloc. Elle effectue pour cela une phase préalable de vectorisation des signaux et des comportements, qui est suivie ensuite d'une phase de regroupement et d'édition des résultats.

La fonction d'édition HDL sert à constituer un fichier Vhdl ou Verilog tel que celui ci soit directement prêt à être utilisé par des logiciels de simulation, de preuve formelle ou de synthèse.

C'est ainsi que les fichiers HDL qui sont produits contiennent une description d'interface et un corps de description. Un en-tête peut être inséré pour faire référence aux catalogues nécessaires comme c'est le cas par exemple avec les bibliothèques et paquetages Vhdl. Le corps de description est ensuite réalisé à partir des comportements individuels des circuits reconnus qui sont collectés. Des commentaires seront insérés dans le HDL produit pour documenter le comportement résultant et aider ainsi à son interprétation textuelle.

Dans le cas par exemple du fichier Vhdl de la Figure 5-9, la fonction d'édition vient insérer un en-tête de référence aux bibliothèques et paquetages Vhdl. Ces descriptions d'en-tête sont incluses à partir d'un fichier externe préalablement cité sur la ligne de commande du logiciel. Par défaut, c'est le catalogue *IEEE.STD_LOGIG_1164* qui est utilisé.

La fonction d'édition décrit ensuite l'*entity* Vhdl en donnant les orientations correctes des connecteurs d'entrée sortie du bloc. Ces orientations sont évaluées à partir des affectations des signaux concernés dans les attributs *input*, *command* et *output* des Unités Abstraites Grog.

Cette fonction d'édition exprime enfin dans la partie *architecture* le comportement du bloc en venant écrire les comportements Vhdl associés à chaque Unité Abstraite Grog (circuit reconnu).

De cette façon le fichier HDL obtenu est prêt à l'emploi, et il peut être immédiatement compilé.

Remarque: Il est à noter que la méthode que nous avons employée donne une totale transparence au procédé d'abstraction. En effet, dans le fichier HDL produit, on vient placer les comportements de chaque type de circuit sous une même bannière qui donne le nom de l'Unité Abstraite et le nombre d'instances trouvées.

La Figure 5-9 donne un exemple de Vhdl remonté

Figure 5-9 Exemple de fichier Vhdl obtenu par abstraction

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
--
ENTITY mmdcq_t3 IS
PORT(
    da : in std_logic ;
    db : in std_logic ;
    sa : in std_logic ;
    q : out std_logic ;
    cp : in std_logic ;
    cdn : in std_logic
 ) ;
END mmdcq_t3;

ARCHITECTURE abstracted OF mmdcq_t3 IS
SIGNAL gnum17 : std_logic ;
SIGNAL gnum19 : std_logic ;
SIGNAL la1 : std_logic ;
SIGNAL gnum21 : std_logic ;
SIGNAL lal1 : std_logic ;
SIGNAL gnum23 : std_logic ;
SIGNAL la2 : std_logic ;
SIGNAL lal2 : std_logic ;

BEGIN

-- abstraction of 'inverter' (nb instances = 4) --

q <= not la2;
gnum17 <= not sa;
gnum23 <= not cp;
gnum21 <= not gnum23;

-- abstraction of 'mos_dual_arn22' (nb instances = 1) --

 gnum19 <= not ((da and sa) or (db and gnum17));

-- abstraction of 'latch_asy' (nb instances = 2) --

process(gnum19,gnum23)
begin
if(gnum23='1') then la1 <= not gnum19; end if;
end process;
lal1 <= not la1;

process(cdn, la1,gnum21)
begin
if(cdn='0') then la2 <= '1'; end if;
if(gnum21='1') then la2 <= not la1; end if;
end process;
lal2 <= not la2;

END abstracted;
```

6. Tests et résultats

Nous décrivons dans ce chapitre les tests qui, grâce au logiciel Grog, ont permis de valider sur des circuits industriels les concepts que nous avons introduits dans cette étude. Nous commentons également dans ce chapitre les résultats que nous avons obtenus.

Les tests effectués permettent d'évaluer les performances du logiciel à partir d'applications de taille significative, et de mettre aussi en évidence certaines caractéristiques de la méthode d'abstraction.

Ces tests ont tous été effectués sur une station UNIX de type *Bull-DPX20* avec un processeur de 62MHz, 131 MIPS, et ayant 128MO de mémoire centrale.

6.1 Tests d'efficacité en temps de calcul

L'objectif de ces tests est de comparer les temps d'abstraction obtenus à partir de circuits de même structure et de tailles croissantes pour évaluer à la fois la linéarité et l'efficacité du procédé en termes de temps de calcul.

Les tests ont ainsi été effectués sur plusieurs tailles de blocs FIFO et RAM et se sont déroulés de la façon suivante:

a. Nous avons tout d'abord réalisé l'abstraction d'un ensemble de 5 circuits FIFO de tailles comprises entre 8 et 512 mots de 9 bits. Ces 5 circuits FIFO sont tous issus d'un projet de composant VLSI asynchrone nommé *KLOE*. Les résultats des tests sont donnés en Table 6-1,

b. Nous avons de la même façon réalisé l'abstraction d'un ensemble de 5 circuits de mémoires RAM de tailles croissantes. De telle mémoires sont utilisées dans un projet d'ordinateurs de type DPS8 de *Bull*. Les résultats obtenus sont donnés en Table 6-2,

c. A partir de ces données, nous avons exprimé sous une forme graphique les résultats comparatifs des temps d'abstraction en fonction du nombre de transistors pour les deux types de circuits FIFO et RAM. On constate d'après la Figure 6-1 que le temps d'abstraction des circuits pour des tailles usuelles allant jusqu'à 50000 transistors, et pour un ensemble de circuits de même type, peut être considéré comme quasiment linéaire en fonction du nombre de transistors. Au delà, on perd en linéarité, mais les temps d'abstraction obtenus pour des blocs de 80000 transistors, qui représentent 15 minutes de *cpu* restent cependant très acceptables.

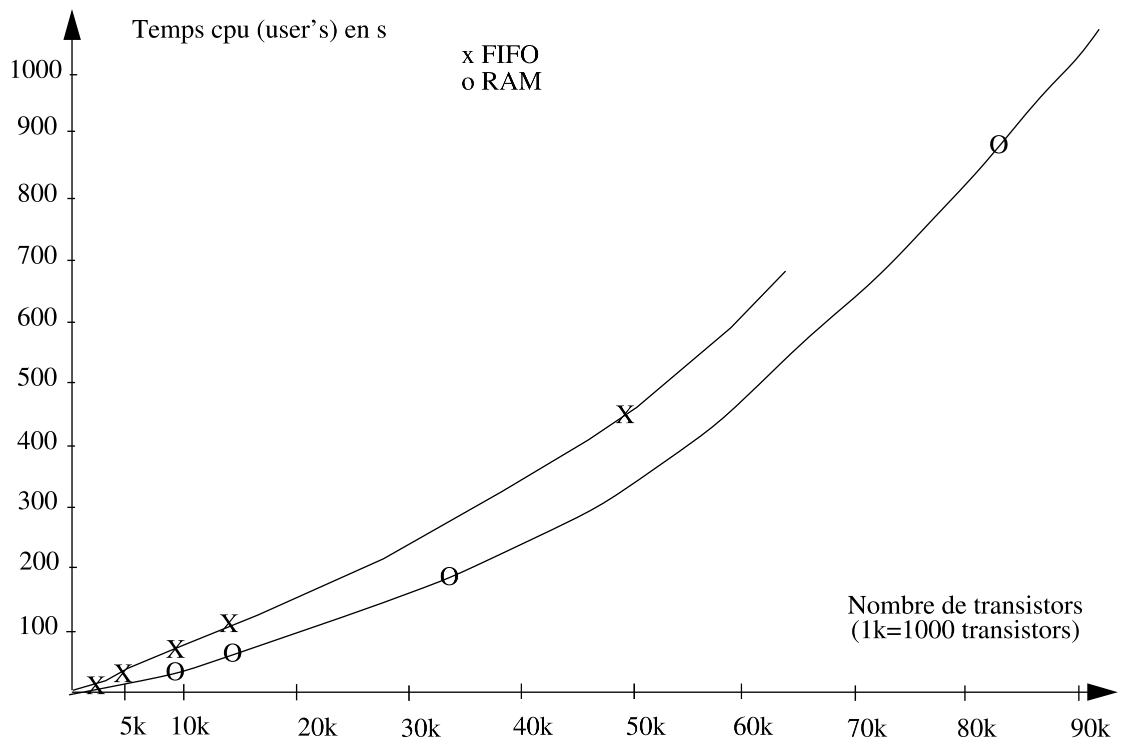
Table 6-1 abstraction de circuits FIFO

désignation	nombre de mots de 9 bits	nombre de transistors	temps total d'abstraction (utilisateur en s)
FIFO-0	8 mots	1587	8,4
FIFO-1	32 mots	4830	32
FIFO-2	64 mots	8661	58
FIFO-3	128 mots	13785	100
FIFO-4	512 mots	49217	420

Table 6-2 abstraction de circuits RAM

désignation	nombre de bits	nombre de transistors	temps total d'abstraction (utilisateur en s)
RAM-0	1024 bits	7634	24
RAM-1	2048 bits	13948	52
RAM-2	4096 bits	26576	131
RAM-3	5120 bits	32890	193
RAM-4	13312 bits	83402	895

Figure 6-1 graphique du temps d'abstraction des circuits FIFO et RAM



d. Nous voulons également estimer l'importance relative de chacune des activités du programme d'abstraction. A partir d'une analyse plus détaillée des essais précédents, nous pouvons caractériser les temps passés dans les principales parties du programme. Les tables référencées Table 6-3 et Table 6-4 donnent les temps passés pour effectuer la lecture et le chargement des règles, ceux de la *net-list*, l'abstraction proprement dite, et l'édition du Vhdl. On constate ainsi que le temps dédié à l'abstraction seule est le plus important, et que vient ensuite le temps de la phase dite de chargement de la *net-list*. L'importance relative de ce dernier temps est justifiée par le fait que c'est lors de cette phase que sont mises en place les structures internes de la base d'objets et que sont réalisés les liens entre les objets dans les structures internes du programme. Le temps de chargement des règles est constant puisqu'il s'agit pour tous les essais du même catalogue de règles.

Table 6-3 répartition des temps obtenus sur les blocs de type FIFO et pourcentage / temps total

blocs FIFO	FIFO0	FIFO1	FIFO2	FIFO3	FIFO4
Chargement Netlist	1s *(12%)*	2s *(6%)*	3,6s *(6%)*	6s *(6%)*	21s *(5%)*
Chargement règles	2s	2s	2s	2s	2s
Abstraction	5,1s *(61%)*	26,2s *(82%)*	51,2s *(89%)*	89,8s *(90%)*	391,5s *(93%)*
Editions du Vhdl	0,1s *(1,2%)*	0,4s *(1,2%)*	0,8s *(1%)*	1,4s *(1%)*	5,3s *(1,2%)*
Autres	0,2s	0,2s	0,2s	0,2s	0,2s
TOTAL	8,4s	31.8s	57.8s	99.4s	420s

Table 6-4 répartition des temps obtenus sur les blocs de type RAM et pourcentage / temps total

blocs RAM	RAM0	RAM1	RAM2	RAM3	RAM4
Chargement Netlist	3,3s *(13.7%)*	6s *(11.5%)*	11,8s *(9%)*	14,6s *(7.5%)*	38,5s *(4.3%)*
Chargement règles	2s	2s	2s	2s	2s
Abstraction	17,7s *(73,8%)*	42s *(80%)*	113,5s *(86,8%)*	173s *(89%)*	845,4s *(94,4%)*
Editions du Vhdl	0,8s *(3,3%)*	1,5s *(2,8%)*	3,1s *(2,4%)*	3,7s *(1,9%)*	9s *(1%)*
Autres	0,2s	0,2s	0,3s	0,3s	0,3s
TOTAL	24s	52,3s	130.7s	193.6s	895.2s

Table 6-5 importance relative des activités du processus d'abstraction sur blocs FIFO et RAM

opérations	pourcentage du temps total
Chargement de la 'net-list'	5 à 14 %
Abstraction	61 à 94 %
Edition Vhdl	1 à 3 %

6.2 Tests d'efficacité de la vectorisation

Nous savons que le problème qui se pose fréquemment avec la remontée d'un comportement de type flot de données est celui de la taille importante du volume de données obtenu. De trop gros volumes de code Vhdl ou Verilog peuvent être en effet difficiles à simuler ou à re-synthétiser, voir à interpréter visuellement. C'est pourquoi nous avons mis en oeuvre l'option vectorisation de *Grog* qui permet de diminuer de façon importante la taille du code Vhdl ou Verilog qui est produit. Nous avons voulu au travers des tests suivants évaluer l'efficacité de cette option.

Pour réaliser ces tests, nous avons traité trois blocs extraits du projet *ISL* de *Bull* qui possèdent de nombreuses structures de BUS et procédé aux évaluations suivantes:

a. Nous voulions d'abord évaluer le gain de taille qui peut être obtenu grâce à la vectorisation. Nous avons donc exprimé les tailles des fichiers Vhdl abstraits avec et sans l'option vectorisation de Grog. D'après les résultats de ces tests qui sont exprimés dans la Table 6-6, on constate que le gain de taille par rapport à un même fichier non vectorisé peut selon les cas atteindre 23 à 76%,

b. Comme il est important de savoir si un tel procédé ne pénalise pas le temps d'exécution du programme, nous avons aussi relevé le temps passé pour la vectorisation qui, comme nous pouvons le constater, accroit respectivement de 12 à 30% le temps total de l'activité. Un tel surcoût en temps est tout à fait acceptable vis à vis des gains de taille obtenus,

Table 6-6 gains de taille obtenus lors de la vectorisation

référence circuit	taille (octets) du Vhdl sans vectorisation	taille (octets) du Vhdl avec vectorisation	réduction de taille	temps total sans vectorisation (en s)	temps total avec vectorisation (en s)	augmentation de temps due à la vectorisation
tr_fifo.vhd	32531	17376	47%	7	8	14%
tbuf_r.vhd	418526	322397	23%	49	55	12%
adm_s.vhd	2131758	510144	76%	268	345	30%

c. Nous voulons également que la vectorisation facilite l'interprétation visuelle du Vhdl abstrait en faisant apparaître les transferts entre les vecteurs de signaux. Les regroupements des signaux en vecteurs doivent faciliter leur analyse lors de la simulation fonctionnelle. C'est ce que nous pouvons constater à partir de l'extrait de code Vhdl de la Figure 6-2 qui présente un extrait de code Vhdl décrivant les comportements vectorisés de 3 tranches de latches remontées par *Grog* à partir du circuit *adm_s*. Sans la vectorisation, on aurait une description de *process* Vhdl par bit de signal, ce qui ici par exemple pour une tranche de 18 latches ferait 18 *process* au lieu d'un seul après vectorisation. Associée à notre méthode d'abstraction, la vectorisation constitue donc bien un avantage supplémentaire pour l'interprétation des résultats. Il est à noter aussi dans cet exemple que les vecteurs référencés par un nom de type *vnum* sont construits automatiquement par Grog, lors de l'étape de propagation de la vectorisation, et ceci en regroupant des signaux vectorisables qui initialement ne portaient pas un nom significatif dans la *netlist*, comme nous l'avons décrit dans le paragraphe 4.13, page 58.

Figure 6-2 : Vhdl obtenu à partir de la vectorisation de tranches de registres

```
 ......
process(adm_pioadr_l1(17 downto 0),vnum5(53 downto 36),gnum1358)
begin
if(gnum1358='1') then adm_pioadr_l1(17 downto 0) <= vnum5(53 downto 36);
end if;
end process;
vnum12(17 downto 0) <= not adm_pioadr_l1(17 downto 0);

process(adm_pioadr_l1(23 downto 18),vnum5(59 downto 54),gnum1358)
begin
if(gnum1358='1') then adm_pioadr_l1(23 downto 18) <= vnum5(59 downto 54);
end if;
end process;
vnum289(59 downto 54) <= not adm_pioadr_l1(23 downto 18);

process(adm_isl_nodenb_l1(7 downto 4),lpa_islnb_x1(7 downto 4),gnum1358,
vnum273(7 downto 4),gnum1782)
begin
if(gnum1358='1') then adm_isl_nodenb_l1(7 downto 4) <= lpa_islnb_x1(7 downto 4);
end if;
if(gnum1782='1') then adm_isl_nodenb_l1(7 downto 4) <= vnum273(7 downto 4);
end if;
end process;
vnum274(7 downto 4) <= not adm_isl_nodenb_l1(7 downto 4);

......
```

6.3 Tests d'adaptabilité du procédé

Nous voulions que notre méthode soit suffisamment générale pour être utilisée sur plusieurs projets de VLSI différents. Celle-ci a effectivement été utilisée en milieu industriel sur plusieurs projets, et nous donnons ici à titre d'information les caractéristiques très différentes des projets de VLSI de *Bull* sur lesquels Grog a été appliqué. Certains des résultats obtenus sur ces projets seront détaillés ultérieurement dans ce même chapitre, et vont nous permettre en particulier de démontrer la généralité du procédé.

- projet *KLOE*

  Ce projet avait pour objectif de réaliser pour un Institut de Physique Nucléaire un composant VLSI en technologie CMos 0.5 µm, fonctionnant en logique asynchrone à 1GHz pour effectuer du comptage de particules. Il était important à cet effet de savoir reconnaître des circuits complexes et hors de la méthodologie usuelle de Bull, et de pouvoir introduire dans le Vhdl remonté par *Grog* certains délais systématiques afin d'améliorer les simulations. Ce VLSI totalisait 500000 transistors, et il contenait en particulier 4 blocs FIFO de 16 mots de 32 bits. Chacun des blocs FIFO remontés par *Grog* représentait 2600 lignes de Vhdl. Grog a permis ici de remonter du Vhdl à partir de la *netlist* issue de la description des masques après extraction, et aussi de remonter du Vhdl à partir de descriptions schématiques réalisées sous COMPASS et alors exprimées sous forme de *netlists* de transistors. Les résultats de l'abstraction par *Grog* étaient ici dirigés vers un simulateur *Synopsys,* ce qui selon l'origine de la *netlist* permettait de vérifier soit la réalisation physique soit les schémas.

- projet *ISL*

  Ce projet consistait à intégrer des liaisons série haute vitesse 1GHz dans des composants VLSI devant servir de liens entre les noeuds d'un cluster de multiprocesseurs. *Grog* a été utilisé pour fournir un comportement Vhdl exploitable par des simulateurs, par des outils internes de preuve formelle (*fproof*), et pour là aussi, vérifier que les structures reconnues correspondaient bien aux spécifications. Des blocs de taille allant jusqu'à 80000 transistors ont pu ainsi être remontés facilement.

- projet *BULLIT*

  Ce projet consistait à concevoir les VLSI d'une liaison série haute vitesse à 1GHz en technologie CMos. *Grog* a été utilisé pour fournir un comportement Vhdl exploitable par des simulateurs et vérifier que les structures reconnues correspondaient bien aux schémas des circuits initialement spécifiés. Sur ce projet, Grog a été utilisé sur de petits blocs de quelques milliers de transistors.

- projet *RCUBE*

  Le circuit *RCUBE* (Rapide Reconfigurable Routeur) est un routeur de messages qui dispose de huit liens séries bidirectionnels interconnectés au moyen d'un “cross-bar” de dimension 8x8. Son architecture est entièrement parallèle, ce qui fait que chaque message présent sur l'une des huit entrées peut être dirigé indépendamment des autres messages vers l'une des huit sorties. Ici *Grog* a été utilisé pour fournir un comportement Vhdl exploitable par des simulateurs, et pour vérifier que les structures reconnues correspondaient bien aux spécifications. Le composant VLSI réalisé compte environ 700000 transistors, la complexité des blocs remontés allant de 2000 à 12000 transistors.

- projet EUROPRO

  Dans ce projet *[Europro]*, Bull doit réaliser en collaboration avec l'Université de Paris6 un routeur statique pour des applications de type temps réel. Le composant VLSI réalisé totalise environ 1,7 millions de transistors. Les tailles des blocs remontés par Grog vont de 13000 à 49000 transistors. Ces blocs ont la particularité là aussi de comporter plusieurs circuits FIFO synchrones de 128 mots de 9 bits. Grog est utilisé dans ce projet pour effectuer de la vérification conjointement avec le logiciel de preuve formelle *VFormal [Vformal].*

- projet TWIN (Twin Way Interconnect Network)

  Le composant qui est réalisé en collaboration avec l'Université de Paris6 dans le cadre de ce projet *[TWIN]* doit permettre de connecter efficacement des systèmes de traitement de données exigeant une bande passante importante et une faible latence. Ceci afin d'apporter une solution aux problèmes posés par les transferts de données dans les systèmes dits massivement parallèles, les serveurs, ou les systèmes de disques redondants. Le composant VLSI en cours de réalisation comporte environ 2 millions de transistors. La taille des blocs remontés va jusqu'à 50000 transistors. Là aussi, Grog est utilisé dans une chaîne de vérification conjointement avec le logiciel de preuve formelle *VFormal.*

6.4 Tests de généralité du procédé

Un de nos objectifs initiaux était d'obtenir une méthode d'abstraction suffisamment souple afin de s'adapter à l'abstraction de n'importe quels schémas de circuits qui seraient conçus dans n'importe laquelle des technologies micro-électroniques connues à ce jour et exploitées industriellement. C'est pourquoi nous allons montrer ici comment notre méthode d'abstraction a permis de s'adapter à la diversité des méthodes de conception et des technologies.

Tout d'abord, l'utilisation de *Grog* chez *Bull* devait permettre de remonter les comportements Vhdl des blocs pour les vérifier directement avec des outils de preuve formelle internes à *Bull*, ou avec des simulateurs tels que celui de *Synopsys*. A cause de la diversité des projets de VLSI, *Grog* a eu à traiter un grand nombre de modèles de circuits. Pour cela un ensemble de catalogues (un par projet) ont été constituées. Ces catalogues regroupaient en tout 329 règles, ce qui représente 7800 lignes de texte, certaines de ces règles étant alors partagées par plusieurs projets.

Les résultats obtenus à partir de ces catalogues de règles sur deux projets nommés *ISL* et *KLOE* sont cités ici à titre d'exemple. Les composants VLSI de ces projets rassemblent typiquement une grande diversité de circuits, et c'est pourquoi nous avons jugé utile de présenter ici quelques uns des résultats obtenus afin de montrer que l'on peut traiter dans une même abstraction une grande variété de circuits.

- reconnaissance d'éléments mémorisants

  Tous les types de circuits mémorisants peuvent facilement être reconnus et modélisés par Grog. Nous pouvons voir comment Grog sait exprimer de tels comportements à partir du bloc *tr_fifo* du projet *ISL* qui réalise une fonction de type FIFO (first in first out) et qui contient à ce titre une grande diversité d'éléments mémorisants.

  L'inventaire des circuits reconnus dans ce bloc est donné en Table 6-7, tandis que la liste de règles donnée en Figure 6-3 permet de reconnaître les éléments mémorisants cités. Pour avoir un descriptif détaillé de ces règles, on peut se reporter en ANNEXE D.

Table 6-7 inventaire des circuits reconnus dans le bloc *tr_fifo*

types de circuits reconnus	quantité
portes ‘et’ de type ’and2’	2
chaînes d’inverseurs de type ’even_nb_inv’	12
points mémoires de type ’reg_point_mem’	72
circuits échantilloneurs bloqueurs de type dit ’ech_bloqueur’	2
registres de types ‘basc_di’, ‘basc_dni’	18
registre de type ’register_full	1
registre de type ’register_empy’	1
circuits à précharge de type ’res_pr’	12
bleeders de type ’bleeder1’	12
portes ‘non ou’ de type ’nor3’	12
inverseurs de type ’inverter’	354

Figure 6-3 liste des règles de reconnaissance des éléments mémorisants du bloc *tr_fifo*

```
Règle “POINT MEMOIRE REGISTRE” pour ‘reg_point_mem’.
Règle “ECHANTILLONEUR BLOQUEUR” pour ‘ech_bloqueur’.
Règle “BASCULE DI” pour ‘basc_di’
Règle “BASCULE_DNI” pour ‘basc_dni’
```

Nous voyons que grâce à notre méthode, il est facile de spécifier précisement et de façon concise les comportements des circuits mémorisants reconnus, et dans le cas de plusieurs énoncés Vhdl *concurrents* dans un même *process*, qu’il est possible de définir avec exactitude leur ordre d’occurrence. C’est ce qu’illustrent les figures référencées <Figure 6-4, Figure 6-5, Figure 6-6 et Figure 6-7> dans lesquelles sont donnés des extraits de Vhdl obtenus à partir des 4 règles de la Figure 6-3.

Figure 6-4 Comportement Vhdl remonté par la règle *POINT MEMOIRE REGISTRE*

```
process(selw3,din(11 downto 0))
variable var : std_logic_vector (11 downto 0) ;
attribute svar of var : variable is latch ;
begin
if (selw3 = ’1’) then var(11 downto 0) := not din(11 downto 0) ;
end if ;
rdata3(11 downto 0) <= not var(11 downto 0) ;
end process ;
```

Figure 6-5 Comportement Vhdl remonté par la règle *ECHANTILLONNEUR BLOQUEUR*

```
process (gnum165)
begin
if(gnum165='1') then
 selr2 <= sr2 ;
 selr4 <= sr4 ;
 selr5 <= sr5 ;
 selr1 <= sr1 ;
 selr0 <= sr0 ;
 selr3 <= sr3 ;
else
 selr2 <= '0' ;
 selr4 <= '0' ;
 selr5 <= '0' ;
 selr1 <= '0' ;
 selr0 <= '0' ;
 selr3 <= '0' ;
end if ;
end process ;
```

Figure 6-6 Comportement Vhdl remonté par la règle *BASCULE DI*

```
process(gnum459)
begin
if (gnum459 = '0') then dout(11) <= vnum2(11) ; end if ;
end process ;
```

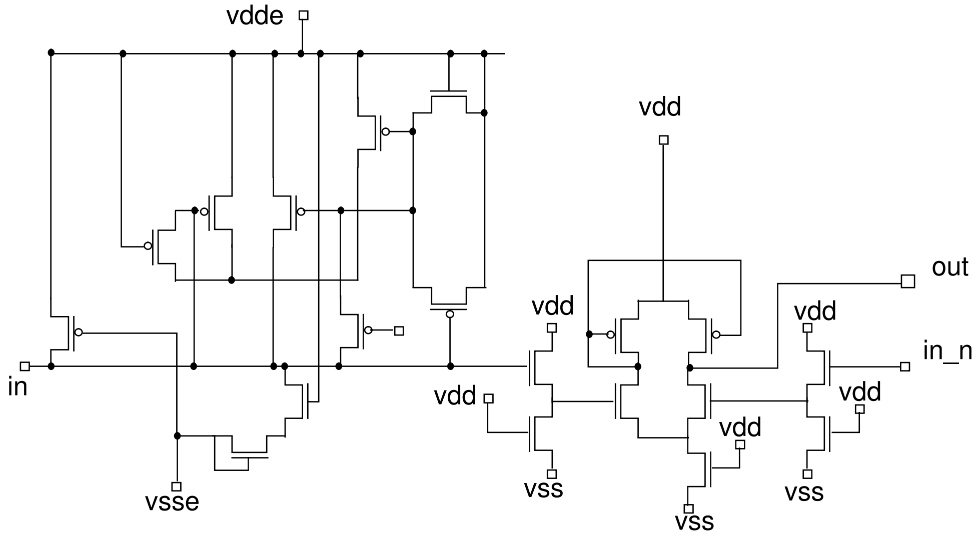
Figure 6-7 Comportement Vhdl remonté par la règle *BASCULE_DNI*

```
process(gnum171, selr4, selw3)
begin
if (selw3 = '1' and (gnum171 = '1' or selr4 = '1')) then rp3 <= 'X' ;
elsif (gnum171 = '1') then rp3 <= '0' ;
elsif (selw3 = '1') then rp3 <= '1' ;
elsif (selr4 = '1') then rp3 <= '0' ;
end if ;
end process ;
rp3r <= not rp3 ;
```

- Reconnaissance de circuits de protection

  Toujours pour montrer la souplesse de notre méthode d'abstraction, et son aptitude à traiter des circuits complexes, nous donnons à titre d'exemple en Figure 6-8 le schéma de l'un des nombreux circuits de protection et d'adaptation de niveaux de tension qui sont associés aux plots d'interface des VLSI CMos. Nous n'analyserons pas en détail ces circuits, si ce n'est pour dire qu'ils mettent en oeuvre de nombreuses interactions électriques, tandis que leur comportement se résume le plus souvent, du point de vue logique, à une simple équation de transfert entre la sortie et l'entrée. L'exemple donné montre surtout comment grâce à notre techniques de reconnaissance à partir de modèles, nous sommes à même d'identifier rapidement de tels circuits sans faire intervenir de procédure d'analyse particulière, puisqu'il suffit d'en décrire le schéma au moyen d'une règle d'abstraction. Une telle opération nécessiterait autrement, avec d'autres procédés d'abstraction, de mettre en oeuvre des algorithmes spécifiques pour effectuer ponctuellement l'analyse électrique détaillée du circuit.

Figure 6-8 exemple de circuit de protection



- Abstraction de circuits asynchrones

  La particularité du VLSI du projet *KLOE* est d'être complètement asynchrone, et nous allons voir comment l'on a pu grâce au langage de règles faire apparaître facilement des instructions de type *after* dans le Vhdl résultant pour exprimer des délais temporels. Grog permet en effet de spécifier des éléments de retard pour que ceux ci puissent être pris en compte par la simulation, ce qui montre aussi que *Grog* permet d'adapter exactement le comportement Vhdl ou Verilog aux modes de vérification envisagés: simulation avec prise en compte d'éléments de retard pour les circuits asynchrones, ou preuve formelle dans la cas de circuits synchrones [SynchroD92].

  Le type de comportement produit par *Grog* en Figure 6-11, correspond au schéma de la Figure 6-9 qui sert à produire une impulsion sur 2ns après un front montant. Ce comportement a pu être remonté grâce à la règle ”CHAINE RET_LEC RESET ASYNCHRONE” de la Figure 6-10.

Figure 6-9 schéma d'un générateur d'impulsion à chaîne de retard

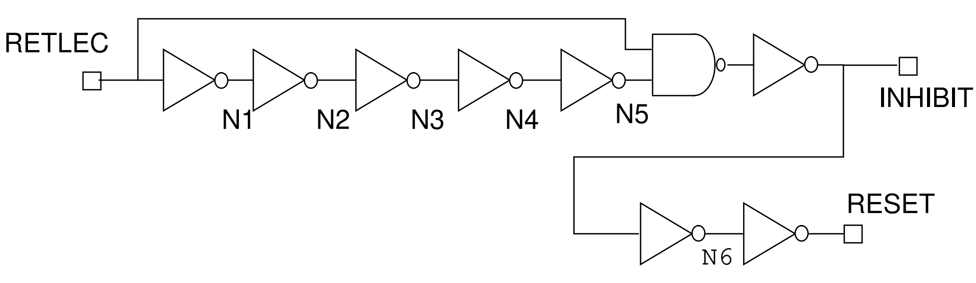


Figure 6-10 règle d'abstraction du générateur d'impulsion à chaîne de retard

```
-> "CHAINE RET_LEC RESET ASYNCHRONE"
 if
       and_equiv with input = I1,I2 ; output = INHIBIT
   and
       I1,I2 ?= N5 ,RETLEC
   and
       inverter with input = RETLEC ; output = N1
   and
       reduced N1
   and
       inverter with input = N1 ; output = N2
   and
       reduced N2
   and
       inverter with input = N2 ; output = N3
   and
       reduced N3
   and
       inverter with input = N3 ; output = N4
   and
       reduced N4
   and
       inverter with input = N4 ; output = N5
   and
       reduced N5
   and
       inverter with input = INHIBIT ; output = N6
   and
       inverter with input = N6 ; output = RESET
   and
       reduced N6
   then
       build ret_lec_reset with
            input := RETLEC,APRES ;
            command := '0','1' ;
            output := INHIBIT, RESET ;
            vhdl :=
            {
                 process(RETLEC)
                 begin
                 if (RETLEC = '1') then
                 INHIBIT <= `1' after 0 ns , '0' after 2 ns ;
                 end if ;
                 end process ;
                 RESET <= INHIBIT ;
            }
 end
```

Figure 6-11 comportement Vhdl remonté du générateur d'impulsion à chaîne de retard

```
........
process(nnum165)
begin
if (nnum165 = '1') then
nnum104 <= '1' after 0ns , '0' after 2ns ;
end if ;
end process ;
nnum143 <= nnum104 ;
...
```

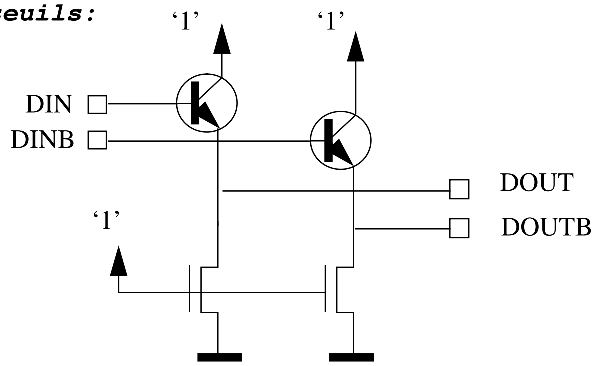
6.5 Tests d’abstraction de technologies BiCMos

Nous donnons ici à titre d’exemple les règles qui ont permis de réaliser l’abstraction d’un bloc de 91 primitives (transistors et résistances) qui réalise un circuit d’interface en technologie BiCMos. La taille relativement faible d’un tel bloc s’explique par le fait que la technologie BiCMos est surtout utilisée dans les composants VLSI pour réaliser des interfaces.

Nous avons détaillé en Figure 6-13, Figure 6-14 et Figure 6-12 des schémas de quelques une des portes BiCMos de ce circuit avec leurs règles d’abstraction, et on trouvera en ANNEXE E l’ensemble complet des règles BiCMos qui ont permis de remonter la description de ce bloc. Nous voyons d’après ces schémas et leurs règles associées comment grâce à notre méthode il est possible de décrire aisément n’importe quel type de porte dans une technologie autre que le CMos et d’exprimer les comportements des circuits correspondants.

Figure 6-12 Schéma et règle d’abstraction d’un circuit adaptateur de seuil

Circuit adaptateur de seuils:

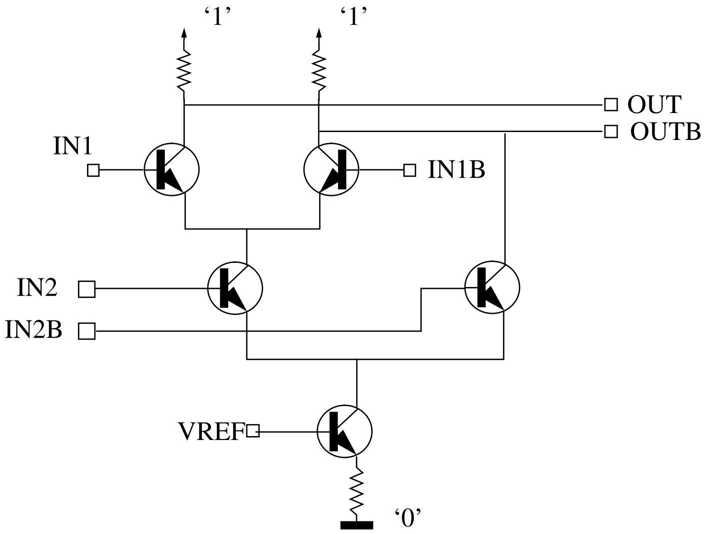


Règle d’abstraction:

```
-> "LOW LEVEL"
if
        npn with b=DIN ; e=DOUT ; c='1'
   and
        mos_n with g='1' ; d='0' ; s=DOUT
   and
        mos_n with g='1' ; d='0' ; s=DOUTB
   and
        npn with b=DINB ; e=DOUTB ; c='1'
then
        build lowlev with
             input := DIN, DINB ;
             output := DOUT, DOUTB ;
             vhdl := { DOUT <= DIN ;
                  DOUTB <= DINB ;
                  } ;
             verilog := { assign DOUT = DIN ;
                  assign DOUTB = DINB ;
                  }
end
```

Figure 6-13 Schéma et règle d'abstraction d'une porte BiCMos

Porte nand:

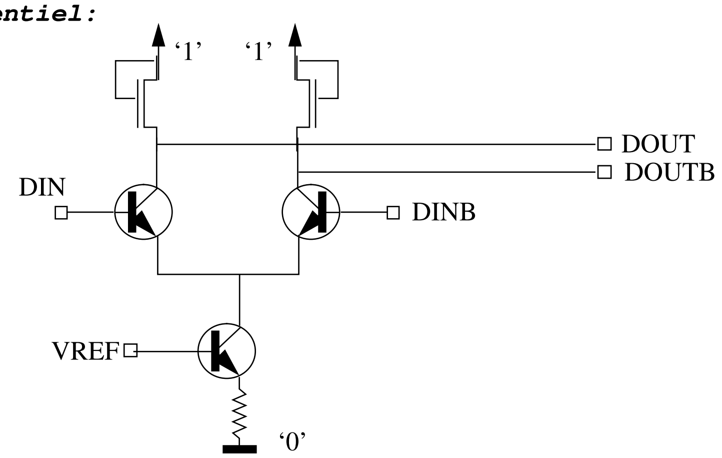


Règle d'abstraction:

```
-> "NAND BiCmos 2"
 if
       resistor with e1=OB ; e2='1'
 and
       npn with b=I1B ; e=ND1 ; c=OB
 and
       npn with b=I1 ; e=ND1 ; c=O
 and
       resistor with e1=O ; e2='1'
 and
       npn with c=ND1; b=I2; e=ND2
 and
       reduced ND1
 and
       npn with c=OB; b=I2B; e=ND2
 and
       npn with b=VREF ; e=M2 ; c=ND2
 and
       reduced ND2
 and
       resistor with e1=M2 ; e2='0'
 and
       reduced M2
 then
       build nandbc with
            input := I1,I1B,I2,I2B;
            output := O,OB;
            command := VREF;
            internal := ND1, ND2, M2;
            vhdl := { O <= not (I1 and I2) ;
                 OB <= not O ;
                 } ;
            verilog := { assign O = ~(I1 & I2), OB = ~O;
                 }
 end
```

Figure 6-14 Schéma d'un amplificateur différentiel d'adaptation

Adaptateur différentiel:



Règle d'abstraction:

```
-> "BOOST2"
 if
          mos_p with g='1' ; d=DOUTB ; s='1'
   and
          npn with b=DIN ; e=ND1 ; c=DOUTB
   and
          npn with b=DINB ; e=ND1 ; c=DOUT
   and
          mos_p with g='1' ; d=DOUT ; s='1'
   and
          npn with b=VREF ; e=M2 ; c=ND1
   and
          reduced ND1
   and
          mos_p with g='1' ; d=M2 ; s='0'
   and
          reduced M2
 then
          build boost2 with
               input := DIN,DINB;
               output := DOUT,DOUTB;
               internal := ND1, M2;
               vhdl := { DOUT <= not DINB ;
                    DOUTB <= not DIN ;
                    } ;
               verilog := { assign DOUT = ~DINB ;
                    assign DOUTB = ~DIN ;
               }
end
```

6.6 Tests d'externalisation du procédé

Les utilisations industrielles externes que nous allons maintenant décrire prouvent qu'il est possible d'exporter notre méthode.

La première utilisation externe a été effectuée par des partenaires de Bull dans le programme Européen JESSI *[Jessi-AC3],[Jessi-EDAC94],[JessiSp3Tcs96], [Jessi-BBA96]* qui sont pour *Grog Thomson TSF-TCS* et *SGS-Thomson*. Suite à cela, la commercialisation du logiciel par Bull a permis de le faire adopter par des clients en Europe et aux USA.

Nous décrivons donc à titre d'exemple les projets des deux premières sociétés qui ont utilisé Grog:

- Thomson CSF-TCS

  Grog a été utilisé ici dans le cadre de projet Européen *JESSI-AC3* pour faire de la vérification en étant associé à de la simulation et à de la preuve formelle, et pour faire de la migration technologique afin d'assurer la pérennité des composants, et enfin pour produire une documentation actualisée en Vhdl à partir d'anciens composants.

  Grog a permis en particulier de faire migrer un composant nommé *ARC* en technologie CMos de 1.4 µm qui était alors composé de 38 types différents de circuits et totalisait 54000 transistors vers une technologie CMos de 0.8 µm. Il n'existait pas de spécification Vhdl ou Verilog du composant, et une description *GDS2* du dessin des masques était seule disponible. Une *net-list* au format *SPICE/CDL* a pu être extraite du GDS2 par l'outil CADENCE/DRACULA pour être ensuite entrée dans *Grog*. L'abstraction dans un tel contexte s'est faite selon un mode itératif avec un enrichissement progressif d'un catalogue de règles déjà existant à partir des transistors non reconnus jusqu'à obtenir la reconnaissance complète du composant. Cette approche a ainsi permis de détecter et d'identifier précisement tous les circuits particuliers, et de les analyser pour en entrer ensuite un modèle précis au moyen des règles.

  Après abstraction, le circuit a pu ensuite être synthétisé par l'outil de synthèse *Synopsys* avec le catalogue TSBC4_hd en 0.8 µm pour produire un fichier en format *EDIF* qui a pu être à son tour entré dans l'outil de placement-tracé de Cadence.

  La taille du catalogue de règles développée par Thomson pour couvrir l'ensemble de ses projets est évaluée à 120 règles.

- SGS-Thomson

  Grog a été utilisé à *Crolles* en France et à *Agrate* en Italie, et ceci toujours dans le cadre du projet Européen *JESSI-AC3.*

  Grog a permis ainsi d'effectuer la migration de composants d'une vieille technologie CMos vers une technologie *standard cell* plus récente de type HCMOS5. Le Vhdl décrit dans les règles *Grog* était prévu pour pouvoir être entré dans l'outil de synthèse *Synopsys*. 45 règles ont été écrites pour reconnaître plus de 450 circuits combinatoires, et 75 règles pour plus de 130 circuits de type registre.

  Grog a permis également d'effectuer de la Vérification et de la documentation de composants, et il est à noter qu'à cette occasion, un bloc de 241975 transistors a pu être abstrait en moins de 10 minutes cpu sur une machine *Sun* de type *Ultra Sparc*.

6.7 Conclusion

Nous avons tout d'abord dans ce chapitre examiné la linéarité du procédé qui est convenable au regard des tailles de données habituellement traitées.

Nous avons vu ensuite que pour être performants, certains composants VLSI utilisés dans les projets d'ordinateurs chez *Bull* utilisent des schémas de circuits complexes et spécialement optimisés. Les résultats qui ont été obtenus sur les 2 projets *KLOE* et *ISL* montrent que *Grog* a permis d'abstraire des circuits complexes, ce qui prouve la puissance du langage de règles.

Nous avons vu que certains circuits asynchrones réalisés pour des clients extérieurs échappaient totalement à la méthodologie de conception qui chez *Bull* est traditionnellement en logique synchrone. Comme tous les schémas spécifiques à chacun des projets cités ont pu être pris en compte et décrits dans les règles d'abstraction, *Grog* a permis de gérer efficacement la diversité des circuits.

Grog permet d'abstraire une grande variété de circuits, en logique synchrone ou asynchrone, et de donner exactement la forme souhaitée au comportement HDL remonté. La technologie BiCmos a pu elle aussi être traitée aisément par notre procédé.

Notre méthode est donc souple puisqu'elle modéliser facilement une grande variété de circuits, quelle que soit leur complexité et la technologie employée.

Nous avons vu que le principe de constitution des catalogues de règles a permis de faire partager par l'ensemble des projets un noyau commun de règles et de lui adjoindre ensuite pour chaque projet les règles spécifiques qui le concernent. Notre méthode permet donc de gérer efficacement les modèles de circuits.

Nous avons vu quelles étaient les utilisations externes de Grog chez *Thomson TCS* et *SGS Thomson*, ce qui montre que la méthode est exportable.

Par les résultats obtenus nous avons vu que la méthode pouvait être utilisée à la fois dans des contextes de vérification et de migration technologique.

7. Conclusions

Notre méthode de reconnaissance de circuits à partir de modèles permet de construire rapidement plusieurs catalogues de modèles, chacun de ces catalogues pouvant être individualisé et dédié à un projet et à une technologie. Toutes les astuces de conception des schémas peuvent être prises en compte par *Grog*, ce qui lui permet de reconnaître comme nous l'avons vu les circuits d'entrée-sortie, les amplificateurs différentiels, et la grande variété des styles de registres.

Pour obtenir cela, nous avons défini un langage de règles dont les principaux atouts sont à la fois sa souplesse et sa puissance de description des modèles de circuits électroniques.

Au niveau de la mise en oeuvre, des non informaticiens ont pu facilement constituer eux mêmes les catalogues décrivant les modèles de circuits. En effet, à la différence de *Desa et de Dialog* pour lesquels la description des modèles exigeait une connaissance approfondie respectivement des langages *Prolog* et *Lisp*, la description des modèles au moyen de règles dans Grog ne nécessite pas de connaissances préalables en programmation, et elle fait abstraction de toute notion relative au fonctionnement interne du programme. De plus, le langage de règles se suffit à lui même, pour décrire les circuits, et il n'est pas nécessaire d'écrire des fonctions annexes. C'est ce qui permet de transférer facilement vers des non informaticiens la gestion des applications.

Le principe utilisé permet aussi d'adapter aisément l'abstraction aux méthodes de conception de n'importe quelle entreprise. Il est possible de traiter avec une telle méthode non seulement la technologie *CMos*, mais également les technologies *NMos*, *Bipolaire* et *Bicmos*.

Le fait de savoir traiter des circuits au fonctionnement électrique complexe montre que l'approche se prête de par son principe au traitement de circuits mixtes analogiques et digitaux. Il sera donc tout à fait possible d'établir une spécification de type *Vhdl-A* ou *Verilog-A* de ces circuits lorsque les standards correspondants seront définitivement adoptés.

Notre procédé d'abstraction a pu être utilisé chez Bull et il a pu aussi être aussi exporté. A partir de là, il a été utilisé aussi bien pour effectuer de la vérification de composants que pour effectuer de la migration technologique. Une conséquence immédiate de notre méthode de reconnaissance de formes est que celle ci permet de reconnaître uniquement les circuits qui ont été spécifiés au préalable dans le catalogue de règles. C'est pourquoi, dans une chaîne de vérification, l'abstraction pilotée par des règles permet de garantir la conformité des circuits par rapport à une méthodologie de conception donnée. En effet, les circuits hors normes se trouvent non reconnus, c'est à dire automatiquement rejetés. *Grog* peut donc être utilisé non seulement pour abstraire un comportement mais aussi pour vérifier la conformité des circuits vis à vis des spécifications en cours. Dans un tel contexte, l'expérience nous montre que l'abstraction constitue à elle seule une part importante dans le processus de vérification global des VLSI dès lors qu'un grand nombre d'erreurs de conception peut être détecté immédiatement lors de l'abstraction, c'est à dire au plus près de la phase de conception.

Tandis que la taille des composants VLSI et la densité d'intégration augmentent avec le progrès de la technologie, notre approche hiérarchique de l'abstraction qui permet de traiter de grandes quantités de données en des temps acceptables est tout à fait adaptée à de tels changements d'échelle.

Un avantage important de notre méthode est aussi la concision des descriptions abstraites. En effet, comme cette méthode d'abstraction n'est pas limitée à la seule reconnaissance de portes de base, des sous ensembles de circuits complexes tels que par exemple des registres, et même des bancs de registres, peuvent être reconnus, et des fonctions logiques de plus haut niveau leur être alors directement associées. Une description du style comportement de registre est alors plus explicite que sa décomposition en latches, ce qui la rend préférable à cette dernière.

La possibilité de vectorisation contribue à rendre le comportement abstrait plus facilement lisible parce que plus concis, et celui ci devient plus rapidement simulable et plus facile à traiter par les outils de synthèse.

Enfin, la description fonctionnelle qui est remontée est conforme aux standards industriels Vhdl et Verilog. Le langage de règles permet d'associer simultanément aux descriptions des structures des circuits des spécifications de comportements aussi bien de bas niveau dit *data-flow* que de plus haut niveau dit *comportemental*, en prenant en compte simultanément les 2 standards Vhdl et Verilog. Les comportements produits sont immédiatement assimilables à la fois par les principaux simulateurs du commerce (*Synopsys*, *Cadence*,...), par des logiciels de preuve formelle tels que *VFormal* de *Bull* et aussi par des outils de synthèse automatique (*Compass*, *Synopsys*). Le principe utilisé pour la mise en place des comportements dans les règles d'abstraction permet au besoin d'étendre l'abstraction à d'autres langages de description comportementaux.

Comparée aux techniques dites 'algorithmiques', notre application est générale puisqu'elle permet de prendre en compte tous les types de portes logiques simples ou complexes ainsi que les circuits analogiques, et elle est aussi multi-technologies. Comme elle est configurable par ses utilisateurs, cela lui permet d'être exportée dans d'autres environnements industriels.

Comparé aux autres logiciels d'abstraction basés eux aussi sur des techniques de reconnaissance de forme tels que *Desa*, *Vera* et *Dialog*, l'avantage de Grog est d'être simple à configurer grâce au format du langage de règles GRL que nous avons élaboré. Ce langage de règles permet de décrire aisément dans un formalisme unique toutes les configurations de circuits (*portes*, *registres*, *multiplexeurs*, *précharges*, *mémoires*, *amplificateurs*). Son formalisme est suffisant et homogène, car il ne nécessite pas de faire appel à d'autres mode de fonctionnement, et il ne nécessite pas non plus de passer par des procédures programmées.

Nous avons choisi des solutions qui rendent le procédé d'abstraction à la fois simple à appréhender et facilement contrôlable. C'est ainsi que le concept de retour arrière que nous avons introduit dans les règles, permet, en l'associant à une structure prédicative, non seulement de décrire de façon concise tous les circuits, mais de savoir aussi prendre en compte leur variantes. La solution que nous avons choisie permet en effet de spécifier des alternatives de construction et d'introduire facilement plusieurs opérateurs additionnels tels que en particulier les opérateurs d'équivalence; elle permet aussi de prendre en compte le contexte d'utilisation des circuits à reconnaître. De plus c'est grâce au mécanisme particulier de déclenchement des règles de Grog que les règles d'abstraction peuvent décrire de façon générique des structures répétitives sans qu'il soit nécessaire de spécifier au préalable la taille exacte de ces structures.

Enfin, toujours en termes comparatifs, un atout supplémentaire de Grog est de réaliser directement une abstraction fonctionnelle vectorisable et donc compacte, et non pas seulement structurelle. La méthode employée permet aux utilisateurs de spécifier exactement dans les règles d'abstraction les comportements qui doivent être associés aux modèles de circuits. Des comportements HDL tels que Vhdl et Verilog peuvent être associés facilement aux modèles de circuits, et ceci sans limitation de complexité vis a vis des niveaux de spécification Vhdl et Verilog existants. Il est ainsi possible de cibler précisement n'importe quel sous ensemble de Vhdl ou Verilog de n'importe quel outil de vérification ou de synthèse, ce qui contribue à rendre l'application totalement exportable. De plus, dans l'approche hiérarchique, les comportements des sous ensembles de circuits peuvent être facilement assemblés pour construire les comportements de niveau supérieur.

Nous pouvons maintenant à partir des résultats obtenus chez *Bull* et chez des utilisateurs externes envisager certaines évolutions.

Nous avons vu que la vectorisation était une fonction importante qui permettait d'obtenir une description résultante beaucoup plus compacte qui passe mieux en simulation, preuve formelle et re-synthèse, et qui soit aussi plus facilement lisible textuellement. Il est encore possible d'améliorer la méthode pour arriver à produire des descriptions génériques telles que la directive *for... generate...* en Vhdl. Ceci pourra alors se faire en considérant les circuits qui partagent un ensemble de signaux de commande d'indices successifs, et en les regroupant sous l'instruction précitée.

Enfin, un grand nombre de règles concernent des circuits simples et communément utilisés (portes *nand, nor,...*), c'est pourquoi nous pensons qu'il serait judicieux par la suite d'associer dans le même outil deux modes d'abstraction tels que l'on dispose à la fois d'un mode de reconnaissance piloté par des règles et d'un mode de reconnaissance qui mette en oeuvre des procédures algorithmiques de reconnaissance. En effet, ceci permettrait d'effectuer d'abord la reconnaissance pilotée par les règles pour reconnaître tous les circuits complexes ou réalisés en technologie non CMos, puis d'enclencher la procédure automatique pour traiter les configurations usuelles qui constituent le reste du circuit. Une telle approche qui combinerait les 2 méthodes d'abstraction existantes à ce jour permettrait d'alléger d'autant le contenu des catalogues de règles et la charge de travail nécessaire pour les constituer, tout en disposant de la souplesse de notre méthode pour les circuits complexes ou de technologie non CMos.

BIBLIOGRAPHIE

[AlgoSwi87]	R.E.Bryant	Algorithmic Aspects of Symbolic Switch Network Analysis IEEE transactions on CAD, Vol CAD-6, no4, July 1987
[AiPrincip82]	N. J.Nilson	Principles of Artificial Intelligence, Springer-Verlag Berlin Heidelberg New York 1982 ISBN 3-540-11340-1
[CadRm91]		Cadence Design Systems, Inc, 555 River Oaks, San Jose, California 95134. Reference Manual, October 1991.
[CompNetl92]		Compass Design Automation, 505, route des Lucioles 06560 Sophia Antipolis, France - Netlist Utility 1992
[CompSynth]		ASIC Synthesizer for Vhdl Design V8R4.0
[Cosmos87]	R.E.Bryant	Boolean Analysis of MOS Circuits IEEE transactions on CAD, July 1987
[Dax89]	Ph.Deverchère,	The Layout Extractor DAX, Bull internal report, February 1989
[Desa90]	J.B.Guignet	Désassemblage de circuits électroniques, Journées Prolog de l' AFCET, Jeudi 5 Avril 1990, PARIS
[Desb92]	M.Laurentin, A.Greiner, R.Marbot	DESB: a Functional Abstractor for CMos VLSI Circuits", Eurodac 92.
[Desb95]	M.Laurentin	Désassemblage de circuits électroniques, Thèse de Doctorat - Université de PARIS6
[Dialog89]	Ivo Bolsens	Electrical Correctness Verification Of Mos Digital Circuits Using Expert System And Symbolic Analysis Techniques Thèse du Centre Interuniversitaire de Micro-Electronique (IMEC) Mai 1989 - Universite Catholique de Louvain
[Edif87]		EDIF Steering Comittee Electronic Industries Association Electronic Design Interchange Format, version 2.0, 1987
[Europro]		Projet du programme ESPRIT - Projet ESPRIT/HPCN Numero 21040
[Faon86]	J.Y.Murzin,	A Functional Abstractor of Netlist, Séminaire de Programmation en Logique, Mai 1986 Lannion.
[FMetho87]	D.S.Reeves, M.J.Irwin	Fast Methods for Switch-Level Verification of MOS Circuits IEEE Transactions on CAD, Vol. CAD-6 , Sept87
[Gnuc++90]	M.D.Tiemann, D.Lea	User's Guide to GNU C++ and its Libraries Free Software Foundation, Inc - 1990
[GrogFsd91]	J.B.Guignet	FSD Functional Specification Document Bull internal specification, 2 fevrier 1991, Les Clayes sous Bois
[GrogHldd91]	J.B.Guignet,	HLDD High Level Design Document Bull internal specification,27 Janvier 1992, Les Clayes sous Bois

[GrogUg97]	J.B.Guignet	GROG User's Guide - Version 07 - Juin 97
[IaRp86]	J.L. Lauriere	Intelligence Artificielle, resolution de problémes par l'Homme et la machine - EYROLLES 1986
[IntroVlsi80]	C.Mead, L.Conway	Introduction to VLSI Systems Addison-Wesly Publishing Company 1980
[Jessi-AC3]	P.Garcin	Eurodac 1994 - Achieved Results - Current activities Subproject 3: Computer subsystem modeling and formal proof.
[Jessi-EDAC94]	J.P.Tual	CAD Developments in JESSI: Overview and user perspective
[JessiSp3Tcs96]	J-F.Agaesse, S.Dalpont, E.Vouriot, E.Zardi	JESSI AC3-SP3 - THOMSON TSF-TCS - Rapport d'activité Report on the application of layout abstraction on the basis of a real asic example. R.TCS-SP3-W2.1-Q8
[Jessi-BBA96]	J.Papanuskas,Ph.Oddo, P.Olive,S.Nicot, JF.Agaesse,D.Borrione, P.Magarshak, R.ZinsZner,H.Durantis, D.Bouteille,	JESSI AC3 - HDL, Component Modeling & Libraries JESSI APPLICATION - Blue Book Abstract SGS-Thomson (Project Leader) - 10 Mai 1995
[Logex88]	M. Boehner	LOGEX An Automatic Extractor from Transistor to Gate Level for CMOS Technology Siemens AG, Research Laboratories 25th ACM/IEEE Design Automation Conference
[Mosi80]	R.E. Bryand	An Algorithm for MOS Logic Simulation, Lambda, 4th qtr. 1980, pp.46-53
[NetwPart87]	V.B.Rao, T.N.Trick	Network Partitioning and Ordering for MOS VLSI Circuits IEEE Transactions on computer-aided design, vol. CAD-6, JAN87
[Paris91]	A.Debreil	Vhdl Formal Proof, Bull New CAD Project 1991
[Priam88]	J.C.Madre, J.P.Billon,	Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behavior, Proc. of the 25th Design Automation Conference, June 1988
[Pcvd85]	N.Weste, K.Eshraghian	Principles of CMOS VLSI Design, a Systems Perspective Addison-Wesly Publishing Company - ISBN 0-201-08222-5
[SynchroD92]	A.Debreil, D.Jaillet	Synchronous Description in Vhdl and Vhdl Subset for Formal Proof - Ecip2 deliverable R_B_2.5_43_a-Row41 - Juillet 1992
[RFormes85]	L.Miclet , C.Faure	Tends in structural pattern recognition, 5éme Congrés De Reconnaissance De Formes Et Intelligence Artificielle 27-29 Novembre 1985, Grenoble
[SigFlow87]	N.P.Jouppi	Derivation of Signal Flow Direction in Mos VLSI IEEE transactions on CAD, Vol. CAD-6 no3, May87
[SynopComp]		Synopsys700 East Middlefield Road, Mountain View, CA 94043 HDL Compiler Family Vhdl Compiler , HDL Compiler for Verilog
[Sp3St96]	U.Rossi, A.Fedeli, V.Anjubault	JESSI AC3-SP3 - THOMSON ST - Central R&D Crolles, Agrate Technical meeting - Genève 1996
[Survey87]	R.E.Bryant	A survey of Switch Level Algorithms IEEE Design & Test - 1987
[TWIN]		Sous-projet C du projet européen JESSI-AE96

[Vera91]	T.Kostelijk, B. De Loore	Automatic Verification of Library-Based IC Designs, IEEE Journal of Solid-State Circuits, Vol 26, No.3,March 1991
[VerilogIeee95]		IEEE Standard Verilog Reference Manual, IEEE P1364-1995/A
[Vformal]	A.Debreil	User's Manual - Formal Verification for VLSI Designs Bull internal specification
[VhdlMod90]	R.Airiau, J.M.Bergé, V.Olive, J.Rouillard	Vhdl du Langage à la modélisation, presses polytechniques et universitaires romandes 1990.
[VhdlIeee87]		IEEE Standard Vhdl Reference Manual IEEE Std 1076-1987
[VhdlCook90]	P.J.Ashenden	The Vhdl Cookbook - Dept. Computer Science, University of Adelaide, South Australia - July 1990
[Vti]	B.Shur	User Manual, HNL Format and Programming Tools VTI Design Technology Group.

ANNEXES

ANNEXE A : Le format GND

A.1 Syntaxe du format GND

- **Déclarations**
  - Commentaires:
    - # *une chaîne de caractères alphanumériques jusqu'à la fin de la ligne*
  - Connecteur du bloc:
    - NC *numéro_de_signal nom_de_signal*;
  - Signal interne:
    - NI *numéro_de_signal nom_de_signal marque*;
  - Transistor Mos N à enrichissement:
    - MN *g d s x y w l cg;*
  - Transistor MosP à enrichissement:
    - MP *g d s x y w l cg;*
  - Transistor Mos N à déplétion:
    - MD *g d s x y w l cg;*
  - Transistor Bipolaire NPN:
    - BN *b e c x y es;*
  - Transistor Bipolaire PNP:
    - BP *b e c x y es;*
  - Diode:
    - DD *anode cathode x y;*
  - Résistance:
    - DR *e1 e2 x y r;*
  - Fin de description:
    - EE;

- **Signification des éléments terminaux**
  - ***numéro_de_signal*** (entier) est un nombre courant qui identifie le signal.
  - ***nom_de_signal*** (alphanumérique) est le nom du signal.
  - ***marque*** est un simple caractère optionnel (pris dans [a-z]) qui sert à repérer si besoin des signaux remarquables (horloges, précharges réparties...).
  - ***g, d*** et ***s*** (entiers) sont les numéros des signaux attachés aux connecteurs de grille, drain, source d'un transistor Mos.
  - ***e, b*** et ***c*** (entiers) sont les numéros des signaux attachés aux connecteurs d'émetteur, base et collecteur d'un transistor Bipolaire.
  - **anode** et **cathode** (entiers) sont les numéros des signaux attachés à l'anode et à la cathode d'une diode.
  - **e1** et **e2** (entiers) sont les numéros des signaux attachés aux 2 connecteurs d'une résistance.

- **r** (flottant) est une valeur de résistance
- **cc** (flottant) est une valeur de capacité
- ***x*** et ***y*** (flottant) sont des coordonnées.
- ***w*** et ***l*** (flottant) sont les largeur et longueur d'une grille de transistor mos.
- ***cg*** (flottant) est la valeur capacitive d'une grille de transistor Mos.
- ***es*** (flottant) est la surface émetteur d'un transistor Bipolaire.

- **Remarques**
  - L'ordre suivant des déclarations doit être respecté: *NC,NI* en tête de fichier suivies des autres déclarations, le tout terminé par *EE*.
  - Les noms de signaux '0' et '1' servent à identifier les signaux d'alimentation.
  - Les lignes de commentaire peuvent être placées à n'importe quel endroit.
  - Quand elles sont indéfinies, les valeurs flottantes doivent être égale à 0.0.

A.2 Un exemple de fichier au format GND

- Fichier décrivant une *net-list* BiCmos au format GND:

`# GROG Netlist Description` `#`	Commentaires

`NC 1 d1(0);`
`NC 2 d1(1);`
`NC 9 '0';`
`NC 10 q(0);`
`NC 12 q(1);`
`NC 13 q(2);`
`NC 14 q(3);`
`NC 15 '1';`
	Connecteurs du bloc

`NI 16 sig1 ;` `NI 17 sig2 ;`	Signaux internes
`...` `...`	
`MN 5 7 41 0.0 0.0 8.80 0.80 0.000 ;` `MN 9 67 41 0.0 0.0 8.80 0.80 0.000 ;`	Transistors Mos N
`...`	
`MP 12 13 58 0.0 0.0 13.20 0.80 0.000 ;` `MP 68 72 58 0.0 0.0 13.20 0.80 0.000 ;`	Transistors Mos P
`...`	
`BN 1044 1004 13 11.5 20.5 22.0 ;` `BP 1044 1004 12 11.5 20.5 22.0 ;`	Transistors bipolaires
`...`	
`DD 2010 2011 222 11.5 20.5 ;` `DD 2015 2014 222 11.5 20.5 ;`	Diodes
`...`	
`DR 2020 2021 21.5 22.5 333.0 ;` `DR 2022 2022 21.5 22.5 333.0 ;`	Résistances
`EE ;`	Fin de description

ANNEXE B :Spécification du langage de règles GRL

B.1 Structure d'une règle

Une règle d'abstraction consiste en une déclaration du type

-> NOM **if** MODELE **then** ACTION **end**

dans laquelle:

- ❐ NOM est une chaîne de caractères qui sert à **identifier** la règle.
- ❐ MODELE est une description du **modèle** à rechercher.
- ❐ ACTION est une description des **actions** à exécuter lorsque la partie modèle est satisfaite.

- Structure du modèle

Un modèle consiste en une suite d'**expressions logiques** s'évaluant à Vrai ou à Faux et séparées par un des opérateurs de conjonction ou de disjonction qui sont ***and*** et ***or***.

Une expression logique peut consister en:

- un prédicat, ou bien un prédicat précédé par l'opérateur d'observation ***observed***.
- un opérateur ***reduced*** suivi d'une variable ou d'une liste de variables.
- deux listes de variables séparées par l'un des opérateurs d'équivalence entre deux listes noté ***?=*** ou d'appartenance d'un élément à une liste noté **@=**.
- des variables ou des constantes ou des listes de variables et constantes séparées par l'opérateur d'unification noté **=**.
- un opérateur ***not*** suivi d'une expression.

- Structure de la partie action

La partie action permet de réaliser une ou plusieurs actions élémentaires séparées obligatoirement par l'opérateur ***and***. La partie action d'une règle ne peut pas contenir plus d'un ordre *build*. Une action élémentaire qui s'évalue toujours à Vrai peut consister en:

- un opérateur de construction ***build*** suivi d'un prédicat décrivant l'Unité Abstraite à construire.
- un opérateur de mise à jour ***update*** suivi par un prédicat décrivant le signal à mettre à jour.
- un opérateur ***echo*** suivi d'un texte pouvant contenir des variables et placé entre les caractères *{* et *}* .

B.2 Variables et constantes

Une **variable** commence par un caractère majuscule.

Une **constante** est un entier, une valeur décimale ou bien un nom qui commence par un caractère minuscule ou qui est placée entre simples quotes ‘’'.

Les signaux zero et un logique sont représentés par des constantes particulières et ils seront ainsi respectivement désignés par ***‘0’*** et ***‘1’***.

On peut constituer des listes de variables et constantes en utilisant le caractère séparateur ‘,’.

B.3 Syntaxe des prédicats

Un prédicat est décrit sous la forme d'un nom d'identification suivi du mot clé ***with*** et d'une liste d'opérations sur les attributs du prédicat.

On distingue 2 types d'opérations sur les attributs d'un prédicat, selon que ce prédicat décrit un accès à un composant ou à un signal ou bien que ce prédicat décrit la création d'une Unité Abstraite ou la mise à jour d'un signal:

- lors d'une opération d'accès, la lecture de la valeur associée à l'attribut se fait grâce à l'opérateur '='.
- lors d'une opération de création ou pour la mise à jour d'un attribut, on utilise l'opérateur ':='.

Les opérations sur les attributs doivent être réalisées en accord avec les seules opérations légales suivantes:

- accès en lecture aux signaux, primitives ou Unités Abstraites.
- accès en mode création pour les Unités Abstraites
- accès en mode mise à jour pour les signaux.

Les opérations sur les attributs sont séparées par le caractère ';'.

Les attributs de chaque prédicat font référence aux attributs de l'objet correspondant (par exemple l'attribut ***g*** de l'Unité Abstraite Mos correspondra à l'attribut ***gate*** de l'objet Mos).

A droite du nom d'un attribut, on trouve soit le symbole '**=**' ou le symbole '**:=**' qui doivent être suivis par une constante ou par une variable et qui sont spécifiés ainsi:

❐ le symbole =

Ce symbole est toujours présent dans un prédicat de la partie modèle d'une règle, et il permet:

- d'imposer une valeur de filtrage si l'élément situé à sa droite est une constante ou une variable instanciée
- de recueillir la valeur de l'attribut si l'élément de droite est une variable.

❐ le symbole ':='

Ce symbole est toujours présent dans la partie conclusion d'une règle, il permet d'affecter une valeur à un attribut de l'objet concerné.

Il est utilisé lors d'une construction par ***build*** ou lors d'une mise à jour par ***update***.

B.4 Les différents prédicats

• Les prédicats d'accès aux transistors Mos

Ils se composent d'un identificateur ***mos_n***, ***mos_p***, ou ***mos_d*** suivi du mot clé *with* et d'une suite d'accès par '=' aux attributs.

Les attributs sont:

- la connection de grille désigné par ***g***
- la connection de drain désigné par **d**
- la connection de source désigné par ***s***
- l'abscisse désigné par ***x***
- l'ordonnée désigné par ***y***
- la largeur de grille désigné par ***w***
- la longueur de grille désigné par ***l***

• Les prédicats d'accès aux transistors bipolaires

Ils se composent d'un identificateur ***npn***, ou ***pnp*** suivi du mot clé *with* et d'une suite d'accès par '=' aux attributs. Les attributs sont:

- la connection de base désigné par ***b***
- la connection de collecteur désigné par ***c***
- la connection d'émetteur désigné par ***e***
- la surface de collecteur désigné par e***s***
- l'abscisse désigné par ***x***
- l'ordonnée désigné par ***y***

- **Les prédicats d'accès aux diodes**

Ils se composent d'un identificateur ***diode*** suivi du mot clé *with* et d'une suite d'accès par '=' aux attributs.

Les attributs sont:

- l'anode désigné par ***anode***
- la cathode désigné par ***cathode***
- l'abscisse désigné par ***x***
- l'ordonnée désigné par ***y***

- **Les prédicats d'accès aux résistances**

Ils se composent d'un identificateur ***resistance*** suivi du mot clé *with* et d'une suite d'accès par '=' aux attributs.

Les attributs sont:

- les attributs ***e1*** et ***e2*** qui désignent les signaux d'accès à la résistance
- l'abscisse désigné par ***x***
- l'ordonnée désigné par ***y***

- **Les prédicats d'accès et de création des Unités Abstraites**

- Ils se composent d'un identificateur qui est le nom du circuit, lequel est choisi par la personne qui écrit les règles. Cet identificateur est suivi du mot clé *with* et d'une suite d'opérations sur les attributs réalisées par les opérateurs '=' en accès ou ':=' en création.

Les attributs sont:

- la liste des signaux de commande désigné par ***command***
- la liste des signaux d'entrée désigné par ***input***
- la liste des signaux de sortie désigné par ***output***
- l'équation VHDL désigné par ***vhdl***
- l'équation VERILOG désigné par ***verilog***
- des champs supplémentaires à la disposition de l'utilisateur, et qui permettent de mémoriser des valeurs ou des listes de valeurs quelconques: ***field1***, ***field2, field3, field4, field5.***

- **Les prédicats d'accès et de mise à jour des signaux**

Ils se composent de l'identificateur *signal* suivi d'une variable puis du mot clé *with* et d'une suite d'accès aux attributs réalisés par l'opérateur '=' ou de mise à jour réalisée par l'opérateur ':=', et séparés par le caractère *';'*.

Ces attributs sont:

- le type du signal ***type*** qui caractérise la visibilité du signal dans le bloc (***internal*** ou ***external***).
- le nom de signal ***name***
- le numéro de signal ***signal_number***

- le champ ***tag*** qui sert à qualifier le signal et qui peut recevoir une des valeurs suivantes: ***latch, register, precharge, tristate***

B.5 Les opérateurs du langage de règles

- ❒ **and**

  Cet opérateur permet établir une conjonction entre deux prédicats ou expressions logiques d'un modèle

- ❒ **or**

  Cet opérateur permet d'établir des disjonctions entre deux prédicats ou expressions logiques d'un modèle.

- ❒ **not**

  L'opérateur ***not*** permet d'inverser le résultat de l'évaluation d'une expression logique de comparaison ou d'un prédicat.

- ❒ **build**

  L'opérateur ***build*** permet de construire une Unité Abstraite. Cet opérateur ne peut apparaître que dans la partie conclusion de la règle et une seule fois.

- ❒ **observed**

  L'opérateur ***observed*** est destiné à vérifier la présence d'un composant sans que celui ci ne fasse partie de l'Unité Abstraite qui sera construite.

- ❒ **update**

  L'opérateur ***update*** est destiné à mettre à jour les différents champs d'un objet signal. L'opérateur ***update*** ne peut apparaître que dans la partie conclusion de la règle.

- ❒ **reduced**

  L'opérateur ***reduced*** permet de vérifier qu'un signal N est isolé, c'est à dire qu'il réponde aux contraintes suivantes:

  - n'est pas un signal d'alimentation
  - n'est pas un connecteur du bloc
  - n'est relié à aucun composant autre que ceux qui le précédent dans la description du modèle dont il fait partie.

- ❒ **?=**

  L'opérateur **?=** permet de réaliser toutes les permutations possibles entre deux listes. L'utilisation de cet opérateur permet d'exprimer des équivalences.

  L'instruction A,B ?= X,Y est alors équivalente à l'écriture de

  A,B = X,Y or A,B = Y,X

- ❒ **@=**

  L'opérateur **@=** permet d'unifier successivement l'élément (constante ou variable) situé à gauche de l'opérateur avec chaque élément de la liste de située à droite.

- ❒ **echo**

  Cet opérateur permet d'éditer des messages sur le flot standard d'erreur. Il doit être suivi d'une description composée d'éléments de texte et de Variables situés entre caractères { et } .

B.6 Spécification de la syntaxe

La grammaire simplifiée en notation BNF des règles est la suivante:

❐ description d'une règle

règle	:	-> "nom_de_règle" ***if*** modèle ***then*** action ***end***;

❐ modèle

modèle	:	modèle ***and*** modèle_description
	|	modèle ***or*** modèle_description
	|	modèle_description
	;	
modèle_description	:	prédicat_accés_composant
	|	prédicat_accés_signal
	|	condition_de_test
	|	unification
	;	

❐ action

action	:	action_élémentaire ***and*** action_élémentaire
	|	action_élémentaire
	;	

❐ description des prédicats d'accès aux composants et signaux

prédicat_accés_composant	:	[***not***] [***observed***] type_objet ***with*** accés_attributs ;
type_objet	:	*constante_alphanumérique* ;
prédicat_accés_signal	:	[***not***] ***signal*** *variable* ***with*** accés_attributs ;

❐ affectations

accés_attributs	:	accés_attributs ';' accés_attribut
	|	accés_attribut
	;	
accés_attribut	:	nom_attribut '=' *variable*
		nom_attribut '=' *constante_numérique*
	;	
unification	:	liste = liste
	|	liste **?=** liste
	|	variable @= liste
	|	*constante_alphanumérique* @= liste
	;	
nom_attribut	:	*constante_alphanumérique* ;
	;	
liste	:	liste ',' variable
	|	liste ',' *constante_alphanumérique*
	|	variable
	|	*constante_alphanumérique*
	;	

ANNEXE C L’écriture des règles

Nous donnons ici la méthode d’écriture des règles et les heuristiques qui permettent d’optimiser les performances du processus d’abstraction.

C.1 Méthode d’écriture

Lors de l’écriture d’une règle, il faut s’assurer la continuité dans le flot d’affectation des variables. Ceci parce que l’accès aux objets primitives et Unités Abstraites de la base de données interne est indexé par les références aux objets signaux de leur connectique. Connaissant un signal, l’abstracteur sait adresser directement le sous ensemble de composants qui est connecté à ce signal.

La mise en oeuvre d’une telle heuristique est illustré par les exemples de la Figure 7-1 et de la Figure 7-2 grâce auxquels on voit respectivement une bonne et une mauvaise façon de procéder pour construire une porte *nand2*.

Figure 7-1 : exemple de mauvaise forme d’écriture de règle

```
-> "NAND2 : mauvais exemple"
if
   mos_n with g=A ; d=N ; s='0'
and
   mos_p with g=B ; d='1' ; s=OUT
and
   mos_p with g=A ; d='1' ; s=OUT
and
   mos_n with g=B ; d=N ; s=OUT
and
   reduced N
then
   ...
end
```

Figure 7-2 : exemple de bonne forme d’écriture de règle.

```
-> "NAND2 : bon exemple"
if
   mos_p with g=B ; d='1' ; s=OUT
and
   mos_p with g=A ; d='1' ; s=OUT
and
   mos_n with g=B ; d=N ; s=OUT
and
   mos_n with g=A ; d=N ; s='0'
and
   reduced N
then
   ...
end
```

On peut voir dans l’exemple de la Figure 7-1 que les variables A et N seront instanciées lors du filtrage du composant *mos_n*, mais que les valeurs filtrées par ces 2 variables ne seront en aucun cas utilisées pour réaliser l’indexation lors du filtrage du composant *mos_p* suivant.

L’exemple de la Figure 7-2 au contraire montre une bonne disposition des prédicats d’accès aux composants puisque la valeur filtrée par la variable *OUT* lors de l’accès au premier composant *mos_p* est utilisée pour accéder aux deux composants suivants. La continuité de l'indexation est ensuite préservée grâce à la variable *N*.

C.2 Heuristiques d'écriture des règles

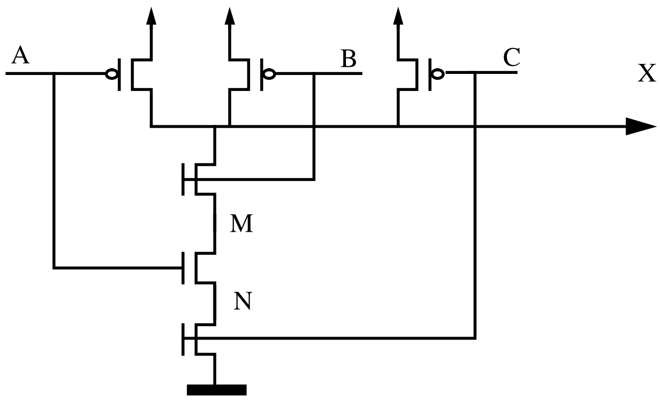
1. Choix du point d'ancrage

   Il est préférable de choisir judicieusement un point d'ancrage qui serve à initialiser la recherche des circuits. Lors de l'écriture des règles, il est bon d'avoir une idée des ordres de grandeur d'occurrence des différents types de composants, afin de choisir comme points d'ancrage ceux dont on estime l'occurrence la plus faible possible. C'est ainsi par exemple, le nombre de Mos N est souvent supérieur à celui des Mos P, le nombre d'inverseurs est souvent supérieur à celui des autres types de portes, le nombre de transistors connectés à une alimentation représente un faible pourcentage de l'ensemble des transistors.

2. Ordre des prédicats dans une règle

   On peut chercher à minimiser les opérations de backtrack dans une règle en ordonnançant judicieusement les prédicats. C'est ainsi que dans l'exemple de la Figure 7-3, on décrira d'abord les prédicats des 3 transistors *mos_n* série avant ceux des transistors *mos_p*.

Figure 7-3 description optimisée d'une porte *nand3*




```
-> "NAND_3.1"
   if
       mos_n with g=A ; d='0' ; s=N
   and
       mos_n with g=B ; d=N ; s=M
   and
       reduced N
   and
       mos_n with g=C ; d=M ; s=OUT
   and
       reduced M
   and
       mos_p with g=A ; d='1' ; s=OUT
   and
       mos_p with g=B ; d='1' ; s=OUT
   and
       mos_p with g=C ; d='1' ; s=OUT
   then
       build   nand3 with
            input     := A, B, C ;
            internal  := N,M ;
            output    := OUT ;
            vhdl      := { OUT <= not (A and B and C);
                                 } ;
            verilog   := { assign OUT = ~(A & B & C);
                                 } ;
end
```

3. Ordre d’écriture des règles

Nous savons que si lors de l’ordonnancement des règles, celles ci ne sont pas soumises à des relations de dépendance, c’est alors l’ordre d’écriture qui est pris en compte. Dans ce cas, il est judicieux de classer explicitement les règles dans un ordre de complexité croissante.

Cette organisation permet en effet de dégrossir le travail de l’abstracteur tout en allégeant le contenu de la base des objets, parce que les éléments peu complexes sont souvent ceux qui ont la plus forte occurrence (par exemple les inverseurs et les portes *nand* et *nor*) et qu’ils prennent relativement peu de temps pour être reconnus puisqu’ils sont moins complexes.

ANNEXE D : Exemples de règles CMos

Pour pouvoir traiter les composants VLSI de Bull, nous avons écrit un ensemble de catalogues totalisant 329 règles, ce qui représente 7800 lignes de texte. Nous présentons ici un extrait du contenu de ces catalogues de règles en technologie CMos.

D.1 Règle de reconnaissance des inverseurs

```
-> "INVERTER"
    if
            mos_p with g=G ; d='1' ; s=S
        and
            mos_n with g=G ; d='0' ; s=S ; l=LN
    then
        build   inverter with
                 input     := G ;
                 output    := S ;
                 field1    := LN ; % on conserve la longueur de grille du mos_n
                 vhdl      := { S <= not G;
                                  } ;
                 verilog   := { assign S = ~G;
                                  }
    end
```

D.2 Règles de reconnaissance de quelques portes nand et nor

```
-> "NAND_2"
    if
        mos_n with g=A ; d='0' ; s=N
    and
        mos_n with g=B ; d=N ; s=OUT
    and
        reduced N
    and
        mos_p with g=A ; d='1' ; s=OUT
    and
        mos_p with g=B ; d='1' ; s=OUT
    then
        build   nand2 with
                    input    := A, B ;
                    internal:= N ;
                    output   := OUT ;
                    vhdl     := { OUT <= not (A and B);
                                  } ;
                    verilog := { assign OUT = ~(A&B);
                                  }
    end


-> "NOR_2"
    if
        mos_p with g=A ; d='1' ; s=N
    and
        mos_p with g=B ; d=N ; s=OUT
    and
```

```
      reduced N
  and
      mos_n with g=A ; d='0' ; s=OUT
  and
      mos_n with g=B ; d='0' ; s=OUT
  then
     build   nor2 with
                  input     := A, B ;
                  internal  := N ;
                  output    := OUT ;
                  vhdl      := { OUT <= not (A or B);
                                 } ;
                  verilog   := { assign OUT = ~(A|B);
                                 }
  end


-> "NAND_4"
   if
       mos_n with g=A ; d='0' ; s=N
   and
       mos_n with g=B ; d=N ; s=M
   and
       reduced N
   and
       mos_n with g=C ; d=M ; s=P
   and
       reduced M
   and
       mos_n with g=D ; d=P ; s=OUT
   and
       reduced P
   and
       mos_p with g=A ; d='1' ; s=OUT
   and
       mos_p with g=B ; d='1' ; s=OUT
   and
       mos_p with g=C ; d='1' ; s=OUT
   and
       mos_p with g=D ; d='1' ; s=OUT
   then
       build   nand4 with
            input       := A, B, C, D ;
            internal    := M,N,P ;
            output      := OUT ;
            vhdl        := { OUT <= not (A and B and C and D);
                                  } ;
            verilog     := { assign OUT = ~(A & B & C & D);
                                  }
   end
```

D.3 Règles de reconnaissance de latches

```
% On reconnait ici les latches composes de 2 inverseurs reboucles formant un
% element memorisant et commande par un reseau switch realisant le set et le reset
% du latch.

-> "LATCH LOOP"
   if
       inverter with
            input    = OUT ;
            output   = PLATCH ;
            field1   = L1 % longueur de grille du mos_n de l'inverseur de maintien
   and
       inverter with
            input    = PLATCH ;
            output   = OUT ;
```

```
          field1  = L2 % longueur de grille du mos_n de l'inverseur direct
   and
       L1 > L2 % on compare les 2 longueurs de grille pour orienter le latch
   then
       build latch_loop with
               input    := PLATCH ;
               output   := OUT ;
               field1   := PLATCH ; % pour construire la liste de sensibilite
               field2   := {OUT <= not PLATCH;
                                         } ; % equation de la sortie
               field5   := PLATCH ; % point memoire du latch
               vhdl     :=
                    {process(PLATCH)
                     begin
                         OUT <= not PLATCH;
                     end process ;
                    }
          and
               update signal PLATCH with tag := latch
        end


% == on recherche un ou plusieurs transistors de mise a '0' du latch

-> "LATCH: reset switch on Latch - R1"
   if
        observed latch_loop with field5 = PLATCH  % point memoire du latch
   and
        mos_n with g=G1 ; d=PLATCH ; s='0'
   then
        build   la_switch with
                     input := S1 ;
                     command := G1 ;
                     output  := PLATCH ;
                     field1  := {G1} ;
                     field2  := {if (G1='1') then PLATCH <= $0; end if;
                                  }
    end


% == on recherche un ou plusieurs transistors de mise a '1' du latch

-> "LATCH: set switch on Latch - R2"
   if
       observed latch_loop with field5 = PLATCH % point memoire du latch
   and
       mos_p with g=G1 ; d=PLATCH ; s='1'
   then
        build   la_switch with
                     input := S1 ;
                     command := G1 ;
                     output  := PLATCH ;
                     field1  := {G1} ;
                     field2  := {if (G1='0') then PLATCH <= $1; end if;
                                  }
    end


% == on cumule les reseaux de mise a 1 et de mise a 0 qui commandent le meme latch

-> "LATCH: latch switch"
    if
        la_switch   with
                input   = I1 ;
                command = C1 ;
                output  = OUT ;
                field1  = F1 ;
                field2  = V1
    and
        la_switch   with
```

```
                 input   = I2 ;
                 command = C2 ;
                 output  = OUT ;
                 field1  = F2 ;
                 field2  = V2
     then
         build   la_switch with
                 input   := I1,I2 ;
                 command := C1,C2 ;
                 output  := OUT ;
                 field1  := {F1,F2} ;     % construction de la liste de sensibilite
                 field2  := V1,V2         % descriptif Vhdl intermediaire

     end


% == on associe le reseau switch a l'element memoire du latch

-> "LATCH: full description"
   if
       latch_loop with
            output      = OUTL ;
            field1      = F1 ;      % liste de sensibilite du latch
            field2      = VL ;      % element d'equation du latch
            field5      = PLATCH    % point memoire du latch
   and
       la_switch with
            input       = INS ;
            command     = CS ;
            output      = PLATCH ;
            field1      = F2 ;      % liste de sensibilite du switch
            field2      = VS        % element d'equation du switch
   then
       build latch_asy  with
            input     := INS ;
            command   := CS ;
            output    := OUTL ;
            field1    := {F1,F2} ;  % liste de sensibilite du latch+switch
            field2    := VS,VL ;    % element d'equation du latch+switch
            field5    := PLATCH ;
            vhdl      := { process(F1,F2)
                           begin
                           VS VL end process;
                            }
     end
```

D.4 Règles de reconnaissance d'échantillonneurs bloqueurs

```
-> "ECHANTILLONNEUR BLOQUEUR R1"
  if
       nand2 with input = A4, B4 ; output = OUT
  and
       N,CK ?= A4,B4
  and
       nand2 with input = A3, B3 ; output = N
  and
       reduced N
  and
       M,OUT ?= A3,B3
  and
       nand2 with input = A1, B1 ; output = M
  and
       IN,Q ?= A1,B1
  and
       nand2 with input = A2, B2 ; output = Q
 and
       reduced Q
```

```
 and
        M,CK ?= A2,B2
and
        reduced M
then
        update signal OUT with tag:=flipflop
 and
        build ech_bloqueur with
             input := IN ;
             command := CK ;
             internal:= M,N,Q ;
             output := OUT ;
             vhdl := {
                  process (CK)
                  begin
                       if(CK='1') then
                            OUT <= not IN ;
                       else
                            OUT <= $1 ;
                       end if ;
                  end process ;
             }
 end
```

D.5 Règles de reconnaissance d'un circuit d'entrée sortie

```
-> "BUFFER: level becg"
   if
        mos_p with g=OUT; d='0'; s=N13
   and
        mos_p with g=I; d=N13; s='1'
   and
        mos_p with g=I; d=OUT; s=N13
   and
        reduced N13
   and
        mos_n with g=OUT; d='1'; s=N12
   and
        mos_n with g=I; d=N12; s='0'
   and
        mos_n with g=I; d=OUT; s=N12
   and
        reduced N12
   then
        build level_becg with
             input := I ;
             output :=OUT ;
             internal := N13,N12;
             vhdl := { OUT <= I ;
                       };
             verilog := { assign OUT = I ;
                       }
   end


-> 'BUFFER: becg_t'
   if
        level_becg with input = I; output = N10
   and
        buf_prot_in with input = I
   and
        trst_1 with input = N10; output = TR ; command = N7,N8
   and
        reduced N10
   and
        inverter with input = VN; output = N7
```

```
    and
        inverter with input = N7; output = N8
    and
        reduced N7,N8
    then
        build becg_t with
             input := I ;
             output := TR ;
             command := VN ;
             internal := N7,N8,N10 ;
             vhdl := { TR <= I when VN = '0' else $Z ;
                       };
             verilog := { assign TR = (VN=0)? I :Z;
             }
        and
             update signal TR with tag := tristate
    end
```

D.6 Règles de reconnaissance de bascules

```
-> "SUB BASCULE DI"
    if
        inverter with
             input = Q1 ;
             output = QB1 ;
             field1 = L
    and
        inverter with
             input = QB1 ;
             output = Q1 ;
             field1 = L
    and
        mos_n with g=DI ; d=Q1 ; s=N1
    and
        mos_n with g=CK ; d=N1 ; s='0'
    and
        reduced N1
    and
        mos_n with g=DIB ; d=QB1 ; s=N2
    and
        mos_n with g=CK ; d=N2 ; s='0'
    and
        reduced N2
    then
             update signal QB1 with tag:=latch
        and
             build sub_basc_di with
                  input :=DI,DIB ;
                  command := CK ;
                  output := Q1,QB1 ;
                  internal := N1, N2 ;
                  vhdl :=
                       { process(CK,DI,DIB,QB1)
                        begin
                            if (CK = '1' and DI = DIB) then QB1 <= $X ; end if ;
                            if (CK = '1') then QB1 <= DI ; end if ;
                       end process ;
                       Q1 <= not QB1 ;
                       }
  end



-> "BASCULE DI"
  if
```

```
        sub_basc_di with input = A,B ; command = CK ; output = C,D
    and
            A,B,C,D = DI,DIB,Q1,QB1
        or
            A,B,C,D = DIB,DI,QB1,Q1
    and
        inverter with input = DI ; output = DIB
    and
        sub_basc_di with input = C,D ; command = CKB ; output = Q2,QB2
    and
        reduced Q1,DIB, QB1,QB2
    and
        inverter with input = Q2 ; output = OUT
    and
        reduced Q2
    and
        inverter with input = CK ; output = CKB
    and
        reduced CKB
    then
            update signal OUT with tag := flipflop
        and
            build basc_di with
                input := DI ;
                output := OUT ;
                command := CK ;
                internal := Q1,QB1,Q2,QB2,DIB,CKB ;
                vhdl := { process(CK)
                          begin
                          if (CK = '0') then OUT <= DI ; end if ;
                          end process ;
                          }
    end


-> "BASCULE DN"
    if
        nor2 with input = I1, I2 ; output = Q1
    and
        I1,I2 ?= SELW,Q2
    and
        nor3 with input = I3,I4,I5 ; output = Q2
    and
        I3,I4,I5 ?= SELR, INIT, Q1
    then
        update signal Q1 with tag:=latch
    and
        build basc_dn with
            output := Q1,Q2 ;
            command := INIT, SELR, SELW ;
            vhdl :=
                { process(INIT, SELR, SELW)
                begin
                if (SELW = '1' and (INIT = '1' or SELR = '1')) then Q1 <= $X ;
                elsif (INIT = '1') then Q1 <= $1 ;
                elsif (SELW = '1') then Q1 <= $0 ;
                elsif (SELR = '1') then Q1 <= $1 ;
                end if ;
                end process ;
                Q2 <= not Q1 ;
                }
    end


-> "BASCULE DNI"
    if
```

```
    basc_dn with output = Q1,Q2; command = INIT,SELR,SELW
and
    inverter with input = Q1 ; output = O1
and
    reduced Q1
and
    inverter with input = Q2 ; output = O2
and
    reduced Q2
then
    update signal O1 with tag:=latch
and
    build basc_dni with
         output := O1, O2 ;
         command := INIT, SELR, SELW ;
         vhdl :=
         { process(INIT, SELR, SELW)
         begin
              if (SELW = ’1’ and (INIT = ’1’ or SELR = ’1’)) then O1 <= $X ;
              elsif (INIT = ’1’) then O1 <= $0 ;
              elsif (SELW = ’1’) then O1 <= $1 ;
              elsif (SELR = ’1’) then O1 <= $0 ;
              end if ;
         end process ;
         O2 <= not O1 ;
         }
end
```

ANNEXE E : Exemples de règles BiCMos

Nous donnons ici quelques exemples de règles permettant reconnaître des portes en technologie BiCmos:

```
-> "BiCMos LOW LEVEL"
   if
       npn with b=DIN ; e=DOUT ; c='1'
   and
       mos_n with g='1' ; d='0' ; s=DOUT
   and
       mos_n with g='1' ; d='0' ; s=DOUTB
   and
       npn with b=DINB ; e=DOUTB ; c='1'
   then
       build lowlev with
            input := DIN, DINB ;
            output := DOUT, DOUTB ;
            vhdl := { DOUT <= DIN ;
                      DOUTB <= DINB ;
                      } ;
            verilog := {  assign DOUT = DIN, DOUTB = DINB ;
                      }
   end

-> "BiCMos NAND 1"
   if
       resistor with e1=OB ; e2='1'
   and
       npn with b='1' ; e=M1 ; c=OB
   and
       npn with b=I1B ; e=ND1 ; c=OB
   and
       npn with b=I1 ; e=ND1 ; c=O
   and
       resistor with e1=O ; e2='1'
   and
       mos_n with d=ND1; g=I2; s=ND2
   and
       reduced ND1
   and
       mos_p with d=M1; g=I2; s=ND2
   and
       reduced M1
   and
       npn with b=VREF ; e=M2 ; c=ND2
   and
       reduced ND2
   and
       resistor with e1=M2 ; e2='0'
   and
       reduced M2
   then
       build nandbc with
            input := I1,I1B,I2;
            output := O,OB;
            command := VREF; % tension de reference
            internal := ND1, M1, ND2, M2;
            vhdl := { O <= not (I1 and I2) ;
                      OB <= not O ;
```

```
                } ;
           verilog := { assign O = ~ (I1 & I2), OB = ~ O ;
                }
   end

-> "BiCMos NAND 2"
   if
       resistor with e1=OB ; e2='1'
   and
       npn with b=I1B ; e=ND1 ; c=OB
   and
       npn with b=I1 ; e=ND1 ; c=O
   and
       resistor with e1=O ; e2='1'
   and
       npn with c=ND1; b=I2; e=ND2
   and
       reduced ND1
   and
       npn with c=OB; b=I2B; e=ND2
   and
       npn with b=VREF ; e=M2 ; c=ND2
   and
       reduced ND2
   and
       resistor with e1=M2 ; e2='0'
   and
       reduced M2
   then
       build nandbc with
            input := I1,I1B,I2,I2B;
            output := O,OB;
            command := VREF; % tension de reference
            internal := ND1, ND2, M2;
            vhdl := { O <= not (I1 and I2) ;
                      OB <= not O ;
                      } ;
            verilog := { assign O = ~ (I1 & I2), OB = ~ O ;
                      }
   end

-> "BiCMos BOOST"
   if
       resistor with e1=DOUTB ; e2='1'
   and
       npn with b=DIN ; e=ND1 ; c=DOUTB
   and
       npn with b=DINB ; e=ND1 ; c=DOUT
   and
       resistor with e1=DOUT ; e2='1'
   and
       npn with b=VREF ; e=M2 ; c=ND1
   and
       reduced ND1
   and
       resistor with e1=M2 ; e2='0'
   and
       reduced M2
 then
       build boost with
            input := DIN,DINB;
            output := DOUT,DOUTB;
            command := VREF; % tension de reference
            internal := ND1, M2;
            vhdl := { DOUT <= not DINB ;
                      DOUTB <= not DIN ;
                      } ;
            verilog := { assign DOUT = ~DINB, DOUTB = ~DIN ;
                      }
   end
```

```
-> "BiCMos BOOST2"
   if
       mos_p with g='1' ; d=DOUTB ; s='1'
   and
       npn with b=DIN ; e=ND1 ; c=DOUTB
   and
       npn with b=DINB ; e=ND1 ; c=DOUT
   and
       mos_p with g='1' ; d=DOUT ; s='1'
   and
       npn with b=VREF ; e=M2 ; c=ND1
   and
       reduced ND1
   and
       mos_p with g='1' ; d=M2 ; s='0'
   and
       reduced M2
   then
       build boost2 with
            input := DIN,DINB;
            output := DOUT,DOUTB;
            command := VREF; % tension de reference
            internal := ND1, M2;
            vhdl := { DOUT <= not DINB ;
                      DOUTB <= not DIN ;
                      } ;
            verilog := { assign DOUT = ~DINB, DOUTB = ~DIN ;
                      }
   end
```